A Survey of Software Development Approaches Addressing Dependability

Sadaf Mustafiz and Jörg Kienzle

School of Computer Science, McGill University, Montreal, Quebec, Canada
sadaf@cs.mcgill.ca, joerg.kienzle@mcgill.ca

Abstract. Current mainstream software engineering methods rarely consider dependability issues in the requirements engineering and analysis stage. If at all, they only address it much later in the development cycle. Concurrent, distributed, or heterogeneous applications, however, are often deployed in increasingly complex environments. Such systems, to be dependable and to provide highly available services, have to be able to cope with abnormal situations or failures of underlying components. This paper presents an overview of the software development approaches that address dependability requirements and other non-functional requirements like timeliness, adaptability and quality of service. Software development methods, frameworks, middleware, and other proposed approaches that integrate the concern of fault tolerance into the early software development stages have been studied. The paper concludes with a comparison of the various approaches based on several criteria.

1 Introduction

Due to the increasing responsibilities and number of requirements that modern applications have to address, the average complexity of software systems is growing. Elaborate user interfaces, multi-media features or interaction with real-time devices require software to respond promptly and reliably. Situations such as node failures, network partitions, overloaded resources, irregular load, component failures, heterogeneity, abnormal behavior of subsystems or the environment, and also software design faults must be handled in order to provide highly available services.

Surprisingly enough, dependability and fault tolerance are not addressed by current mainstream software engineering methods. In general, dependability and fault tolerance are considered "non-functional" requirements, and therefore considered too late during the development of an application. Ad-hoc solutions that try increase dependability by adding fault tolerance once the main functionality of the system has been implemented often result in complex system structure, hard-to-maintain code and poor performance.

This paper summarizes the results of a survey of specialized software development methods, frameworks, middleware, software architectures, and other approaches that assist developers in producing dependable software. Dependability can be attained by fault prevention, fault removal, fault tolerance[1] and fault forecasting [3]. The investi-

[1] An overview of software fault tolerance techniques can be found in [36].

N. Guelfi et al. (Eds.): FIDJI 2004, LNCS 3409, pp. 78–90, 2005.

gation focuses on the non-functional requirements that are part of dependability, i.e. availability, reliability, safety, security, and maintainability [2], but also timeliness, which includes responsiveness, orderliness, freshness, temporal predictability and temporal controllability [29]. Adaptability, i.e. the need to remain functional even when modifications are carried out in the system, is also considered. In addition the review includes, for each approach, its application environment, the covered failure domain, and what fault tolerance techniques, if any, have been incorporated into the process.

The approaches presented in this paper are structured into three categories. Section 2 reviews *software development methods*. Section 3 discusses *software architectures*, *middlewares* and *frameworks*. Section 4 presents other approaches that propose notations or consider elements that help in the development of dependable systems. Finally, Section 5 presents a comparison of the surveyed approaches.

2 Software Development Methods

Software development methods define a step-by-step process that leads a developer from the elaboration of an initial requirements document, over analysis, architecture and design phases through to the final implementation.

2.1 HRT-HOOD

HOOD (Hierarchical Object-Oriented Design) [32] is an architectural design method developed by the European Space Agency in 1987, with Ada as the target programming language. HRT-HOOD (Hard Real-Time HOOD) [9] was later developed to addresses issues of timeliness in the early stages of the development process, with explicit support for common hard real-time abstractions. HRT-HOOD introduces cyclic and sporadic type objects to take into account timing properties of real-time systems. These objects are annotated with information about the period of execution, minimum arrival time, offset times, deadlines, budget times, worst-case execution time (WCET), and importance. HRT-HOOD uses exceptions to handle timing faults. The coding language should have support available to program recovery handling. In cases of sporadic objects, method invocation should be monitored in order to prevent early execution or overly high invocation frequency. The method does not provide fault-tolerance support or ways of identifying the mentioned non-functional requirements, but focuses on how to integrate them into the design phase. The STOOD tool supports real-time software development based on the HOOD (version 4) and HRT-HOOD method.

2.2 The OOHARTS Approach

Object-Oriented Hard Real Time System (OOHARTS) [35] is a process for developing dependable hard real-time systems. It is based on UML and the hard real-time constructs of HRT-HOOD. Various extensions to UML are proposed, e.g. stereotypes such as <<cyclic>>, <<aperiodic>>, <<protected>>, <<passive>>, and <<environ-

ment>> to describe different real-time objects. A special form of UML state diagram called Object Behavior Chart (OBC) is used to define object behavior. It provides means for representing timing constraints like deadline and period. The UML concurrency attribute, which can be sequential, guarded, or concurrent, is extended to include <<mutex>> (mutual exclusion), <<wer>> (write execution request), and <<rer>> (read execution request).

The OOHARTS method follows the traditional software development phases. Both functional and non-functional requirements are specified in the requirements definition phase. It introduces an additional phase in the HRT-HOOD software development life cycle, hard-real time analysis, which provides a framework for defining the structure and behavior of hard real-time systems using UML and the new extensions defined [35].

2.3 Extension of the Catalysis Method

In [31], a fault-tolerant software architecture for component-based systems based on the *idealized fault-tolerant component (IFTC)* [38][44] is proposed. The architecture can handle software faults, providing higher levels of dependability.

Based on this work, [45] proposes a way of incorporating exception handling and error recovery into the Catalysis [17] process. At the requirements level, exceptional behavior, which includes recovery scenario and failure scenario, is added to use-case specifications in a formal manner. The system is structured with IFTC and the propagation of exceptions is clearly modeled. In the next phase, collaborations are derived from the use-cases. Pre- and post conditions are mapped to actions, which include refinements of the defined exceptions. A template is used to describe the collaboration, and class hierarchies of normal and exceptional behavior are produced. Following the design, ways to move on to implementation are suggested.

2.4 The KAOS Approach

The KAOS framework [22] provides a goal-oriented approach for requirements modeling, specification, and analysis, which address both functional and non-functional requirements. Three types of non-functional goals are considered: quality-of-service, development, and architectural constraints. These goals address the need for safety, security, usability, performance, interoperability, accuracy, maintainability, reusability, and issues of distribution, and physical and logical organization. Exceptional behavior, defined as *obstacles*, is also addressed during requirements engineering [48]. Goals and obstacles are expressed in a formal language. Based on this framework, Lamsweerde has proposed a method in [49] for deriving the software architecture from the requirements. To begin with, the software specification is developed from the requirements, which is then used to build the architectural design. The design evolves with recursive refinements, which consider constraints and non-functional goals. The refinement is pattern-based; Figure 1 for example shows how to introduce reliable communication by means of replication. The KAOS approach is supported by the GRAIL tool [22].

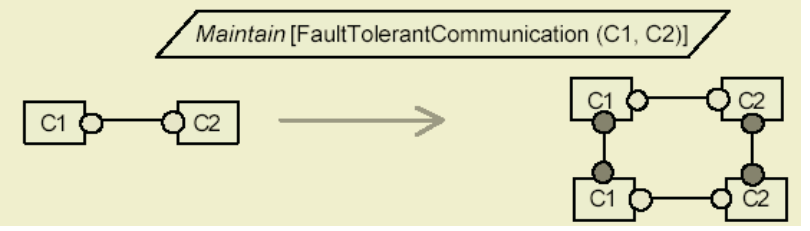

Fig. 1. Architectural refinement pattern for a QoS goal [49]

2.5 The B Method

The B formal method [1] covers the development process from the specification to the implementation phase and is based on a mathematical model of set theory and first order logic. The B method mainly comprises two activities: writing formal texts and proving the texts. The process evolves from specification to coding using a series of refinements. The methodology has been used for developing error-free software for critical systems by focusing on traceability of safety-related constraints [10]. The method is supported by the tools B-Toolkit and Atelier-B. Modeling tools (B4free), editors, and parsers are also available for B.

3 Fault Tolerance Frameworks and Middleware

This section reviews software architectures, middlewares and frameworks. *Software architectures* do not offer any methodological support, but instead provide a structure (usually hardware design) based on which applications can be built. *Middleware* is software that is used to integrate heterogeneous software applications or products efficiently and reliably in a distributed computing environment. It is the middle layer between the application program and the platform and provides abstractions necessary for interfacing. A *framework* is an environment composed of software components that can be tailored according to the needs of the application being developed. Finally, a *middleware framework* is a structure that offers users multiple middleware styles that can be customized for application as well as device constraints.

3.1 TARDIS

The **Timely and Reliable Distributed Information Systems (TARDIS)** project [8] was initiated in 1990, and is targeted towards avionics, process control, military, and safety critical applications. The proposed framework addresses non-functional requirements (dependability, timeliness, and adaptability), and implementation constraints from the early stages of software development. In the architectural design phase, issues of choices are addressed, for example, between replication and dynamic reconfiguration for improving reliability. The framework is generic, and does not impose any software design methods or languages on the developer.

The initial proposal, however, was not completed. The project continued with focus on development of real-time systems. [28] discusses the architectural design of non-functional requirements related to real-time issues using the specification language *Z* and RTL (Real-Time Logic). Detailed design using TARDIS is considered in [7][28]. According to [28], the TARDIS framework can also be applied to the design of systems where non-functional requirements like reliability, security, safety, fault tolerance, and system reconfiguration need to be satisfied.

3.2 TIRAN

TaIlorable fault toleRANce frameworks for embedded applications (TIRAN) [20] is a European Strategic Program for Research in Information Technology (ESPRIT) project completed in October 2000. The primary goal of the project was to develop a software framework to provide fault tolerant capabilities to embedded automation systems. The framework, to reduce development costs, aims to solve problems in fault-affected applications by considering error detection, isolation and recovery, reconfiguration and graceful degradation. It considers physical and design faults in the permanent, temporary omission and byzantine failure domains.

The framework provides a library of basic tools implementing fault tolerance mechanisms like watchdog, distributed memory, local voter, output delay, stable memory, distributed synchronization and time-out management. A control backbone, which functions as a middleware, extracts information about the application's topology, its progress and its status. It maintains this information in a replicated database and coordinates fault tolerance actions at runtime via user-defined recovery strategies. A domain-specific language named ARIEL was developed as part of the project to configure the basic tools and to specify the recovery strategies. More information about ARIEL can be found in [20].

TIRAN provides the users of the framework with a methodology for collecting, specifying, and validating fault tolerance requirements, with a characterization of framework elements, and guidelines for using the framework. The specification of fault tolerance is based primarily on UML package diagrams and class diagrams, and TRIO (Tempo Reale ImplicitO) temporal logic, a language that has been developed by ENEL (Italy's largest power distributer) specifically for real-time systems. The use of the methodology has been experimented on a pilot application, a primary substation automation system, and is discussed in [18][25].

3.3 DepAuDE

Dependability for embedded **Au**tomation systems in **D**ynamic **E**nvironments with intra-site and inter-site distribution aspects (DepAuDE) [24] is an IST (Information Society Technologies) project partially based on TIRAN completed in 2003. It has been developed primarily for two target application areas: monitoring/control of energy transport and distribution, and distributed embedded systems.

The DepAuDE framework provides "a methodology and an architecture to ensure dependability for non-safety critical, distributed, embedded automation systems with both IP (inter-site) and dedicated (intra-site) connections" [21]. The methodology support is similar to that outlined in TIRAN, but includes inter-site communication

features for specification, validation, and modeling of requirements. It also adds support for quality-of-service (QoS) levels. Furthermore, the DepAuDE framework has been applied on the pilot applications to evaluate and show the feasibility of the framework.

3.4 EFTOS: FT Approach to Embedded Supercomputing

Embedded Fault-Tolerant Supercomputing (EFTOS) is an ESPRIT project completed in 1998, targeted towards industrial process-control, real-time applications, and embedded systems. It aims to provide a middleware framework to implement fault-tolerance to make embedded supercomputing applications more dependable.

Similar to TIRAN, EFTOS also follows a layered approach comprising of basic fault-tolerance tools and mechanisms, a backbone, and a high-level recovery language for specifying recovery strategies [23]. The FT tools provided include a watchdog timer, a trap handler for exception handling, an atomic action tool, assertions, and a distributed voting mechanism.

3.5 Middleware Architectures

DCE (Distributed Computing Environment), **DCOM** (Distributed Component Object Model), **Java RMI** (Remote Method Invocation), and **CORBA** (Common Object Request Broker Architecture) are general middleware that have limited fault tolerance support, like mechanisms for replication and time-outs [46]. TAO (The ACE ORB) implementation of CORBA supports fixed-priority real-time scheduling. Electra, another CORBA implementation, provides fault-tolerance with object replication. Real-time CORBA 1.0 supports QoS with standard policies and techniques [46]. CORBA also defines a transaction service (OTS).

Chameleon is an adaptive infrastructure, which supports multiple fault-tolerance strategies in a networked environment. Chameleon uses reliable agents that support user-specified levels of fault-tolerance. It considers satisfying dependability in terms of availability. With some additional features, chameleon can be used for real-time applications [19][4].

ROAFTS is a middleware architecture providing real-time object-oriented adaptive fault-tolerance support. ROAFTS offers fault-tolerance schemes that can be applied to both process-structured and object-structured distributed real-time (RT) applications. These schemes are used to tolerate processor faults, communication link faults, interconnection network faults, and application software faults. ROAFTS is meant for implementation on COTS (Commercial Off-The-Shelf) and guarantees RT fault-tolerance when required [19][37].

FRIENDS (Flexible and Reusable Implementation Environment for your Next Dependable System) is a software architecture, which provides fault-tolerance and limited security support. It is built on subsystems and libraries of meta-objects. There is a fault-tolerance sub-system that incorporates fault-tolerance mechanisms for error detection, failure detectors, replication, reconfiguration, and stable storage. It does not provide specific support for real-time and quality-of-service requirements [19][27].

AQuA (Adaptive Quality of Service for Availability) is an adaptive architecture for building dependable distributed systems. Fault tolerance is provided by Proteus, a

dependability manager integrated into the architecture. Fault tolerance support is given to CORBA applications with replication of objects, and different levels of desired dependability and quality-of-service are provided. AQuA is capable of handling crash failures, value faults, and time faults. It incorporates means for detecting errors, treating faults, and reliable communication [19][14].

3.6 Software Architectures

Some architectures considering fault-tolerance and other dependability attributes worth mentioning are discussed below. Because of space constraints, it was not possible to describe them in details.

Delta-4 [5], an ESPRIT project, provides an open architecture for development of dependable distributed real-time systems. Delta-4 tolerates hardware failures with hardware and software redundancy, and also supports active and passive replication of software components residing in homogeneous computers. Voting mechanisms and systematic and periodic strategies for check-pointing are provided.

MAFTIA (Malicious and Accidental Fault Tolerance for Internet Applications) is a European Union project completed in 2003 and is said to be the first project to address the need to tolerate malicious and accidental faults in large-scale distributed systems [39].

GUARDS (Generic Upgradeable Architectures for Real-Time Dependable Systems) [41] is an ESPRIT project aiming to provide methods, techniques, and tools for design, implementation, and validation support in safety-critical real-time systems.

MARS (Maintainable Real-Time System) [43] is an architecture specialized for time-triggered applications, and addresses fault-tolerance with active replication means and other hardware FT measures to satisfy hard real-time requirements.

3.7 Other Frameworks

This section presents some software frameworks that were came about to aid in development of dependable systems.

HIDE (High-level Integrated Design Environment for Dependability), an ESPRIT project, addressed the need for early validation of UML-based design [6]. In [12], Chin proposes an approach, as part of HIDE, to extend UML towards a useful OO-Language for modeling dependability features. It provides abstractions to incorporate common dependability requirements in the UML model.

Aurora Management Workbench provides a software framework for developing reliable, scalable, and configurable distributed applications [10].

DOORS is a framework developed to provide support for building fault-tolerant applications in CORBA [30].

4 Other Related Work

This section reviews work that addresses the need to integrate dependability requirements in the model specifications.

4.1 Developing Safety Critical Systems with UML

It is crucial when developing safety-critical systems to consider means to achieve the highest level of dependability. In [33], a technique is presented which is based on using the UML extension mechanisms to incorporate safety requirements in a UML model. The mechanisms consider crash/performance failures and value failures which may cause message loss, delay, or corruption. For example, the stereotype <<risk>> can be used to describe a risk that arises in the physical level with the tag {failure} and <<error handling>> provides an object for handling errors in the subsystem level and is associated with the tag {error object}. The approach also considers analyzing the UML model with a prototypical XMI-based tool to check if it satisfies the requirements [33]. The approach considers non-functional requirements during the design phase in terms of safety. Jürgens has also proposed using UML to develop security-critical systems in [34]. Previously, he has also, in collaboration with others, described some approaches for systems development using UML, which consider various criticality requirements.

4.2 A Framework for Integrating Non-functional Requirements into Conceptual Models

In [15], an interesting approach is presented addressing the need to capture non-functional requirements (NFR) at the early stages of development, by integrating NFR into conceptual models, specifically into the entity-relationship (ER) and object-oriented (OO) models. The proposed method describes the use of the LEL (Language Extended Lexicon) [15], and a NFR taxonomy to elicit the requirements. A comprehensive taxonomy of NFR can be found in [13]. A LEL-NFR tool is required that captures terminologies relevant to the target field, referred to as the UoD (Universe of Discourse). This tool along with the NFR taxonomy is used to derive the NFR knowledge-base for a particular domain. These NFR are decomposed and represented in graphs which a slight variants of Chung's NFR graphs [13]. Finally, the NFR are integrated into the conceptual models. In ER models, a NFR is shown in a rectangle with the UoD labeled over it, and connected to the relevant entity or relationship. In the OO model, NFR are added to class diagram by attaching two rectangles to the right bottom of the class with the UoD name in one and the NFR name in the other.

In [16], this approach has been applied to UML, starting from use cases to class diagrams, sequence diagrams, and collaboration diagrams.

5 Survey Results

This section shows a comparison of the major approaches discussed in this paper based on some important non-functional requirements. The requirements considered have been introduced in Section 1: *dependability*, *timeliness*, *adaptability*, and *quality-of-service (QoS)*. Dependability refers to availability, reliability, safety, confidentiality, integrity, and maintainability [2]. Availability and reliability can be together classified as "avoidance or minimization of service outages" [2]. Also, a specializa-

tion of availability and integrity with respect to authorization, and confidentiality can be grouped together as the *security* requirement [2]. The approaches have been evaluated based on the requirements that are satisfied or taken into consideration, and a comparison is illustrated in Table 1.

Table 1. Comparison Based on NFR

	Availability/ Reliability	Safety	Security	Maintainability	Timeliness	Adaptability	QOS	Comments
HOOD	×	×	×	✓	✓	×	×	limited maintainability (only exception handling); timeliness (only SRT);
HRT-HOOD	✓	✓	×	✓	✓	✓	×	limited maintainability (only exception handling and maybe replication); adaptability (mode changes);
TIRAN	✓	✓	×	✓	✓	✓	×	assumes reliable communication; safety (only by criticality level)
DepAuDE	✓	✓	✓	✓	✓	✓	✓	considers intra- and inter-site communication; safety (only by criticality level)
TARDIS	✓	✓	✓	✓	✓	✓	×	not targeted to specific NFR – open framework
OOHARTS	✓	✓	×	✓	✓	✓	×	limited maintainability (exceptions); adaptability (mode changes);
EFTOS	✓	×	✓	✓	✓	✓	×	security (integrity); timeliness (esp. SRT);
DELTA-4	✓	×	✓	✓	✓	×	×	user-specified level of dependability; maintainability (only replication);
Chameleon	✓	×	×	✓	×	✓	×	supports different levels of availability requirements; adaptability (mode changes);
ROAFTS	✓	✓	×	✓	✓	✓	×	guarantees RT FT; adaptability (mode changes); survivability;
FRIENDS	✓	✓	✓	✓	×	×	×	security (communication);
AQuA	✓	×	×	✓	×	✓	✓	user-specified level of availability;

Table 2 presents a comparison of the fault-tolerance support in each approach and is classified based on the failure domain, error processing, and fault treatment support. The main techniques considered in each approach have also been mentioned.

6 Conclusion

This paper presented an assortment of methods, frameworks, middleware, architectures, and other development techniques that address dependability, timeliness, adaptability, or other QoS requirements. Due to space reasons, none of the addressed approaches have been discussed in much detail. The interested reader is encouraged to consult the detailed review given in [40].

The software development methods, HRT-HOOD and OOHARTS, consider real-time issues in isolation. The KAOS and B formal methods address dependability issues, and present a high-level approach for specifying requirements and deriving the

Table 2. Fault Tolerance Support

	Failure Domain		Error Processing	Fault Treatment	Means
	Value	Timing			
HOOD	×	×	detection	no support	exception processing, deadlock avoidance techniques
HRT-HOOD	×	✓	detection	reconfiguration	exception processing, replication
TIRAN	✓	✓	detection, localization, recovery	diagnosis, masking confinement, dynamic reconfiguration, graceful degradation	exception handling, design diversity, stable memory, watchdog, local voter, distributed synchronization, time-out, standby sparing, recovery blocks, NMR
	computing failures only				
DepAuDE	✓	✓	same as above	same as above	same as above & group communication
TARDIS	✓	✓	detection, recovery	diagnosis, isolation, confinement, reconfiguration	timeout, HW/SW repair/replacement, NMR, failure messages
OOHARTS	×	✓	detection	reconfiguration	exceptions, timeout, deadlock avoidance techniques
EFTOS	✓	✓	detection, isolation, recovery	masking, fault-tolerance	exception handling, watchdog, atomic actions, distributed voting, recovery language
DELTA-4	✓	✓	detection, recovery	reconfiguration, fault-tolerance	active/passive/leader-follower replication, voting, timeout
Chameleon	✓	✓	detection, recovery	masking, system reconfiguration, failure recovery	reliable agents, TMR (H/W), checkpoints, voting (distributed & majority)
ROAFTS	✓	✓	detection, recovery	fault-tolerance	watchdog timer, predictable communication, process scheduling, backward recovery (SRT), forward recovery (HRT), recovery blocks; active replication
FRIENDS	physical crash failures only		detection, recovery	reconfiguration, fault-tolerance	leader-follower-replication, stable storage, primary backup, failure suspectors, group communication
AQuA	✓	✓	detection, recovery	fault-tolerance, system reconfiguration	active/passive replication & degree, voting, monitors, group communication

design based on refinements. The frameworks TIRAN and DepAuDE are two significant contributions to the development of dependable systems, but cater to a specific domain. The TARDIS project provides a general framework that addresses various

non-functional requirements, but does not define a step-by-step development process. Similarly, middleware infrastructures like EFTOS and ROAFTS provide FT tools that the users can adapt according to their needs. Software architectures, like DELTA-4, Chameleon, FRIENDS, and AQuA, attempt to provide hardware fault tolerance by supporting techniques like replication, but they do not provide guidelines to the user for making design decisions during software development. Several other approaches propose extension mechanisms to integrate non-functional requirements into design models, and stereotypes have been defined that add dependability-related notions to UML. However, these approaches concentrate on one particular phase, and do not provide the necessary continuity throughout the development life cycle. In short, the survey shows that there is a lot more work to be done to make dependability, and in particular fault tolerance, an integral part of software development.

References

1. Abriel, J.-R.: *The B-book*, Cambridge University Press, 1996.
2. Avizienis, A., Laprie, J.-C., et al.: "Dependability of computer systems: Fundamental concepts, terminology, and examples", in *Proc. 3rd IEEE Information Survivability Workshop (ISW-2000)*, Boston, Massachusetts, USA, October 24-26, 2000, pp. 7-12.
3. Avizienis, A., Laprie, J.-C., Randell, B.: "Fundamental Concepts of Dependability", CS-TR: 739, Department of Computing Science, University of Newcastle, 2001.
4. Bagchi, S, Whisnant, K., et al.: "Error Detection and Recovery in Chameleon", Center for Reliable and High-Performance Computing, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Nov 98, presentation .
5. Barrett, P.A.: "Delta-4: An open architecture for dependable systems", in *IEE Colloquium on Safety Critical Distributed Systems*, 1993, pp. 2/1-2/7.
6. Bondavalli, A., Cin, M.D., et al.: "Dependability Analysis in the Early Phases of UML Based System Design", *International Journal of Computer Systems - Science & Engineering*, Vol. 16 No. 5, Sep 2001, pp. 265-275.
7. Burns, A., Lister, A. M., McDermid: "TARDIS: an architectural framework for timely and reliable distributed information systems", in *Proc. Sixth Australian Software Engineering Conf.*, Sydney, Australia, July 1991, pp. 1-15.
8. Burns, A., Lister, A. M.: "A framework for building dependable systems", *The Computer Journal*, Vol. 34 No. 2, April 1991, pp. 73- 181.
9. Burns, A., Wellings, A.: *HRT-HOOD: a structured design method for hard real-time Ada systems*, Elsevier Science BV, 1995, ISBN 0-444-82164-3.
10. Buskens. R, Siddiqui A., et al.: "Aurora Management Workbench", Bell laboratories, 2003, http://www.bell-labs.com/project/aurora.
11. Carnot, M., DaSilva, C., et al.: "Error-free software development for critical systems using the B-Methodology", in *Proc. of 3rd International Symposium on Software Reliability Engineering*, Oct 1992, pp. 274-281.
12. Chin, M.D.: "Extending UML towards a useful OO-language for modeling dependability features", in *the Ninth IEEE Workshop on Object-Oriented Dependable Real-Time Systems*, October 2003.
13. Chung, L., Nixon, B.A., et al.: *Non-functional Requirements in Software Engineering*, Kluwer Academic Publishers, 2000.
14. Cukier, M., Ren, J., et al.: "AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects", in *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems (SRDS.98),* Indiana, SA, October 20-23, 1998, pp. 245-253.

15. Cysneiros, L.M., Leite, J.C.S.P, et al.: "A Framework for Integrating Non-Functional Requirements into Conceptual Models", *Requirements Engineering Journal*, Vol. 6, Issue 2, Apr. 2001, pp. 97-115.
16. Cysneiros, L.M., Leite, J.C.S.P: "Non-Functional Requirements: From Elicitation to Conceptual Model", *IEEE Transactions on Software Engineering*, May 2004.
17. D'Souza, D., Wills, A. C.: *Objects, components, and frameworks with UML: The Catalysis Approach*, Addison-Wesley: Reading, MA, USA, 1998.
18. D1.1 - Requirement specification V2, TIRAN Project Deliverable, October 1999, confidential.
19. D2.1 and D2.2: Updated Investigation, evaluation, and selection, DepAuDE Deliverable, 2002.
20. D7.9 – Project Final Report, TIRAN Project Deliverable, October 2000, confidential.
21. D8.6: Final Report, DepAuDE Deliverable, 2003.
22. Darimont, R., Delor, E., et al.: "GRAIL/KAOS: An Environment for Goal-Driven Requirements Engineering", in *Proc of. ICSE'98 - 20th Intl. Conf .on Software Engineering,* Kyoto, Vol. 2, April 1998, pp. 58-62.
23. Deconinck, G., De Florio, V., et al.: "The EFTOS approach to dependability in embedded supercomputing", *IEEE Transactions on Reliability*, Vol. 51, Mar. 2002, pp. 76–90.
24. DepAuDE project website, April 22, 2004, http://www.depaude.org/
25. Dondossola, G., Botti, O.: "System fault tolerance specification: proposal of a method combining semi-formal and formal approaches", in *Proc. of Int. Conf. FASE2000, part of ETAPS2000 - The European Joint Conferences on Theory and Practice of Software*, Berlin, D, March 2000, LNCS, No. 1783, Springer-Verlag, Berlin, Heidelberg, New York, 2000, pp. 82-96.
26. European Dependability Initiative: Inventory of EC Funded Projects in the area of Dependability, Issue 2.2, 11 January 2000.
27. Fabre, J.-C., Pérennou, T.: "A Metaobject Architecture for Fault Tolerant Distributed Systems: The FRIENDS Approach", *IEEE Trans. on Computers*, Jan. 1998, pp. 78-95.
28. Fidge, C.J., Lister, A.M.: "A disciplined approach to real-time systems design", *Information and Software Technology*, Vol. 34 No. 9, September 1992, pp. 603-610.
29. Fidge, C.J., Lister, A.M.: "The challenges of non-functional computing requirements", in *Seventh Australian Software Engineering Conference (ASWEC'93)*, Sydney, September 1993, pp. 77-84.
30. Gokhale, A., Natarajan, B., et al.: "DOORS: Towards high-performance fault-tolerant CORBA", in *Proc. 2nd Intl. Symp. Distributed Objects and Applications (DOA '00)*, Sept. 2000.
31. Guerra, P. A. de C., Rubira, C., et al.: "Fault-Tolerant Software Architecture for Component-Based Systems", in R. de Lemos, C. Gacek, A. Romanovsky (Eds). Architecting Dependable Systems, *LNCS* 2677, Springer, 2003, pp. 129-149.
32. HOOD Reference Manual, Issue 4, 1995. Available at ftp://ftp.estec.esa.nl/pub/wm/wme/HOOD/HRM4.tar.gz.
33. Jürgens J.: "Developing safety-critical systems with UML", in *Proc. UML 2003 Conference*, *LNCS* 2863, Springer-Verlag 2003, pp. 360-372, San Francisco, California, USA.
34. Jürgens, J.: *Secure Systems Development with UML*, Springer-Verlag, 2004 (to be published).
35. Kabous, L., Nebel, W.: "Modeling Hard Real Time Systems with UML The OOHARTS Approach", in *Proc. UML'99 Conference, LNCS* 1723, pp. 339-355, Springer-Verlag, 1999.
36. Kienzle, J.: "Software Fault Tolerance: An Overview", in *Ada-Europe '2003*, *LNCS* 2655, Springer-Verlag, 2003, pp. 45-67.

37. Kim, K.H.: "ROAFTS: A Middleware Architecture for Real-time Object-oriented Adaptive Fault Tolerance Support", in *Proc. of IEEE CS 1998 HASE Symp.*, Washington, D.C., Nov. 1998, pp. 50-57.
38. Lee, P.A., Anderson, T.: "Fault Tolerance - Principles and Practice", *Dependable Computing and Fault-Tolerant Systems*, Springer Verlag, 2nd ed., 1990.
39. MAFTIA project website, http://www.newcastle.research.ec.org/maftia/.
40. Mustafiz, S.: "Addressing Fault Tolerance in Software Development: A Comparative Study", M.Sc. Thesis, School of Computer Science, McGill University, June 2004.
41. Powell, D., Arlat, J., et al.: "GUARDS: A generic upgradable architecture for real-time dependable systems", in *IEEE Trans. Parallel and Distributed Syst.*, Vol. 10, June 1999, pp. 580–597.
42. Pullum, L.L.: *Software Fault Tolerance Techniques and Implementation*, Artech House, Inc., Boston, 2001.
43. Randell, B., Laprie, J.-C., et al. : ESPRIT Basic Research Series: Predictably Dependable Computing Systems, Springer-Verlag, 1995.
44. Randell, B., Xu, J.: *The Evolution of the Recovery Block Concept*, Chapter 1, pp. 1 – 21, in Lyu, M. R. (Ed.): *Software Fault Tolerance*, John Wiley & Sons, 1995.
45. Rubira, C.M.F., de Lemos, R., et al.: "Exception handling in the development of dependable component-based systems", in *Software – Practice and Experience*, 2004. To appear.
46. Tirtea, R., Deconinck, G.: "A Survey of Middleware and its Support for Fault Tolerance", in *Proc. 6th Int. Conf. Engineering of Modern Electric Systems (EMES-2001)*, Felix-Spa, Romania, May 24-26, 2001, 6 pages.
47. UML Revision Task Force. OMG UML Specification v. 1.5. OMG Document ad/03-03-01. Available at http://www.uml.org, 2003.
48. van Lamsweerde, A.: "Building Formal Requirements Models for Reliable Software", in *Proc. of 6th International Conference on Reliable Software Technologies, Ada-Europe 2001*, *LNCS* 2043, Springer-Verlag, 2001.
49. van Lamsweerde, A.: "From System Goals to Software Architecture", in *Formal Methods for Software Architectures*, M. Bernardo & P. Inverardi (eds), LNCS 2804, Springer-Verlag, 2003, pp. 25-43.
50. Verentziotis, E., Varvarigou, T., et al.: "Fault tolerant supercomputing: a software approach", *International Journal of Computer Research*, Vol. 10, No. 3, Nova Scotia Publishers Inc., 2001, pp. 401-413.