FORMAL TECHNIQUES IN REAL-TIME AND FAULT-TOLERANT SYSTEMS

Edited by
Jan Vytopil

KLUWER ACADEMIC PUBLISHERS

FORMAL TECHNIQUES IN REAL-TIME AND FAULT-TOLERANT SYSTEMS

edited
by

Jan Vytopil
BSO/Origin
University of Nijmegen

KLUWER ACADEMIC PUBLISHERS
Boston * Dordrecht * London

Distributors for North America:
Kluwer Academic Publishers
101 Philip Drive
Assinippi Park
Norwell, Massachusetts 02061 USA

Distributors for all other countries:
Kluwer Academic Publishers Group
Distribution Centre
Post Office Box 322
3300 AH Dordrecht, THE NETHERLANDS

Library of Congress Cataloging-in-Publication Data

Formal techniques in real-time and fault-tolerant systems / edited by
Jan Vytopil.
p. cm. -- (Kluwer international series in engineering and
computer science ; 221. Real-time systems)
Includes bibliographical references and index.
ISBN 0-7923-9332-5
1. Real-time data processing. 2. Fault-tolerant computing.
I. Vytopil, J. (Jan), 1947- . II. Series: Kluwer international
series in engineering and computer science ; SECS 221. III. Series:
Kluwer international series in engineering and computer science.
Real-time systems.
QA76.54.F65 1993
004'.33--dc20 93-16676
CIP

FORMAL TECHNIQUES IN REAL-TIME AND FAULT-TOLERANT SYSTEMS

CONTRIBUTORS

Jos Coenen
Department of Mathematics
and Computing Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven
The Netherlands
E-mail: wsinjosc@win.tue.nl

Hans A. Hansson
Swedish Institute of Computer Science
Box 1263, S-164 28 Kista
and
Department of Computer Science
Uppsala University
Sweden
E-mail: hansh@sics.se

Jozef Hooman
Department of Mathematics
and Computing Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven
The Netherlands
E-mail: wsinjh@win.tue.nl

Mathai Joseph
Department of Computer Science
University of Warwick
Coventry CV4 7AL, Warwick
United Kingdom
E-mail: mathai@dcs.warwick.ac.uk

Zhiming Liu
Department of Computer Science
University of Warwick
Coventry CV4 7AL, Warwick
United Kingdom
E-mail: liu@dcs.warwick.ac.uk

Jan Peleska
Deutsche System–Technik GmbH
Edisonstraße 3, D-2300 Kiel 14
Federal Republic of Germany
E-mail: jap@informatik.uni-kiel.dbp.de

John Rushby
Computer Science Laboratory
SRI International
Menlo Park CA 94025
United States of America
E-mail: rushby@csl.sri.com

Henk Schepers
Department of Mathematics
and Computing Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven
The Netherlands
E-mail: schepers@win.tue.nl

Doug G. Weber
118 West Enfield Center Road
Ithaca, NY 14850
United States of America
E-mail: weber@keysoft.com

PREFACE

Practically every day, the media report that malfunctioning of a computer system resulted in incidents. This does not necessarily mean that the software and hardware making up such a system has not been designed with as much care as is commercially feasible. However, as the burden of controlling complicated systems is shifted onto computers, so does the complexity of computer software and hardware increase.

The sobering description of failures of some systems has led to the belief that there is a need for a distinct engineering discipline with its own theoretical foundations, objective design standards and supporting tools in order to develop reliable systems.

The term 'reliability (of a system or its components)' in computer science is often defined as the "probability that a certain system component functions correctly over a certain period of time". This requires that reliability is modelled in a time-dependant, quantitative probabilistic formal framework. However, reasoning about correctness of a system — i.e. an ability to deliver an a priori defined function, which is a qualitative issue — can be separated from quantitative probabilistic notions of reliability. A reliability of a system (or a subsystem) in qualitative sense can be expressed in terms of properties that qualitatively characterize the behaviour of a system that is error-prone.

The term 'fault-tolerance' describes that a system has properties which enable it to deliver its specified function despite of (certain) faults of its subsystem. Fault-tolerance is achieved by adding extra hardware and/or software which corrects the effects of faults. In this sense, a system can be called fault-tolerant if it can be proved the resulting (extended) system under some model of reliability meets the reliability requirements.

The chapters in this volume deal mostly with reliability from a qualitative point of view. It contains a selection of papers that focus on the state-of-the-art in formal specification, development and verification of fault-tolerant computing systems. Preliminary versions of some papers were presented at the School and

Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems held at University of Nijmegen in January 1992. Other chapters are written versions of lectures and tutorials presented at the same event.

The main theme of this volume can be formulated as follows: How does a specification, development and verification of conventional and fault- tolerant systems differ? How do the notations, methodology and tools used in design and development of fault-tolerant and conventional systems differ?

The purpose of this book is to explore these important issues, the definite answers, if they exist at all, are in my opinion still some years in the future.

The book is divided in two parts: **Concepts and Foundations** and **Applications**. Each part contains a number of contributions written by different researchers. Each chapter is self-contained and may be profitably studied without prior detailed familiarity with previous chapters. However, it is advisable to examine each chapter carefully because only then do many of the important and subtle differences in approach become evident.

The First Part: **Concepts and Foundations** sets the stage for what follows by defining the basic notions and practices of the field of design and specification of fault-tolerant systems. The chapter by Henk Schepers: "Terminology and Paradigms for Fault Tolerance" analyses the interaction between fault-hypothesis and design decisions.

A definition of the notion "fault-tolerance" that does not refer, as usually, to the functional correctness properties is given in chapter "Fault-Tolerance as Self-Similarity" by Doug Weber.

The chapter "Parameterized Semantics for Fault Tolerant Real-Time Systems" by Jos Coenen and Jozef Hooman presents a denotational semantics to describe real-time behaviour of distributed programs. In this semantics, the occurrences of hardware faults and their effects on real-time behaviour of programs can be modelled.

The effects of these faults upon the behaviour of the programs can be described as well. Hans A. Hansson in his chapter "Modeling Real-time and Reliability" provides a framework for specification and verification of distributed systems in which the reliability, timeliness and functionality can be modelled.

The Second Part: **Applications** is the "how-to" Part. It contains examples of the use of formal methods in specification and development of fault-tolerant systems. The chapter by John Rushby: "A Fault-Masking and Transient-Recovery Model for Digital Flight-Control Systems" presents a formal model and analysis for fault-masking and transient-recovery among replicated computers of digital flight-control system. This model has been specified in the language of EHDM and the crucial theorem and its corollary have been mechanically checked.

Zhiming Liu and Mathai Joseph in their chapter "Specification and Verification of Recovery in Asynchronous Communicating Systems" presents a method for specification and verification of general checkpointing programs. It combines the considerations of checkpointing, interference by physical faults and subsequent recovery so that the properties of fault-tolerant programs can be proved.

The chapter by Jan Peleska, "CSP, Formal Software Engineering and the Development of Fault-tolerant Systems", describes the use of formal techniques in development of flight control system in real industrial environment. In this article the Structured Method of Ward and Mellor is combined with formal specification language CSP of C.A.R. Hoare. The transformation schemata of Ward and Mellor are interpreted by means of translation rules so that a structured specification can be transformed into a CSP program. The use of the method is illustrated by showing that a dual computer system is tolerant to certain types of failures.

This book is suitable for graduate or advanced undergraduate course use when supplemented by additional readings that place the material contained herein in fuller context. Most of the techniques and notations described in this book are not yet ready for widespread use in commercial settings although some have been used in realistic setting.

FORMAL TECHNIQUES IN REAL-TIME AND FAULT-TOLERANT SYSTEMS

CONTENTS

Part I

Concepts and Foundations

TERMINOLOGY AND PARADIGMS FOR FAULT TOLERANCE

Henk Schepers[1]

ABSTRACT

To familiarize the reader with the field of fault tolerance, this report discusses the most important terms and paradigms used in that field. After establishing a basic terminology, the fundamental techniques to achieve fault tolerance, i.e. the basic ways to employ redundancy, are identified. In particular, the rôle that fault hypotheses play in the design of a fault tolerant system is illustrated.

To enable the development of formal methods for fault tolerance, the interaction between fault hypotheses and design decisions is analyzed in detail for two fault tolerant systems. The first is a stable virtual disk which is implemented using a number of unreliable physical disks. The second concerns a reliable broadcast protocol.

Keywords Fault tolerance, fault hypothesis, redundancy.

1.1 OF FAULTS AND FAILURES

According to Laprie (cf. [13]) fault tolerance is the property of a system "to

[1]Supported by the Dutch NWO under grant number NWI88.1517: 'Fault Tolerance: Paradigms, Models, Logics, Construction'.

provide, by redundancy, service complying with the specification in spite of faults having occurred or occurring". This report introduces the paradigms and associated terminology commonly used in the field of fault tolerance.

A system consists of components which interact as described by a design. There is no conceptual difference between the notions 'system' and 'component': the system is simply the component under discussion. The major difference between 'system' and 'environment' is that you can control the system but not the environment.

The service delivered by a component is the visible abstraction of the component's behaviour at the point of interaction — the interface. The behaviour of a system can easily be separated into two distinct categories: behaviour in accordance with the specification, and behaviour *not* in accordance with the specification. We refer to these kinds of behaviour as normal and abnormal, respectively. A *failure* occurs when the behaviour of a system deviates from that required by its specification [19]. The failure of a component appears to the system as a *fault*. Notice that there is no basic difference between 'fault' and 'failure': they are merely used to distinguish the cause from the consequence. In this report we do not consider system failures that are caused by design faults.

Faults are usually classified according to the specific aspects of the specification they violate. Timing faults, for instance, can be divided into late behaviour, which can lead to omission, and early behaviour, which can lead to overrun. Another example is the occurrence of a range fault the moment a value does not conform to the specified range.

If it is possible to deduce from assertions about a component's behaviour that some fault has occurred, we call that fault detectable. Different fault models arise from the assumptions about the correctness of the behaviour with respect to the various specification aspects, and, in case that behaviour is not assumed to be correct, the detectability of such faults.

When discussing hardware defects, the notions 'transient' and 'permanent' are well established [1]. A *transient* defect is present for only a limited period of time (no longer than some threshold) after which it spontaneously disappears. Any defect which is present for longer than that threshold period is said to be *permanent*. Analogous to this, a system failure transient or permanent.

The remainder of this report is organized as follows: in Section 2 the various stages of the procedure of tolerating faults are mentioned and it is discussed what designing for fault tolerance is about. In particular, the new element introduced by the specification of fault tolerant systems, i.e. specification relative to the system's fault hypotheses, is illustrated. In Section 3 a number of typical paradigms for fault tolerance are discussed. Section 4 presents, as a case study, the design of a stable storage. Finally, Section 5 discusses, also as a case study, the design of a reliable broadcast protocol.

1.2 TOLERATING FAULTS

As mentioned before, fault tolerance is concerned with providing the specified service in the presence of faults. To do so, fault tolerance depends upon the effective deployment and utilization of redundancy[2].

Of course, a fault tolerant system can tolerate only a limited number of certain types of faults. In fault tolerant systems, three domains of component behaviour are usually distinguished: normal, exceptional and catastrophic (see for instance [14]). Normal behaviour is the behaviour when no faults occur. The discriminating factor between exceptional and catastrophic behaviour is the component's *fault hypothesis* which stipulates how faults affect the component's normal behaviour. An example is the hypothesis that a communication medium may lose but not corrupt messages. Relative to the fault hypothesis an exceptional behaviour exhibits an abnormality which should be tolerated (to an extent that remains to be specified), and a catastrophic behaviour has an abnormality that was not anticipated. Thus, for this communication medium the corruption of messages is catastrophic. In general, the catastrophic behaviour of a component cannot be tolerated by a system. Under certain fault hypotheses, the system is designed as if the hypothetical faults are the only faults it can experience and measures are taken to tolerate (only) those *anticipated* faults.

The most rigorous way to tolerate a fault is to use so much redundancy that it can be masked: for example the triple modular redundancy paradigm as presented in Section 3.3. But this kind of redundancy is generally too expensive.

If faults cannot be masked, then our first concern is how to identify an anticipated fault (*fault detection*). Before the system can be allowed to continue to provide its service, *fault diagnosis* must be applied and the fault's — unwanted — consequences must be undone. The fault diagnosis must identify the components that are responsible for the fault and also whether that fault is transient

[2]Sometimes redundancy is classified by the kind of element that is redundant (e.g. component redundancy and information redundancy). Such a classification, however, is not orthogonal (for instance component redundancy implies information redundancy).

or permanent.

If the fault is only transient, its consequences can be undone by simply restarting the system[3], i.e. by putting it in some initial state, or, in case a valid system state is regularly recorded as a checkpoint, by bringing the system back to its last checkpoint and then continuing operation from that state. This technique is called backward error recovery, and it allows actions to be atomic [15]: they are either executed completely or not at all. Manipulating the current erroneous state to produce an error free new state is called forward error recovery. Once taken to a consistent state the system can continue to provide its service.

If the fault is not transient but permanent the system needs repair first. If the faulty component can be replaced, the system can deliver its service without modification; otherwise, other components must take over the faulty component's tasks in addition to their own, and this may lead to a degradation of the service in case not all the tasks can be fulfilled. *Graceful* degradation allows as many tasks as possible to be still accomplished. Replacing a faulty component can be done either physically or logically by means of *reconfiguration*, where a faulty component is taken out of action and a spare, already present in the system, is put into service.

1.3 PARADIGMS FOR FAULT TOLERANCE

To familiarize the reader with the fault tolerance field a few typical paradigms are presented, and analyzed.

1.3.1 Consistency check

Consistency check paradigms apply to those cases where the output of a component is checked with respect to its specified functionality. Such paradigms are used especially when a component performs a mathematical function, for instance by verifying whether the result conforms to the specified format (*syntax checking*), by verifying whether the result lies in the specified range (*range checking*) or by verifying whether the application of the reverse function to the result yields the input again (*reversal checking*).

[3]This only helps, of course, if the application allows the involved delay; for time-critical applications this usually is not the case.