Lecture Notes in Computer Science 4119

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

Christophe Dony Jørgen Lindskov Knudsen
Alexander Romanovsky Anand Tripathi (Eds.)

Advanced Topics in Exception Handling Techniques

Springer

Volume Editors

Christophe Dony
Montpellier-II University and CNRS
LIRMM Laboratory
161 rue Ada, 34392 Montpellier Cedex 05, France
E-mail: dony@lirmm.fr

Jørgen Lindskov Knudsen
University of Aarhus
Department of Computing Science
Aabogade 34, 8200 Aarhus N, Denmark
E-mail: jlk@daimi.au.dk

Alexander Romanovsky
University of Newcastle upon Tyne
Department of Computing Science
Newcastle upon Tyne, NE1 7RU, UK
E-mail: alexander.romanovsky@ncl.ac.uk

Anand Tripathi
University of Minnesota
Department of Computer Science and Engineering
Minneapolis, MN 55455, USA
E-mail: tripathi@cs.umn.edu

Library of Congress Control Number:2006930921

CR Subject Classification (1998): C.2.4, D.1.3, D.2, D.4.5, F.2.1-2, D.3, F.3

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN 0302-9743
ISBN-10 3-540-37443-4 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-37443-5 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11818502 06/3142 5 4 3 2 1 0

Preface

Exception handling is an essential part of software and system architectures and a crucial element in the tool-set that enables the building of resilient, robust and safe software systems. The term "exception" has been variously defined but most commonly is used to refer to, as explained in the foreword to this book, "predictable but uncommon situations" encountered during the execution of a program. Since the mid-1970s, when John Goodenough, David Parnas and Brian Randell published the first papers to define the research areas of fault-tolerant computing and programmed exception handling, a range of design principles and programming techniques have been developed to cope with such uncommon situations. A 'when others' clause has also been introduced, as it is sometimes clearly impossible to predict all of them.

Having held a fruitful ECOOP workshop in 2000, in 2001 we published the first collection of papers on this topic (*Advances in Exception Handling Techniques*, LNCS 2022), which brought together a range of studies and solutions available at the time in all areas of software engineering where exceptions have to be dealt with: programming languages, software design and modelling, as well as concurrent, distributed, information and workflow systems.

Discussions held and findings presented at the 2000 workshop also clearly showed that exception handling was – and still remains – a vital research area for the following two complementary reasons.

Firstly, while various exception handling mechanisms are supported in languages and design methodologies, there are still very serious problems when applying them in practice. This may be due to a number of factors: the complexity of exception code design and analysis, failing to employ exception handling at appropriate development phases, subtle semantic differences between proposed constructs, lack of methodologies and lack of experimentation. Let us consider such examples as hierarchies of exception classes, exception objects passed to handlers, and context-dependent dynamic scope handlers defined by dedicated "try-catch"-like primitives or via built-in higher-order functions and lexical closures. It is generally agreed that these are state-of-the-art solutions to some of the challenges of exception handling in sequential programs. On a large scale, however, these solutions (whatever their concrete instantiation in a given language) have not yet been used extensively enough, and recent case studies have revealed many unanticipated drawbacks. This indicates that, even in the restricted context of sequential programming, no solution or system for exception handling can yet be regarded as definitively accepted. The situation is clearly exacerbated with regard to concurrent programming, where, despite many sound proposals – largely presented in our 2001 collection – it is difficult to see any standard solution coming; we are obviously still in the exploratory research phase.

Secondly, many new research challenges stem from the unique nature of every emerging application environment or paradigm, such as pervasive computing, ambient computing, grid computing over the Internet, component-based and service-based applications, mobile applications, sensor networks and network transparent

distributed systems. These modern systems are growing increasingly more complex, making it imperative for application software to handle errors and other potentially damaging events. It is also necessary to take into account such orthogonal concerns as self-repair, adaptation, openness and reflectivity, which bring complex crosscutting constraints into the equation.

It was in order to address these issues that we organized two ECOOP workshops on exception handling in 2003 and 2005. This book is primarily an outcome of these two events, where a number of original proposals were presented. In a manner similar to the preparation of our first book, several workshop participants as well as a number of other leading researchers in the field were invited to contribute a chapter each. It is only natural that the choice of contributors to this book reflects our personal views on this research area. We hope, however, that reading this volume will prove rewarding to any computer scientist or practitioner who is striving to build a resilient, robust and safe system.

The book is composed of five parts; the first four deal with topics relevant to exception handling in the areas of programming languages, concurrency and operating systems, pervasive computing, and requirements and specifications. The last one focuses on case studies, experimentation and qualitative comparisons. As any other attempt to carve up a complex field into a few neat domains, this one will suffer from numerous exceptions. For example, a number of papers included elsewhere are also directly related to the programming language part. We regard it as more important to foreground concurrency and pervasive computing, which we believe are major research trends in exception handling. It is also worth noting that almost half of the papers deal with asynchronous and concurrent systems, which have become hugely significant in modern computing environments and pose one of today's most important challenges for the exception handling research community.

Our thanks go to all our authors, whose work made this book possible. We would also like to thank Emeritus Prof. Brian Randell for contributing the foreword. We are grateful to Alfred Hofmann, of Springer, for recognizing the importance of the project. Finally, we appreciate the help of William Bail, Michael E. Caspersen, Dickson Chiu, Erik Ernst, Alessandro Garcia, Jörg Kienzle, Devdatta Kulkarni, Ole Lehrmann Madsen, Robert Miller, Christelle Urtado, Sylvain Vauttier and Ian Welch in the reviewing process.

June 2006

Christophe Dony
Jørgen Lindskov Knudsen
Alexander Romanovsky
Anand Tripathi

Foreword

The subject of exception handling, though it has its roots in programming language design, can and I suggest should be viewed in more general terms. It is of course at base just another "divide and conquer" approach to coping with complexity – originally just the complexity of conventional (sequential) programs. Well-designed language constructs allied to sanitary programming languages enable programmers to simplify their task by identifying and dealing separately with various predictable but uncommon situations – typically error situations. Each such situation, described perhaps by a set of logical pre-conditions, can have its own separately coded exception handler associated with it, designed to deal just with these particular pre-conditions, and if possible to achieve the desired post-conditions. The fact that this might not always be possible leads naturally to the idea of having various different sets of post-conditions, one for the normal (assumed to be error-free) situation, the others for various separate pre-defined error states. These other post-conditions can then, if appropriate, lead to exception handling at a higher level.

In a more interesting and more general case one is concerned not with isolated sequential programs, but rather with programs that define interacting asynchronous processes (often running on separate computers). Such interaction can be in order to cooperate in pursuit of some common goal, for example, to search a huge index of web pages efficiently, and/or in order to compete, for example, in making use of some common resource, such as a shared database. In such circumstances exception handling facilities need to concern themselves not just with the structuring of the text of complex programs, but also with the structuring, in time and space, of the complex asynchronous activities that these programs give rise to. Thus the exception handling facilities have to include means of isolating the various error handling processes from each other (that is, means of error confinement) so that the various error handlers can as far possible be designed (and validated) independently of, and operate separately from, each other. Transactions, conversations and coordinated atomic actions are all examples of means for such error confinement.

But when one takes this view of exception (or error) handling it is but a short, and very useful though not always taken, step to regarding exception handling as a means of *system*, not just software, structuring. One consequence, of course, is the idea of employing exception handling as an architectural structuring principle, used at each stage of system design, including before any actual program code is written – ideally a structuring that is retained very explicitly in the final completed system, not just in the original design documentation. But what is to my mind the more fundamental consequence is that an adequately general method of exception handling in asynchronous systems can be used in the design not only of computer systems and their software, but of large systems made up of computers *and* the devices that they monitor and control. (For example, my colleagues and I have investigated the utility of such techniques in several safety-critical factory automation scenarios, in which various machines needed to be operated in a coordinated fashion so as to avoid

physical clashes – the method of structuring used greatly aided both the design of the overall control system and the formal proof of its safety.)

Indeed such a general method of exception handling can, I believe, be of great use in what can be termed computer-based systems, i.e., systems made up of computers *and* people. For example, an adequately general exception handing approach can be an aid to deciding which tasks are best allocated to computers, for example, because of their predictability and frequency, and which are best left to human beings who can, one hopes, recognize and act sensibly in awkward, unusual or even unpredicted situations – and then of organizing the interactions that have to occur between the computers and the humans. Achieving an effective such separation of logical concerns is of course not easy, and requires very careful consideration of the placement and design of interfaces, and hence the establishment of protocols for communication between the computers and the humans, both during normal operation and when things start going wrong. Nevertheless, such design if done well can lead to an overall computer-based system that makes effective use of the complementary abilities of computers and humans. But done badly, one can end up with a system that is extremely frustrating to the humans involved, and which is in all probability very unsatisfactory, with respect to overall security and dependability as well as usability.

Incidentally, this more general (in fact recursive) approach to exception handling involves abandoning such simplistic notions as "outermost" transactions that assume that the overall information that has been transmitted to a computer system by a user (or vice versa) has been completely and irrevocably validated. And it involves careful consideration of the likely effectiveness of any planned error confinement strategies, and of what to do when such strategies break down, since maintaining error confinement in the world outside the computer can be problematic.

In summary, exception handling techniques, though by no means a panacea, can be a powerful aid to structuring and hence simplifying very complex situations and the design of systems that have to cope with these situations. Moreover, such simplification is not bought at the cost of ignoring complex realities – the pursuit of simplicity in system design is always commendable, but, to quote Einstein: "Everything should be made as simple as possible, but not simpler." The availability of good exception handling facilities encourages system designers to provide for the possibility of various obscure types of faults occurring, and of multiple coincident faults, in software or hardware or among external devices and humans, rather than risk relying on being able to muddle through somehow and then patch their design when the need arises.

June 2006

Brian Randell
University of Newcastle upon Tyne

Table of Contents

Programming Languages

Concurrency and Operating Systems

Pervasive Computing Systems

Requirements and Specification

Engineering and Experience