

Advanced Topics in Exception Handling Techniques

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 handling 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