The Journal of Systems and Software 82 (2009) 874–890

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Improving reliability of cooperative concurrent systems with exception flow analysis

Fernando Castor Filho [a,*], Alexander Romanovsky [b], Cecília Mary F. Rubira [c]

[a] Informatics Center, Federal University of Pernambuco, Av. Prof. Lus Freire s/n, 50740-540 Recife, PE, Brazil
[b] School of Computing Science, Newcastle University, Newcastle NE1 7RU, UK
[c] Institute of Computing, State University of Campinas, P.O. Box 6176, 13084-971 Campinas, SP, Brazil

ARTICLE INFO

Article history:
Received 2 January 2008
Received in revised form 3 December 2008
Accepted 8 December 2008
Available online 24 December 2008

Keywords:
Exception handling
Coordinated error recovery
Verification
B method
Alloy

ABSTRACT

Developers of fault-tolerant distributed systems need to guarantee that fault tolerance mechanisms they build are in themselves reliable. Otherwise, these mechanisms might in the end negatively affect overall system dependability, thus defeating the purpose of introducing fault tolerance into the system. To achieve the desired levels of reliability, mechanisms for detecting and handling errors should be developed rigorously or formally. We present an approach to modeling and verifying fault-tolerant distributed systems that use exception handling as the main fault tolerance mechanism. In the proposed approach, a formal model is employed to specify the structure of a system in terms of cooperating participants that handle exceptions in a coordinated manner, and coordinated atomic actions serve as representatives of mechanisms for exception handling in concurrent systems. We validate the approach through two case studies: (i) a system responsible for managing a production cell, and (ii) a medical control system. In both systems, the proposed approach has helped us to uncover design faults in the form of implicit assumptions and omissions in the original specifications.

1. Introduction

Applications that could potentially endanger human lives or lead to great financial losses are usually made fault-tolerant (Anderson and Lee, 1990) so that they are capable of providing their intended service, even if only partially, when errors occur. Fault-tolerant systems include mechanisms for detecting errors in their states and recovering from them. There are two main types of error recovery (Anderson and Lee, 1990): backward and forward. The former is based on rolling a system back to its previous correct state and generally uses either diversely implemented software or simple retry; the latter involves transforming the system into any correct state, is typically application-specific and relies on an exception-handling mechanism (Cristian, 1989; Goodenough, 1975).

Usually, a significant part of the system code is devoted to error detection and handling (Cristian, 1989; Weimer and Necula, 2004). Cristian (1989) claimed that, for telephone switching applications, this often amounted to more than two thirds of the overall system code. A more recent study (Weimer and Necula, 2004) of a set of open-source applications written in Java discovered that between 1% and 5% of the program text consisted of exception handlers (`catch` blocks) and clean-up actions (`finally` blocks). In another study (Reimer and Srinivasan, 2003), focusing on five large-scale applications based on the Java Enterprise Edition (Bodoff, 2004) platform, the ratio of the number of exception handlers to that of operations in each application varied between 0.058 and 1.79. Finally, some of us have conducted yet another study (Castor Filho et al., 2006), involving four applications. Two of them were produced in industry and two in academia. In this case, the ratio of the number of handlers to that of operations ranged from 0.099 to 0.208.

In spite of the pervasiveness of error detection and handling code, it is usually the least understood, tested or documented (Cristian, 1989) in a system. This is mainly due to the tendency among developers to focus on the normal activity of applications and only deal with the code responsible for error detection and handling at the implementation phase. What is more, there are other issues that aggravate this situation in distributed systems, such as the high cost of reaching an agreement, the lack of a global view on the system state, multiple concurrent errors, difficulties in ensuring error isolation, etc. All of these factors complicate the development of reliable systems in general and of mechanisms that make them reliable in particular. The overall result is that the parts of a system responsible for making it reliable are usually the source of design faults (Cristian, 1989; Reimer and Srinivasan, 2003; Weimer and Necula, 2004).

* Corresponding author. Tel.: +55 81 92251458.
E-mail addresses: fcastor@acm.org (F. Castor Filho), alexander.romanovsky@newcastle.ac.uk (A. Romanovsky), cmrubira@ic.unicamp.br (C.M.F. Rubira).

0164-1212/$ - see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2008.12.015

For the desired levels of reliability to be achieved in a system, error detecting and handling mechanisms should be systematically applied from the early phases of development (Rubira et al., 2005). Moreover, the construction of these fault tolerance mechanisms should follow a rigorous or formal development methodology (Bernardeschi et al., 2002). In this manner, these mechanisms are made more reliable and do not introduce new faults into the system.

1.1. Problem

The concept of coordinated atomic (CA) actions (Xu et al., 1995) was developed by combining distributed transactions and atomic actions. The latter are used to control cooperative concurrency and to implement exception handling (Campbell and Randell, 1986), whereas the former (Gray and Reuter, 1993) are used to maintain the consistency of resources shared by competing actions. CA actions function as exception-handling contexts for cooperative systems, and exceptions raised in an action are handled cooperatively by all of its participants. If two or more exceptions are concurrently raised, an *exception resolution mechanism* (Campbell and Randell, 1986) is employed to identify an exception that represents all the exceptions raised concurrently (a *resolved exception*) in order to handle it. Many case studies (Beder et al., 2000; Romanovsky et al., 2003; Xu et al., 2002; Zorzo et al., 1999) have shown that CA actions are a powerful and useful tool for structuring large distributed fault-tolerant systems. In this paper, we view CA actions as representative of mechanisms for exception handling in distributed systems.

In order for CA actions to be applicable in constructing complex real-world systems with strict dependability requirements, software development based on CA actions needs to be supported with rigorous models, techniques, and tools. Several approaches have been proposed to formalize the CA action concept aiming to either offer a more complete and rigorous description of the concept (Vachon and Guelfi, 2000) or to verify CA action-based designs (Xu et al., 2002). However, there is an important aspect of CA actions that has not been properly addressed by existing work, and that is coordinated exception handling. This is surprising, since exception-handling complements other techniques in improving reliability, such as atomic transactions, and promotes the implementation of specialized and sophisticated error recovery measures. Moreover, in some distributed applications, a roll back is not possible or is prohibitively expensive. In this scenario, exception handling may be the only sensible choice available.

Some authors (Buhr and Mok, 2000) claim that mechanisms for involving multiple participants in order to cooperatively handle exceptions are difficult for both implementation and use. We believe, however, that programmers will make more mistakes in an ad hoc implementation of cooperative exception handling than in applying well-defined mechanisms provided by such general frameworks as CA actions. There is thus a need for techniques and tools that would mitigate the inherent complexity of exception handling in a concurrent setting and help developers in specifying and designing systems that make use of this feature.

In this paper, we examine the problem of specifying a CA action-based design in a way that would allow automatic verification of whether it exhibits certain properties that are relevant to coordinated exception handling. Our aim is to understand what would be required of modeling exception propagation and handling in this1 design. Comprehension and documentation of exception propagation in non-concurrent software systems is by itself a complex issue and an active research area (Cacho et al., 2008,; Castor Filho et al., 2006; Fu and Ryder, 2007; Jiang et al., 2004; Robillard and Murphy, 2003). Concurrency is a serious complicating factor for exception propagation. In CA action-based design, a participant can not only raise and handle exceptions, but also spawn new actions that are, themselves, exception-handling contexts involving multiple participants. What further aggravates matters is that it is possible for two or more exceptions to be concurrently raised inside an action. A model of actions and their participants must contemplate every possible combination of exceptions or, at least, explicitly point out combinations that cannot happen in practice. Moreover, it should make it possible to specify how participants react when faced with different sets of concurrently raised exceptions. Finally, since exception handling is closely related to action structuring, it should also model the nesting and composition (Romanovsky et al., 2003) of CA actions and how these affect exception propagation and handling.

1.2. Proposed approach

In this paper, we present an approach to modeling CA action-based design that makes it possible to automatically verify these models using a constraint solver. The main component of the proposed approach is a formal model of CA actions that specifies the structuring of a system in terms of actions, as well as information relevant to exception flow amongst these actions. This model can be directly specified using well-known specification languages, such as Alloy (Jackson, 2002) or B (Abrial, 1996), and automatically verified using tool sets associated with them. The proposed approach makes it possible to check whether a CA action-based software system satisfies several key properties.

This paper is organized as follows. The next section provides some background on CA actions, the B method and notation, and the Alloy specification language. Section 3 presents the proposed approach, including a description of the generic CA action model and some of the properties that it helps to verify. Section 4 formalizes the basic properties of the generic CA action model. We then illustrate the feasibility and usefulness of the proposed approach in two case studies. Section 6 reviews related work, and the last one sums up the paper and outlines directions for future work.

2. Background

In order to present our approach, we need to introduce several topics first. We begin with CA actions, a scheme for building fault-tolerant concurrent systems that employ exception handling. We then proceed to describe two formal specification languages, Alloy (Jackson, 2002) and B (Abrial, 1996). These languages are examples of formal notations that can be used in combination with the approach proposed here in order to specify and verify some properties of fault-tolerant distributed systems based on CA actions. Both are similar to Z (Woodcock and Davies, 1996), declarative in nature, and supported by automated verification tools. It is important to stress, however, that they were designed with very different goals in mind.

2.1. Coordinated atomic actions

CA actions are a unified scheme for coordinating complex concurrent activities and supporting error recovery among multiple interacting components. It helps to decrease the overall system complexity and simplify development by structuring the system in terms of nested recovery units. A CA action is designed as a set of roles cooperating inside it and a set of resources accessed by these roles. An action starts when its roles are taken by participants. A participant abstracts away the underlying unit of concurrency, i.e., it can be a process, a thread, an active object, or any similar mechanism. In the course of the action, participants can access external resources. The latter must be accessed according to the ACID (atomicity, consistency, isolation, durability) properties