Representing Exceptional Behaviour at the earlier Phases of Software Development

Rogério de Lemos

Computing Laboratory
University of Kent at Canterbury, CT2 7NF, UK
r.delemos@ukc.ac.uk

Exception handling is a structuring technique that facilitates the design of systems by encapsulating the process of error recovery. Exception handling has been traditionally associated with the design phase of the software lifecycle, during which all the effort is made to protect the application software from faults that may be introduced during requirements, design, and implementation, or can occur at the support level. The consequence of such approach is that the appropriate context in which errors should be detected and recovered is lost also it is lost the potential correlation that might exist between the error states of the different contexts and how these should be recovered in an optimised way.

Dealing with concurrent manifestations of several faults at different phases of system development has been recognised as a serious problem that has not received enough attention /Avizienes 97/. Ideally, for each identified phase of the software lifecycle, a class of exceptions should be defined depending on the abstraction level (or context) of the software system being modelled and analysed, as represented in figure 1. As the software development progresses, new exceptions are identified and their respective handlers specified. However, the exceptions identified at the different phases can be causally and timely related, which might constraint the specification of their respective handlers. Moreover, it might be the case that the rationalisation of exceptions might enable the usage of a single handler for different classes of exceptions. At every phase of the software development failure assumptions have to be revised once the system structure is decomposed and behaviour refined. This process of revising failure assumptions, as we progress through the software lifecycle, might also lead to the refinement of the exceptions and handlers previously specified.

Instead of assuming that exception handling should be restricted to the later phases of software development, recent work has attempted to provide systematic and effective approaches on how to deal with exception handling at all phases of the software lifecycle. These approaches provide a stepwise method for defining exceptions and their respective handlers, thus eliminating the *ad hoc* way in which exception handling is sometimes considered during the later phases of the software lifecycle.

In one of these works, the description of exceptional behaviour within the software lifecycle is supported by a co-operative object-oriented approach that allows the representation of collaborative behaviour between objects at different phases of the software development /de Lemos 99/. In a co-operative object-oriented description of a system, the role of a co-operative action (CO action) is to co-ordinate the collaboration between classes, which also involves the description of exception

handling in both objects and co-operations. In terms of objects, when abnormal behaviour cannot be handle locally by an object exceptions have to be propagated to co-operating objects; co-operations provide the support for co-ordinating the propagation and the handling of exceptions between co-operating objects. When exceptions are raised inside co-operations that cannot be related to any specific object involved in the co-operation, all the co-operating objects should handle the exception in a co-ordinated manner, to guarantee that once the co-operation is finished all the co-operating objects are in a known consistent state.

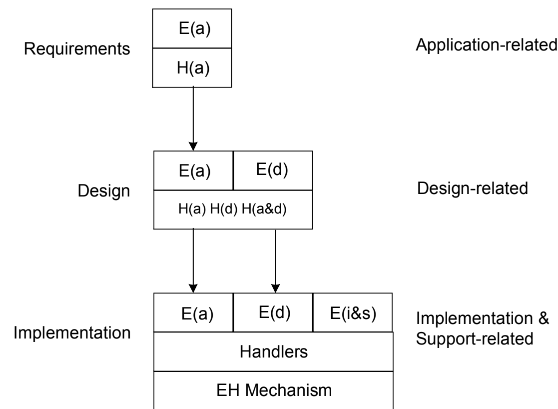


Figure 1. Exception handling in the software lifecycle.

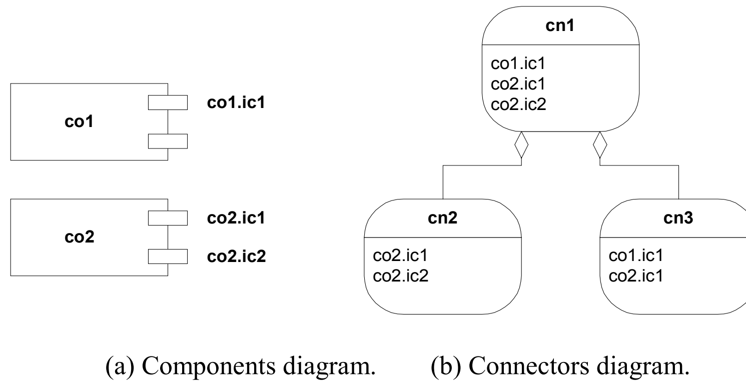
In a more recent work, a systematic approach for incorporating the exception behaviour in the software lifecycle was considered in the context of Catalysis and the idealised fault-tolerant component architecture /Ferreira 01/. Catalysis is a well established technique for the systematic development of component-based systems that has three primary modelling constructs: collaborations, types, and refinement /D'Souza 98/. What differs Catalysis from other traditional approaches are collaborations: these are considered first class entities that incorporate how a group of objects jointly behaves when configured together. Collaborations are defined in terms of actions with their pre- and post-conditions that define collaboration. For representing the propagation and handling of exceptions in a collaboration, the description of actions is extended by including the definition of *signals* and *handlers*, and both the normal and exceptional behaviours of the collaboration are represented in terms of collaboration diagrams. While Catalysis provides the process for developing software, the idealised fault-tolerant component provides the elementary architectural representation for describing the fault tolerant activities of a system /Lee 90/.

While developing the above work, we have come across several deficiencies regarding the modelling and analysis of exceptional behaviour in the software lifecycle, particularly, at the higher levels of abstraction. One of the problems that we have faced was the explicit representation of exception handling in the use cases. These were initially conceived for succinctly describing the problem at hand, but if exceptional behaviour is to be considered at the requirements level, then exceptions have to be represented in the use cases, thus increasing the complexity of their

description. A possible approach is to describe use cases with different levels of detail, and to make sure that these descriptions are consistent and accurate. The three basic stages for this description are the following. In the first stage, the use case would be described in its usual way /Jacobson 92/. In the second stage, we would refine the use case by identifying system variables, providing a table of exceptions representing the causal relation between the abnormal behaviours of actors and use cases, and providing collaboration diagrams for the exception behaviour. The objective of this stage is to model and analyse the interdependencies between actors and use cases taking into consideration their exceptional and failure behaviours. In the third stage, we would formalise the use cases in terms of a first order logic that could be used as a basis to obtain a state model, which can then be model checked. This last stage would be very similar to the process of conducting safety analysis of a critical system /de Lemos 01/. In this proposed approach of three stages, the balance between being rigorous and formal depends on the criticality of the system being analysed. If the second stage is well documented, and consistent with the underlying formal model, then the client would be able to understand and discuss the requirements of the system without the need of understanding a formal language.

Another problem that we have faced when dealing with exceptional behaviour at the early phases of the software lifecycle was the representation of exception handling at the architectural level. If the current trends of software engineering are to be followed, i.e. the development of new systems out of existing components, then the architectural representation of systems will have a key role in the process of adapting either the components or the interactions between them. For example, if mechanisms for enforcing dependability of services have to be added to an untrustworthy component, then it is necessary to provide structural means for incorporating the required changes. The co-operative architectural style being proposed offers the appropriate means for structuring complex applications that are intrinsically collaborative in their nature, and aims to provide fault tolerance in the context of how exception handling mechanisms can be added to untrustworthy components. The key architectural element of this style is a co-operative connector that encapsulates collaborative behaviour between several components, which also involves co-ordinating the handling of exceptions between components. An example of a co-operative architecture is shown figure 2. The depicted system is composed by two components (co1 and co2) that are interconnected by three connectors (cn1, cn2, and cn3). The connector cn1 captures the collaborative activity associated with the three roles of the components co1 and co2. The collaborative activities associated with connectors cn2 and cn3 are nested to connector cn1.

A co-operative connector is described in terms of the roles involved in the collaborative activity, and the specification of collaborative behaviour in terms of normal, exceptional and failures behaviours. The normal behaviour in defined in terms of the pre- and post-conditions, the invariant that should hold, and the collaborative operation to be performed under the control of the connector. For the specification of exceptional behaviour, a handler replaces the operation, and is defined in terms of its start and finish events. Although the pre-conditions for normal and exceptional behaviours are the same, the post-conditions for the exceptional behaviour might be different, depending on the degraded outcomes of a collaborative operation, once an exception has occurred.



(a) Components diagram. (b) Connectors diagram.

Figure 2. Co-operative software architecture.

Figure 3 shows the representation of exceptional behaviour of a co-operative connector in terms of timed automata extended with data variables /Larsen 1997/. The automaton on the left starts a co-operation when the pre-condition is true (pre==true), it executes the collaborative operation (operation:=true) while the invariant holds (invariant==true), and exists the co-operation when the post-condition of normal behaviour is true (post_normal==true). The representation of exception behaviour is related to the raising of an exception (signal!) that starts a handler on the automaton on the right. Once the handler is executed (handler:=true), the co-operation is finished assuming that the post-condition for exception behaviour is true (pos_exceptional==true).

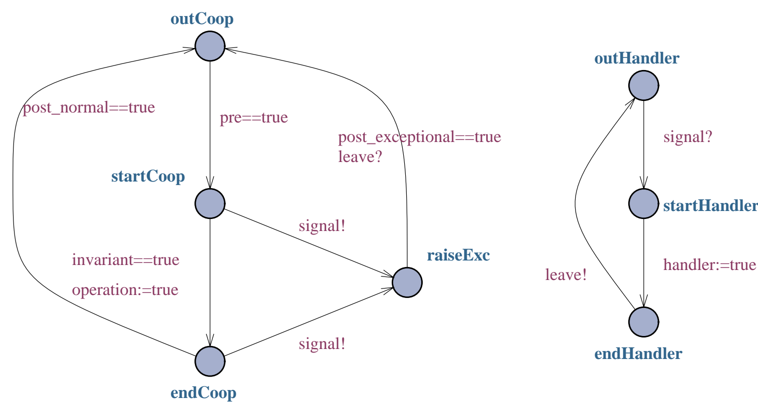


Figure 3. Automata representation of exceptional behaviour.

Most of this work is still on going, and currently we are investigating architectural description languages that would incorporate the adequate capabilities for representing and analysing exceptional behaviour.

Acknowledgements. This work was performed in collaboration with Gisele R. M. Ferreira, Alexander Romanovsky, and Cecília M. F. Rubira.

References

/Avizienis 97/ A. Avizienis. "Toward Systematic Design of Fault-Tolerant Systems". *Computer 30 (4)*. April 1997. pp. 51-58.

/de Lemos 99/ R. de Lemos, A. Romanovsky. "Exception Handling in a Cooperative Object-Oriented Approach". *Proc. of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'99)*. Saint Malo, France. May, 1999. pp. 3-13.

/de Lemos 01/ R. de Lemos. *Analysing Failure Behaviours in Component Interaction*. UKC Computing Laboratory Technical Report. 2001.

/D'Souza 98/ D. D'Souza, and A. C. Wills. *Objects, Components and Frameworks with UML: The Catalysis Approach.* Addison-Wesley, Reading, MA 1998.

/Ferreira 01/ G. Ferreira, C. Rubira, and R. de Lemos. "Explicit Representation of Exception Handling in the Development of Dependable Component-based Software".

/Jacobson 92/ I. Jacobson. *Object-Oriented Software Engineering - A Use Case Driven Approach.* Addison-Wesley. 1992.

/Lee 90/ P. Lee, and T. Anderson. *Fault-Tolerance: Principles and Practice*. Springer-Verlag 2nd Edition. 1990.