Exception Handling in Object Oriented Systems

Alexander Romanovsky[1], Christophe Dony[2], Jørgen Lindskov Knudsen[3], and Anand Tripathi[4]

[1]Department of Computing Science, University of Newcastle upon Tyne
Newcastle upon Tyne, NE1 7RU, UK
alexander.romanovsky@ncl.ac.uk

[2]Université Montpellier-II and LIRMM Laboratory,
161 rue Ada, 34392 Montpellier Cedex 5, France
dony@lirmm.fr

[3]Department of Computer Science, University of Aarhus,
Åbogade 34, DK-8200 Århus N, Denmark
jlk@daimi.au.dk

[4]Department of Computer Science, University of Minnesota
Minneapolis, MN 55455, USA
tripathi@cs.umn.edu

Abstract. Exception handling continues to be a challenging problem in object oriented system design. One reason for this is that today's software systems are getting increasingly more complex. Moreover, exception handling is needed in a wide range of application systems, sometimes requiring domain-specific models for handling exceptions. Also, concurrency, distribution, and code mobility add new dimensions to the existing challenges in this area. The integration of exception handling mechanisms in a design needs to be based on well-founded principles and formal models to deal with the complexities of such systems and to ensure robust and reliable operation. It needs to be pursued at the very start of a design with a clear understanding of the ensuing implications at all stages, ranging from design specification, implementation, operation, maintenance, and evolution. This workshop was structured around the presentation and discussion of the various research issues in this regard to develop a common understanding of the current and future directions of research in this area.

1 Summary of Objectives and Results

Modern object oriented systems are getting more complex and they have to cope with an increasing number of exception conditions representing abnormal situations or failures. The most general way of dealing with these problems is by incorporating exception handling techniques in the design. Over the past decade, several models and mechanisms for handling exceptions have been proposed and developed for object oriented (OO) languages and systems, but there are still many open problems in applying them in practice due to the complexity of exception code design and analysis, due to lack of methodologies supporting proper use of exception handling, and by not addressing exception handling at proper phases of OO system development.

Traditionally, over the past 30 years, the research in exception handling has centered around the design and implementation of programming languages. More recently, there has been an increase in research activities dealing with exception handling models and mechanisms in a wide range of application systems such as databases and transaction processing systems, workflow systems, distributed object management

systems, and concurrent systems. Besides the development of application-driven and domain-specific models for exception handling, the research activities in this area have been driven by the need for formal models of exception handling mechanisms to facilitate design and analysis of large-scale OO systems. The aim of this workshop was to bring together the active and leading researchers in this area to develop a common understanding of the important research directions that are central to the various strands of research in this area. The last workshop on this topic at an ECOOP conference was about 10 years ago, therefore, we felt a clear need to provide a forum for the exchange of ideas to assess the important problems and future directions for research in this area.

We aimed at discussing important topics of developing and using exception handling techniques in OO systems and languages. We invited participants interested in discussing their research on exception handling in areas related to OO systems, in particular: formal models, distributed and concurrent systems, practical experience, new application areas, mobile object systems, new OO paradigms (e.g., OO workflows, transactions), design patterns and frameworks, practical languages (Java, Ada 95, Smalltalk, BETA), open architectures, fault tolerance.

The workshop was attended by 17 researchers who participated in the presentation and discussion of 14 position papers. These presentations and discussions were grouped into four thematic sessions. The first session (five presentations) addressed linguistic issues in the context of contemporary models and exploration of new ideas, the topics included reviews of various models for exception handling in programming languages, fault tolerant computing, aspect-oriented programming and distributed object management systems. The focus of the second session (three presentations) was on semantics and implementation issues. Presented works were related to type theory based formal specification - the semantics of the Java exception handling model, to the impact of exceptions on code optimization in compiling Java programs, and to the relationships between triggers in database systems with the conventional notions of exceptions. The focus of the third session (four presentations) was on issues related to concurrency, distribution, and mobility. These presentations addressed exception handling in mobile agent systems and concurrent transaction-oriented systems. The global topic of the fourth session (two presentations) was the design, achievement and evolution of OO systems in the presence of exception handling mechanisms.

2 Participants

Yolande Ahronovitz	Université Montpellier-II, France yolande@lirmm.fr
Christophe Dony	Université Montpellier-II, France dony@lirmm.fr
Sophia Drossopoulou	Imperial College, UK scd@doc.ic.ac.uk
Bjørn Egil Hansen	DNV, UK Bjorn.Egil.Hansen@dnv.com
Ricardo Jimenez-Peris	Universidad Politécnica de Madrid, Spain rjimenez@fi.upm.es
Mik Kersten	Xerox PARC, USA mkersten@parc.xerox.com
Jörg Kienzle	EPFL, Switzerland jkienzle@di.epfl.ch
Jørgen Lindskov Knudsen	University of Aarhus, Denmark jlk@daimi.au.dk

Martin Lippert	University of Hamburg, Germany lippert@informatik.uni-hamburg.de
Isabella Merlo	Universita' di Genova, Italy merloisa@disi.unige.it
Anna Mikhailova	University of Southampton, UK aam@ecs.soton.ac.uk
Alexander Romanovsky	University of Newcastle upon Tyne, UK alexander.romanovsky@ncl.ac.uk
Cecilia M. F. Rubira	University of Campinas, Brazil cmrubira@dcc.unicamp.br
Markku Sakkinen	Tampere University of Technology, Finland msakkine@cs.tut.fi
Andrew Stevens	University of Sussex, UK andrewst@cogs.susx.ac.uk
Tatyana Valkevych	Imperial College, UK tanya@doc.ic.ac.uk
Anand Tripathi	University of Minnesota, USA tripathi@cs.umn.edu

3 Summary of the Call-for-Papers

The call-for-papers for this workshop emphasized its broad scope and our desire to focus the workshop discussions on problems of "complexity" of using and understanding exception handling: Why programmers and practitioners often believe that it complicates the system design and analysis? What should be done to improve the situation? Why exception handling is the last mechanism to learn and to use? What is wrong with the current practice and education?

We invited the researchers interested in this workshop to submit their position papers aiming at understanding why exception handling mechanisms proposed and available in earlier OO languages (discussed, for example, at ECOOP'91 Workshop on "Exception Handling and Object-Oriented Programming"[1]) are not widely used now. We were interested in papers reporting practical experiences relating both benefits and obstacles in using exception handling, experience in using advanced exception handling models, and the best practices in using exception handling for developing modern applications in existing practical settings.

The original plan was to have up to 20 participants. We asked each participant to present his/her position paper, and discuss its relevance to the workshop and possible connections to work of other attendees. The members of the organizing committee reviewed all submissions. The papers accepted for the workshop sessions were posted on the workshop webpage so the participants were able to review the entire set of papers before attending the workshop.

Additional information can be found on the workshop web page:
http://www.cs.ncl.ac.uk/people/alexander.romanovsky/home.formal/ehoos.html.

4 List of the Workshop Papers

1. Jørgen Lindskov Knudsen (University of Aarhus). *Exception Handling versus Fault Tolerance*

[1] Dony, Ch., Purchase, J., Winder. R.: Exception Handling in Object-Oriented Systems. Report on ECOOP '91 Workshop W4. OOPS Messenger **3**, 2 (1992) 17-30

2. Anand Tripathi and Robert Miller (University of Minnesota). *An Exception Handling Model for a Mobile Agent System*

3. Andrew Stevens and Des Watson (University of Sussex). *The Effect of Java Exceptions on Code Optimisations*

4. Alexander Romanovsky (University of Newcastle upon Tyne) and Jörg Kienzle (Swiss Federal Institute of Technology). *Exception Handling in Cooperative and Competitive Concurrent Object-Oriented Systems*

5. Anna Mikhailova (University of Southampton) and Alexander Romanovsky (University of Newcastle upon Tyne). *Supporting Evolution of Interface Exceptions*

6. Bjørn Egil Hansen (DNV) and Henrik Fredholm (Computas). *Adapting C++ Exception Handling to an Extended COM Exception Model*

7. Yolande Ahronovitz and Marianne Huchard (University of Montpellier). *Exceptions in Object Modeling: Questions from an educational experience*

8. Tatyana Valkevych and Sophia Drossopoulou (Imperial College). *Formalizing Java Exceptions*

9. Marta Patino-Martinez, Ricardo Jimenez-Peris, and Sergio Arevalo (Univesidad Politecnica de Madrid). *Exception Handling in Transactional Object Groups*

10. Alessandro F. Garcia and Cecilia M. F. Rubira (University of Campinas). *An Exception Handling Software Architecture for Developing Robust Software*

11. Cristina Lopes (Xerox PARC), Jim Hugunin (Xerox PRAC), Mik Kersten (Xerox PARC), Martin Lippert (University of Hamburg), Erik Hilsdale (Indiana University) and Gregor Kiczales (University of British Columbia). *Using AspectJ™ For Programming The Detection and Handling of Exceptions*

12. Jörg Kienzle (Swiss Federal Institute of Technology). *Exception Handling in Open Multithreaded Transactions*

13. Elisa Bertino (University of Milan), Giovanna Guerrini (University of Genova), and Isabella Merlo (University of Genova). *Do Triggers Have Anything To Do With Exceptions?*

14. Christophe Dony (University of Montpellier). *A Few Interesting Ideas Ignored (in Java) or Forgotten from the Past*

5 Thematic Classification and Summary of Contributions

The workshop program was organized into four sessions. We summarize below the important ideas that were presented and debated in these sessions.

5.1 Session 1: Linguistic Issues: Existing Models and New Ideas

This session addressed linguistic issues of exception handling. The session was a mixture of presenting existing exception handling models in a new light, presenting existing models that have unfortunately been ignored for several years, along with new models of exception handling, such as those needed for applications built using component software.

In his talk "Exception Handling versus Fault Tolerance", Jørgen Lindskov Knudsen discussed the many proposals that have been put forward on defining language constructs for handling exceptions. Most of these are dynamic in nature (i.e. the exception is handled by some component found by examining the dynamic calling sequence

leading to the exceptional occurrence), while others are static (i.e. the exception is handled by some component found by analyzing the static context of the exception occurrence). Jørgen Lindskov Knudsen then discussed dividing error handling into two distinct disciplines: exception handling and fault-tolerant programming. Exception handling deals with the handling of well-defined error conditions within a well-defined system or framework. On the other hand, fault tolerant programming deals with error handling in all other cases: ill-designed systems, faults in the exception handling code, errors originating from outside the system or framework. Jørgen Lindskov Knudsen then reported his experiences with the static exception handling mechanisms of BETA. He elaborated that static error handling is in fact an effective exception handling mechanism, but also that it is difficult (and in some cases impossible) to use it for fault tolerant programming. Jørgen Lindskov Knudsen therefore proposed to introduce a dynamic error handling mechanism to be used for effective fault tolerant programming. On the other hand, he stressed that the static exception handling model should be used almost exclusively, since it gives the most well-designed exception handling, integrated with the OO tradition, and reserve dynamic exception handling only to those cases where no other error handling solution can be found.

In the second talk Christophe Dony discussed "A Few Interesting Ideas Ignored (in Java) or Forgotten from The Past". The aim of this talk was to remind the exception handling community of previous research results, that seem to have been ignored, more or less, in the current research – ideas that should not be forgotten, and which might inspire future research. The first idea presented was related to the scope and extent of handlers. The work carried out for handling exceptions in procedural languages such as PL/I, Clu, Ada or Mesa, has shown that modularity and the notion of fault-tolerant encapsulation rely on a stack oriented handler research, on the ability for signalers to raise exceptions to the operation callers, and for the callers to handle the exceptions raised by inner modules. However, Christophe Dony illustrated, that Smalltalk-80, in its standard blue book specification, has shown the utility of attaching handlers to object classes (let's call them class-handlers), providing an OO style of handling exceptional situations and of extending and reusing code in which exceptions are signaled. Christophe Dony also presented a way, implemented in its systems for the languages Lore and Smalltalk, to combine class handlers and a stack-oriented handler research discipline to combine both advantages.

Secondly, Christophe Dony presented issues from the exception handling system from Flavors system. Flavors made it popular to describe exceptions as hierarchically organized classes on which methods can be defined, allowing them to be called within handlers to deal with the corresponding exceptional situations in a standard way. Derived from this principle, Christophe Dony discussed relationship between meta object programming (MOP) and exception handling: various system including Flavors, Clos, ParcPlace Smalltalk, Lore have proposed various MOPs allowing system developers to adapt a language basic exception handling mechanism to the application particular needs or to extend the language with new models (e.g. component model).

Finally, Christophe Dony discussed communication protocols between signalers and handlers. The Flavors system proposed signaling protocols allowing signalers to specify, each time an exception is raised, which resumption solutions are currently available. Handler can then choose a specific resumption solution by invoking a corresponding method defined on the current exception class.

The following talk was by Bjørn Egil Hansen on "Adapting C++ Exception Handling to an Extended COM Exception Model". Bjørn Egil Hansen discussed the BRIX exception system which was designed with the following objectives: to simplify and

minimise the amount of code needed for exception handling; to increase system robustness and correctness; and to increase flexibility in system configuration and evolution. By having a notion of both specified and unspecified exceptions, the BRIX exception model allows recovery from more situations than those typically specified in an abstract interface. Exceptions from a component are categorised according to two orthogonal criteria: (i) component state: controlled or uncontrolled and (ii) cause of exception: operational or implementation. Only in the case of controlled operational exceptions should the developer make an effort to mask the exception or to recover and propagate the exception to increase the overall robustness of the system. For the other cases this is generally a waste of effort and the cause of the exception should instead be eliminated, i.e. by fixing lower-level bug or allocating required resources. The BRIX exception system gives extensive support for doing exception handling in C++ in a COM environment according to the model above, providing mechanisms for:

- conversion between COM and C++ exception representation
- shorthand notation for various propagation and handling
- default mechanisms for typical situations
- integration of pre/post-condition checking and general assertion checking
- simple user-interaction enabling local handling of exceptions by involving user
- rich support for logging of exceptional events, simplifying debugging both during development and operation and helping operators in identifying lack of resources.

The fourth talk in this session "Using AspectJ™ for Programming the Detection and Handling of Exceptions" was delivered by Martin Lippert. Martin Lippert took an existing framework written in Java, the JWAM framework, and partially re-engineered some of its exception detection and handling aspects using AspectJ, an aspect-oriented programming extension to Java. Aspect-oriented programming (AOP) has been developed to provide programmers with mechanisms to implement, abstract and compose crosscutting concerns. AOP language design is based on understanding the structure of common crosscutting concerns and developing mechanisms that can capture that structure and can be composed to develop more complex crosscutting structures. Martin Lippert presented how AspectJ™ was used to re-implement the design-by-contract checks inside the JWAM framework, resulting in the amount of code used to implement the contract checks using “contract aspects” – aspects containing only the contract checking code – being considerably reduced. It was also reported, that the contract aspects reduce the risk of forgetting the proper contract checking implementation in a subclass without losing the possibility to redefine pre- and post-conditions using sub-aspects. Martin Lippert also reported using AspectJ™ to re-implement some Java exception handling code inside the redundancy framework since some redundancy was identified in those parts of the framework implementation. Using an aspect to extract the exception handling code out of the functional implementation of the class, the exception handling code can be reused for more than one catch block over more than one class. As a result of the presentation there was some discussions about some general problems using AOP (e.g. the lack of good tool support) as well as the improvement of the presented quantitative numbers using not only the framework but also some applications build on top of the framework for the measurements.

The last talk in this session was given by Cecilia Rubira on "An Exception Handling Software Architecture for Developing Robust Software". Fault-tolerant OO software systems are inherently complex and have to cope with an increasing number of error conditions to meet the system dependability requirements. Dependable OO software detects errors caused by residual faults and employs fault tolerance measures to restore normal computation. The presence of exception handling facilities can reduce

software development efforts since they allow software designers to: represent errors as exceptions, define handlers to deal with them, and use an adequate strategy for exception handling when the occurrence of an exception is detected. Moreover, OO systems may be consisted of various threads (or processes) executing methods concurrently on objects. Exceptions are more difficult to handle in concurrent OO systems than in sequential ones especially because of cooperative concurrency. That is, several concurrent threads usually cooperate to perform some system activity, giving rise to very complex concurrent interactions. The present interest in software architectures and design reuse motivated the authors to develop an exception handling software architecture for building fault-tolerant software. A software system quality requirements (such as dependability and persistence) are largely permitted or restrained by its architecture; so, the authors' believe that if an appropriate architecture (in particular one that supports exception handling) is chosen since the outset of the design phase, then a proper use of exception handling throughout the development life cycle of a system can be obtained. The proposed architecture provides a generic infrastructure that supports uniformly both concurrent and sequential exception handling. Moreover, the exception handling architecture is independent of a specific programming language or exception handling mechanism, and its use can minimise the complexity caused by handling abnormal behaviour. The architecture is composed of four well-defined components: the Exception component, the Handler component, the Exception Handling Strategy component, and the Concurrent Exception Handling Action component. The structural and behavioural aspects of the components are described by means of a set of design patterns. The proposed patterns allow a clear and transparent separation of concerns between the application functionality and the exception handling facilities, easing the task of building fault-tolerant software with high assurance.

The talks and the discussions tried to develop a clear relationship between various commonly used terms and concepts, particularly in regard to relationship and differences between error handling and exception handling. Dependability community has concepts of faults, errors, failures and for these people exception handling is a means for tolerating faults. Knudsen's talk characterised exceptions as anticipated failure conditions within a well-define framework in contrast to errors which are unanticipated conditions that arise due to faults or from outside of the design framework. The discussions ensuing from the this talk led to the conclusion that the static model of exception handling chain is found to be adequate in most cases except in cases designing fault-tolerant systems. Dony's presentation reminded of many important research contributions made in the past in the area of exception handling in OO systems. These past contributions in the area of exception handler scope, use of meta object programming, and flexible models to deal with different resumption requirements should not be forgotten.

5.2 Session 2: Semantics and Implementation

This session addressed issues related to formal descriptions of exception handling constructs in a modern language such a Java, impact of exception handling on code optimization during compilations of Java programs, and possible use of exceptions in Java-based OO database management systems to realize database triggers.

Tatyana Valkevych presented an overview of a formalization of the static and dynamic semantics of Java related to exceptions in her talk on "Formalizing Java Exceptions". The most challenging Java feature with respect to exceptions are the dual nature of exceptions whereby an exception may act as a normal object until explicitly thrown, and the difference beginning execution of a throw-statement and the actual throwing of the exception, whereby the latter exception may be different from the exception indicated

after the throw-statement. This talk distinguished between normal execution, from which no unhandled exception escapes, and abnormal execution, from which an unhandled exception escapes. At the level of objects, one can distinguish those not thrown (these exhibit normal behaviour) and those which are thrown (these exhibit abnormal behaviour). This work develops an operational semantics that gives a concise and precise model of Java. In the type system it distinguishes normal types which describe the possible outcome of normal execution, and abnormal types which describe the possible outcomes of abnormal execution. The type of a term consists of its normal type and its abnormal type. In this approach the meaning of the subject reduction theorem is stronger than usual: it guarantees that normal execution returns a value of a type compatible with the normal type of the term, and that abnormal execution throws an exception compatible with the abnormal type of the term. The formal description proposed in this talk consists of half a page of operational semantics, which is considerably shorter than several pages in the language manual. The authors made the case that their description is a good basis for more precise and succinct language manuals of the future.

In his talk on “Effect of Java Exceptions on Code Optimisations", Andrew Stevens outlined a support for exception based control flow in Java. This talk showed the effect of this support on compiler driven static analysis in quantified terms. The framework used to support exceptions in Java programs is shown to exist at three levels; the language syntax, the bytecode class files and from within the Java Virtual Machine (JVM). A model was described that divides the types of exceptions that can be raised into three categories; explicit, implicit-synchronous and implicit-asynchronous. The explicit exceptions are those described by the 'throw' statement, implicit-synchronous are those caused by an instruction breaking a semantic rule of the language and implicit-asynchronous are generated by JVM internal errors or a *Thread.stop* call. The presentation continued by outlining some standard compiler analysis structures, namely basic blocks, control flow graphs and call graphs. The effect of the exceptions on these structures was described as follows. For basic blocks, an instruction that potentially generates an exception marks the end of the block. The control flow graph and call graph are structures that attempt to map potential routes through the executing code. Since the destination handler of a raised exception depends on runtime type information and call context, the flow graphs must mark the destination as unknown (or dynamic). Further, Andrew Stevens outlined an experiment to show basic block sizes and flow graph dynamics with implicit exceptions enabled and disabled. The results show that such exceptions greatly reduce block sizes and increase the dynamics by a factor or four. Since both these effects can reduce the power of compiler optimisations, a future system that reduces the number of potential exceptions is described.

The third presentation in this session, "Do Triggers Have Anything To Do With Exceptions?", was by Isabella Merlo. The OO paradigm is playing a crucial role in getting the programming language and the database fields closer to each other. The Java programming language is establishing itself as the most popular OO programming language, and the increasing need of persistently storing Java objects, through minimal modifications to applications, led most OO database management systems (OODBMs) to develop a Java version. The ODMG standard also has been enriched with a Java binding. Current relational and object-relational DBMSs provide an important functionality, namely triggers. This typical database functionality has no direct counterpart in OO programming languages, and though trigger usefulness has been recognised also in the OO context and a considerable amount of research has been carried out in the area of active OO databases, triggers are not supported in current commercial OODBMSs. A relevant feature of the Java language is represented by its exception mechanism, which represents a very powerful mechanism for handling error situ-

ations. Since one of the most relevant functionalities achieved through triggers in the database context is integrity constraint enforcement, the two notions seem to be potentially related. In her presentation, Isabella Merlo provided an overview of triggers, then compared it with exception handling. This presentation was followed by a long discussion on how the two notions can be related, and if the support of exceptions can offer any help or insight towards the introduction of triggers in Java-based OO database management systems. This certainly remains an open question for further investigation.

The session clearly highlighted the impact of exception handling in generating efficient code through code optimization. Also, in this session the relationship between triggers and exceptions generated quite an extended debate.

5.3 Session 3: Concurrency, Distribution and Mobility

Developing exception handling models which are adequate for designing modern concurrent, distributed and mobile applications was the theme of the third session consisting of four presentations followed by a discussion.

Anand Tripathi delivered the first position paper entitled “An Exception Handling Model for a Mobile Agent System”. Ajanta is a Java-based system for programming distributed applications using mobile agents in which agents encapsulate code and execution context along with data. A model for exception handling in mobile agent applications in the Ajanta system was presented. When an agent arrives at a server, the server creates a thread to execute the agent's method that was indicated in the transfer request. During its execution, an agent may encounter some exception conditions. Some of these may be anticipated by the programmer and handled within the agent's code. If, however, an exception is not handled by the agent, the exception is signalled to its server thread. To facilitate exception handling, associated with each agent is a guardian object, which is stationary in the system. The name of the guardian object appears in the agent's tamper-proof credentials. When an agent encounters an exception that is not handled by its code, it is co-located with its guardian. In this case the agent carries the exception object identifying the exception. On co-location, the agent invokes the report method of the guardian object and passes to the guardian a reference to itself. The guardian can then examine the agent's state to perform the appropriate recovery actions. Developing of guardians is a responsibility of the application developers. Ajanta provides some additional mechanisms using which the guardian can perform application-wide recovery actions. For example, in an agent-based program consisting of a group of agents, the recovery action may require termination of some or all of the agents in a group when one of the group members encounters a non-recoverable exception. Developing exception handling mechanisms for mobile systems is a vital and difficult topic, unfortunately mobile system developers do not often pay enough attention to this issue. The research reported in this talk is in many respects unique as it offers a general exception handling mechanism which takes into account the specific characteristics of such applications and because it has been designed together with the mobile system.

In his talk "Exception Handling in Cooperative and Competitive Concurrent Object-Oriented Systems" Alexander Romanovsky outlined the exception handling mechanisms developed for concurrent OO systems and showed likely directions of future research. This presentation emphasized that considerable effort has been devoted to developing exception handling models for sequential systems and that a common understanding exists in many topics in the field. The situation is different in concurrent systems. Although several schemes combining concurrency and exception handling have been proposed, this research is still scattered and the majority of concurrent systems use sequential exception

handling. It is the author's belief that the latter is abnormal because exception handling features should correspond to the programming features used in system design. The choice of a way to introduce exception handling depends on the way concurrent systems are to be developed and structured because exception handling is an issue of the system design, and language features should assist in and impose proper design. Exception handling is closely coupled with program structure and therefore the way in which the dynamic execution of concurrent systems is structured influences possible ways of introducing exception handling into such systems. Following C.A.R. Hoare, J.J. Horning and B. Randell, all concurrent systems are classified into cooperative, competitive and disjoint ones, with the corresponding structuring techniques such as atomic actions, atomic multithreaded transactions and blocks of code (the latter is effectively identical to the sequential system structuring). Several schemes have been proposed for introducing first two approaches into concurrent systems, but only rarely do they incorporate exception handling features (few examples were discussed in the talk). And even when they do, they neither provide a general exception handling model nor fit the main principles of OO programming properly. Although this is an area of very active research, there are still many unclear points and unsolved problems. A general common understanding does not seem to exist. Two following presentations, which extend in many ways the seminal research on Argus system, have addressed some of the topics discussed in the talk.

In the next presentation "Exception Handling in Transactional Object Groups" Ricardo Jimenez-Peris proposed a model which combines transactions and group communication and includes an exception handling mechanism. Transactions and group communication are two fault-tolerance techniques that have evolved separately for a long time. However, these two techniques are complementary, and distributed systems can benefit from their integration. In particular, transactional object groups can be used in clustered systems to provide high availability and high performance transactional services. Replicated object groups can tolerate failures without interrupting running services increasing their availability. Co-operative object groups perform services in parallel taking advantage of distribution and thus, improving their performance. Exception handling mechanisms play an important role in this kind of systems. Firstly, exceptions can be used in combination with transactions to cope with anticipated errors within a transaction by providing automatic backward recovery for unhandled exceptions. Secondly, object groups can raise concurrent exceptions due to their concurrent and distributed nature. In the proposed model exception resolution is used to yield a single exception to the client when several servers raise exceptions. New exception resolution algorithm is developed to resolve exceptions at two levels: local and distributed. Local resolution is first applied to concurrent exceptions raised within an object. Then, distributed resolution is applied to resolve the resulting exceptions of the object group. This two-level hierarchical resolution is more rational than a global resolution, because local exceptions that are more related among themselves, are resolved first. This talk reported several important results which are outcome of an active research effort on extending transactional services with an ability to develop co-operative systems. It is very important that the authors have realised that new models to be developed in this area should always include features for disciplined and powerful exception handling

The last talk of the session "Exception Handling in Open Multithreaded Transactions" was presented by Jörg Kienzle who proposed a new model for providing transaction support for OO concurrent programming languages. This support is enhanced by a full-fledge exception handling mechanism incorporated into the transactional support. The aim of this research is to achieve a smooth integration of the native language concurrency and the transactional features, so that the use of the concurrency features provided by the programming language is not restricted inside a transaction. A new transaction model called *Open Multithreaded Transactions* that meets this requirement was presen-

ted. Threads inside such a transaction may spawn new threads, but also external threads are allowed to join a running transaction. A blocking commit protocol ensures that no thread leaves the transaction before its outcome has been determined. Exceptions can be handled locally by each thread. Unhandled exceptions crossing the transaction boundary cause the transaction to abort. Exceptions are propagated to all participant threads when a transaction aborts. An Ada 95 implementation of this model is under development now. Although the discussed results are preliminary, they are important to researchers and practitioners because of two main reasons: this transactional model offers a good balance of the ways multiple transaction participants are affected by each other's exceptions and because this model can be applied in any concurrent language (e.g. in Java).

5.4 Session 4: Design and Evolution

The last session was dedicated to the design and evolution of OO programs dealing with exceptions.

In the first presentation, Yolande Ahronovitz brought to the fore the important issue of how to design hierarchies of exception classes. Almost all OO languages now represent exception as classes and occurrences of exceptional situations as instances of these classes. This idea can be found in Nixon's thesis and has been developed in various systems such as the Flavor one in the 1980s, it provides a simple way to define both the internal representation and behavior of exceptions occurrences with capabilities for extendibility and reusability. If many things have been said on the benefit of such an organization, the problem of modeling exception classes has not been studied much: literature gives good advice, but lacks concepts about how to think up exceptions, and UML lacks concepts to model exceptions as classes. Anyone that has written an OO application has been confronted to issues such as the following ones: Should I have an abstract exception class for my whole application? Should I create an abstract class "StackException" generalizing all the exceptions related to the "Stack" class I am designing? Should "StackException" be a subclass of "MyApplicationException" or of "CollectionException"? Should the various exceptions related to the Stack class be represented by slots of "StackException" or by subclasses?

Yolande Ahronovitz and Marianne Huchard have proposed a kind of guide, based on object model elements, for finding exceptions at modeling stage, and for organizing them into classes hierarchies. More precisely, they have proposed to define a general exception class *ExcTypeX* for each type *X* found in a program, that exception being the root of all exception classes associated with *X*. Under this general root are defined a subroot for each feature associated with *X:* primitive attribute, method, association with other type *Y*. For an association, two subroots are created: one for the exceptions which encapsulate *Y's* exceptions and one for the exceptions which are induced by the association itself. In order to reflect the classes hierarchy of the application, if *SX* is a subclass of *X*, the general root *ExcTypeSX* is a subclass of *ExcTypeX*. Exceptions associated with *SX* can either be related to its own features, or be specializations of exceptions defined on inherited features, or be exceptions generated by new constraints on inherited features. These cases are distinguished in the hierarchy built under *ExcTypeSX*. The last case (new constraints on inherited features) is not allowed in many programming languages (e.g. Java, C++). The way the hierarchy is built gives a possible solution to handle it. The complete hierarchy must not necessarily be entirely implemented; yet, it is helpful in accurately handling exceptions. Besides this guide, an original model of composite exception has also been proposed, to allow users to signal several problems at the same time.

Towards the end of her presentation, Yolande Ahronovitz raised several important

issues related to her work, and these issues were discussed during the workshop session. Certainly, the main issue is how to merge the type-based classification of exceptions, presented in this session, with semantically-based exception hierarchies as found in other systems. For example, it is usual to classify exceptions according to the way they can be handled (fatal or recoverable exceptions) or to classify them around semantic categories (arithmetic exceptions, file-system exceptions). Both classifications could cohabit provided that the implementation language supports multiple inheritance.

The second talk of this session was presented by Anna Mikhailova. Her talk on "Supporting Evolution of Interface Exceptions" focused on problems connected with the evolution of OO applications in presence of methods signaling exceptions. Interface exceptions are explicitly declared exceptions that a method can propagate outside. Evolution in OO systems is achieved via sub-classing and method overriding. When sub-classing is done without conforming to sub-typing rules, runtime type exception can occur. That is why languages wanting to statically check that programs are type safe imposes constraints on subclassing and overriding rules (e.g. contravariance or at least invariance for methods parameters types). In presence of exceptions, to be type-safe, an overriding method interface exception should not include exceptions that are not declared by its corresponding overridden one. Java, for example, imposes such a rule. Indeed, clients of a server class are only prepared to handle the exceptions explicitly declared in the interface of this class and might be unaware of the existence of its subclasses. If a subclass of the server class signals new exceptions and if it is passed by subtyping to the client, the client might get invalidated, as it will be asked to handle an unexpected exception. In her talk, Anna Mikhailova explained, via a convincing example, why it is desirable to introduce new exceptions in subclasses and proposed two complementary solutions to achieve this in a type-safe way. To solve the problem of non-conforming interfaces resulting from the addition of new exceptions, the first proposal uses what she calls "rescue handlers". Rescue handlers are some kind of class handlers (see section 5.1) and are default handlers. They are invoked when an exception is signaled within a method of the class the handler is associated with and when the method client does not itself provide a handler. Thus, when clients do not know how to deal with new interface exceptions of their servers, the rescue handler attached to the server class steps in to rescue the situation. Secondly, while rescue handling is suitable for the top-down system development approach, when there is a need to introduce an interface exception in a development step, it is of little help in a bottom-up approach. In this case, the author proposed to apply a classical forwarding technique (the wrapper pattern) which is an architectural solution used for solving closely related interface mismatch problems.

It was noted that a general solution to the evolution problem would be to systematically define an exception signaled in a subclass method as a sub-exception of an exception signaled in the corresponding superclass method. This would correspond to the organization of exception hierarchies suggested by Ahronovitz and Huchard. Unfortunately, this is only possible when the subclass wants to signal a new exception and not an existing one.

6 Conclusions

The 1970s have seen the development of exception handling systems dedicated to procedural programming. Such tools and the associated mechanisms have been designed to enhance software reliability, reusability, readability and debugging. The 1980s have seen the emergence and dominance of the OO technology that has brought new ways to make program modular and reusable. During the 1980s and 1990s, exception hand-

ling systems were integrated into all OO languages. The first ECOOP workshop on exception handling and OO programming held in 1991 was oriented towards the specification, understanding and implementation of these systems. The wide range of topics addressed by the participants of this new workshop has shown us many new research directions for exception handling in OO systems.

The year 2000 workshop proposed a clear representation of many open and challenging problems that need to be understood and addressed by the programming language designers, software engineers, and application system programmers. Many problems still exist, and there are many new research opportunities in a variety of emerging application domains where exception handling mechanisms have to be application specific. The main conclusions gained from the presentations and discussions held during the workshop are as follows:

- The evolution of exception handling models for standard OO programming is still an open issue:
  - A new model for BETA has been presented that proposes an interesting mix of static and dynamic handler research allowing BETA to deal separately and differently with failures and exceptions
  - Handlers associated with classes, as invented in the 1980s, have disappeared from C++ and Java.These kinds of handlers could be reinvestigated because of their intrinsic interest and because it has been shown that they could solve problems related to evolution in presence of static typing
  - More generally, the workshop also reminded the participants that many of the important past contributions are forgotten and the community should reinvestigate their utility
- Today's tendency is to use statically-typed OO languages and to look for formal models for the integration and use of exception handling in fault-tolerant systems. A set of new research activities is advocated towards the definition of exception handling systems compatible with static and correct typing in OO languages. Besides, the quest for formal models and type safety gives rise to great challenges in the presence of some models such as the resumption. These models, although interesting in term of expressive power, makes program analyses very complex, and might therefore be abandoned (or better formalisms should be developed)
- The representation of exceptions by hierarchically organized classes is now quite a standard but the design issues related to that representation are not solved and design languages such as UML do not correctly take it into account. There is a lack of proper frameworks and principles for designing and analyzing exception hierarchies and exception code in large-scale systems. The separation of the main code of the program from the exception code is an old issue for which new advances can be considered thanks to new techniques for the separation of concerns in programming, such as aspect-oriented programming
- New linguistic mechanisms are needed which take into account recent developments in designing new excerption handling mechanisms for concurrent, distributed, web and mobile OO systems. Future languages intended for such applications should incorporate adequate exception handling mechanisms
- Exception handling mechanisms should be adequate to the language paradigm and the main language features (e.g. inheritance, concurrency), to the system development techniques, to the way systems are structured and to the application-specific characteristics of the system to be designed
- Safety of exception handling mechanisms is the prime concern which should be taken into account while developing new mechanisms, because exception hand-

ling mechanisms are often chosen for developing modern complex applications with high dependability requirements

- Exception handling is not just a linguistic issue. There is a need in applying disciplined exception handling at all phases of the system development. To do this properly the methodologies should support a smooth transition between models used at different phases
- Another promising direction of the research in this area is to enforce good development techniques which use exception handling with a set of well-developed architectural and design patterns and idioms; this can serve as a sound solution complementing research focused solely on developing new linguistic approaches
- New flexible exception handling mechanisms applicable in all phases of system development should be developed; such mechanisms should be both general and flexible enough to allow for simple adjustment or customization
- It is difficult to expect that the research community will come up with a linguistic mechanism which can solve all current and future problems, this is why research on integration of exception handling mechanisms found in different languages, systems and methodologies (e.g. Java and CORBA) is getting more important

The workshop was a clear success in its primary goal of creating a forum to discuss and debate the important research directions in the area of exception handling in object systems. The wide range of topics addressed by the participants was a clear representation of many open and challenging problems that need to be understood and addressed by the programming language designers, software engineers, and application programmers.

Finally and most importantly, this workshop reinforced everyone's view that such meetings are of tremendous value to the research community to provide a forum for direct and quick exchange of ideas on the importance of various issues and problems facing the community. The participants felt that it would be beneficial to have another meeting on this topic in the next 2-3 years.

Acknowledgements: The authors want to thank the participants for making this workshop a success with their unique contributions. This report reflects the viewpoints and ideas that were contributed and debated by all these participants. Without their contributions, neither the workshop nor this report would have materialized.