Exception Handling and Asynchronous Active Objects: State of the Art Elements

Christophe Dony, Christelle Urtado, Sylvain Vauttier

Exception Handling and Asynchronous Active Objects: State of the Art Elements

Christophe Dony[1], Christelle Urtado[2], Sylvain Vauttier[2]

[1] LIRMM - CNRS and Montpellier II University - 161 rue Ada
34 392 Montpellier - France
dony@lirmm.fr

[2] LGI2P - Ecole des Mines d'Alès - Parc scientifique G. Besse - 30 035 Nîmes - France
{Christelle.Urtado, Sylvain.Vauttier}@site-eerie.ema.fr

1 Introduction

The ANR Facoma project puts a strong emphasis on exception handling as one of the tools for ensuring reliability of cooperative multi-agent applications and more generally with applications built on top of active objects. The related task in the project is to install an appropriate exception handling system in the DIMA agent language and to adapt it so that it can be used properly together with the DARX replication system.

Active objects are "*objects having their own computing resources i.e. their own private activity*" [1], or, said differently, objects "*decoupling method execution from method invocation*" [2]. *Asynchronous Active Objects* (AAOs) come in many forms (actors, agents or components), with various interaction schemes (request / response or publish / subscribe [3]) and various forms of asynchronous communication (one-way or two-ways). They are more and more used as, for example, in multi-agents systems [4], in some distributed components architectures such as J2EE's with *Message Driven Beans*, in programming languages dedicated to grid applications (e.g. [5]) or to wireless devices on top of mobile networks as in [6]. AAOs, particularly in these new contexts, raise numerous issues regarding exception handling that have only been partially studied.

The program structures for handling exceptional events [7,8,9,10] have been designed to implement software entities able to return well defined and foreseen answers, whatever may happen while they are active, even though an exceptional situation occurs. The end of the 1970s saw the development of exception handling systems dedicated to procedural programming. All specifications have all been influenced by Goodenough's seminal paper [7]. Well known implementations include MESA[11], CLU[9] or ADA[12]. Exception handling systems have later been specified for or integrated into object-oriented languages at the end of the 1980s (Zetalisp+Flavors, CommonLisp(+CLOS) [13,14], Eiffel[15], Smalltalk[16], C++[17], or more recently in Java.

While the masterpieces of a generally accepted solution for exception handling in sequential programs have been defined in the above quoted systems, this is not yet the case for concurrent systems [18], even if some agreements and

many proposals [19] exist. When systems with asynchronous communications are concerned, research works are still much more scattered. Initial actor languages included basic proposals to cope with exceptions [20] in which handlers were some dedicated actors, ancestors of today's exception supervisors, that had the same lacks, regarding handler contextualization (see Sect. **??**), as Smalltalk or Ada initial lexical-scope handlers. Asynchrony has more recently motivated many research works in various contexts [21,22,23,24,25,26,27,28]but they only partially address AAO needs. Actually, agent systems are the AAO context in which exception handling proposals are the most achieved. However the supervisor model described in [29,30] does not properly handle the contextualization issue. Guardian [31,32] is a general and powerful solution which nonetheless proves to be complex to master and use. As explained in [31], *"Often exception handling in a program is the most complex [...] part of the system [...] and has to be either simplified or taken out of the hand of the average programmer"* and a solution for this is to *"separate global level exception handling from the application agents"*.

Concurrent programming systems fall into three main categories when exception handling is considered. These categories correspond to the kind of concurrency that is supported [18]. This directly determines how AAOs can interact, and, as part of their interactions, how exceptions can be signaled between them and handled. We consider existing research on exception handling from the point of view of these three categories and compare the existing capabilities with the one of our SAGE system[33,34].

2 Isolated Concurrency and exception handling

Isolated concurrency is provided by standard programming languages such as Java. Its goal is to allow several AAOs (threads) to execute concurrently in a shared context (the address space of a virtual machine) as if each of them was the only existing AAO. To achieve this, the system enforces that the activity of an AAO does not interfere with another one. For example, locks are managed on shared resources in order to transparently serialize concurrent accesses to them. In the same way, no standard means is provided to send information from a thread to another. When an exception is raised in an AAO, it is signaled along its own execution stack (in its separated execution thread). When the exception is not caught and reaches the top level of the execution stack, the AAO is destroyed by the system (the thread is discarded by the thread manager). The other AAOs are not warned of the failure in order to maintain their isolation.

3 Cooperative Concurrency and Exception Handling

Isolated concurrency is only suitable when strictly parallel computations are to be managed,for example when handling requests from different clients. But when a set of entities are intended to participate together to the achievement of a global activity, means to handle their cooperation are then required. More

specifically, in such forms of cooperative concurrency, there is a crucial need to manage how the individual failure of an AAO impacts the global activity and, as a consequence, should impact the activity of other entities.

3.1 Monitors.

A first technique is to provide specific entities which role is to monitor other entities and to implement how errors are to be handled when the global context of the monitored entities is considered. Java proposes that a thread can belong to a *ThreadGroup*. When an exception is raised and uncaught in a thread T , it is then signaled to the thread group to which T belongs. A unique handler, that catches all the signaled exceptions, can be associated with the thread group. This allows basic actions to be carried out, such as to kill threads that are still running in the thread group, in order to terminate the whole activity of the group. Some SMAs provide such a mechanism in the form of supervisor agents. Supervisors are agents that monitors other agents in the system and to which exceptions are signaled.They are used, for example, to react to the death of an agent (killed by an uncaught exception) and warn other agents that it cannot be reached any more. In Erlang [27], supervisor processes can be tied to other ones to be informed of their termination. In Oz [25], asynchronous exception related to distribution are handled thanks to dedicated monitors. Monitors enable a good separation of concerns, because they keep behaviors dealing with errors well separated from behaviors dealing with normal activities. However, they raise encapsulation and contextualization issues. When used for AAOs, monitors can only perform external, platform-level, generic actions such as to suspend, restart or destroy an AAO. Monitors are finally somehow restricted to the handling of generic exceptions because they have no access (unless breaking encapsulation) to objects' internal state and, generally, to any contextual information about the cause of the exception. This drastically limits the applicability of monitors when specific errors, regarding the specific coordinated activity of a set of entities, are to be handled.

3.2 CA Actions.

A common solution, in systems that tackle this contextualization issue, is that collective activities of coordinated concurrent entities must become explicit, in order to structure the global execution contexts and provide a support to handle exceptions. A CA Action [18] allows the representation of a collective activity to which different entities, called participants, contribute concurrently. Different variants of this concept, along with different EHSs, have been proposed. In [35], a CA Action is defined as a sort of contract that ties together all the participant entities. As a part of the contract, these participants must provide support for the handling of a common set of exceptions. When an exception is raised by one of the participants, it is signaled to the others. This way, all the participants to a CA Action suspend their individual activity (and thus suspend the execution of the whole global activity). The system then enforces a synchronization point between

all the participants. When multiple, concurrent exceptions are signaled, this policy ensures that a same set of exceptions is finally signaled to each participant. Each participant resolves this exception set thanks to an exception resolution tree provided by the CA Action (and thus common to all the participants): this entails that each participant finally handles the same resolved exception. However, the handlers that are eventually triggered are specific to each participant and are cohesive parts of their behavior. This model addresses the contextualization issue stressed above. Exceptions pertaining to a global activity are handled by having its participants contribute collectively to their treatment, as a result of the coordinated execution of their own handlers. One concern with such a model is the cost of the coordination between the participants. Indeed, it implies the exchange of numerous messages in order to inform the other participants of exceptions and of execution suspensions. Moreover, it entails a strong coupling between the participants as it requires a common set of exception types and a common exception resolution tree to be used. This model is therefore not perfectly suited for highly distributed and open systems.

3.3 Guardians.

Among other things, various improvements have been introduced to the above model in [31,36,32]. A CA Action is monitored by a special participant, called a "guardian". Participants signal to the guardian exceptions that are global to the CA Action. The guardian then suspends the execution of all the participants, while collecting concurrent exceptions. The set of exceptions collected by the guardian is then resolved thanks to a first set of rules that determines what unique global exception is to be handled. Next, a second set of rules is used to transform the global exception into the specific exceptions that will finally be signaled to each participant. This way, only one participant, the guardian, needs to track the exceptions and the status of other participants. Moreover, the cooperation of the participants, when handling global exceptions, is defined by the set of rules of the guardian. Rules are tailored to adapt to the specific behavior of each participant, so that no predefined requirement is to be imposed to the participants. Providing a complex and powerful solution, Guardian is especially relevant to deal with exceptions related to shared environments where all participants can effectively cooperate to restore a consistent state. This kind of exceptions encompasses system-level exceptions that warn of the faulty state of some shared resource (disk, memory, network, ...). SaGE provides a simpler solution when handling exceptions related to collaborating pairs of objects such as clients and servers.

4 Collaborative Concurrency and Exception Handling

Models discussed in the previous section indeed share the idea that when an exception occurs in the context of a collective activity, handlers are sought and

executed in all its participants. Besides, in situations in which couples of entities collaborate together, for example when a server informs its client that it has failed to achieve some requested service, signaled exceptions are to be handled in the context of the caller. Exceptions are therefore much more efficiently handled as responses sent by the server than as broadcasted information. We have primarily oriented the Sage solution in that way and have defined the following key requirements : encapsulation and reactivity enforcement, ability to write context-dependent handlers, ability to coordinate and control group of active objects collaborating to a common task, ability to configure the exception propagation policy by defining *exception resolution functions*, ability to immediately handle exceptions that are critical or to only log under-critical ones until their conjunction enables a diagnosis to be established. Sage proposes dynamic scope handlers associated to requests, services and objects. Resolution functions can be defined at the service level, which is the place where collaborative tasks can be coordinated. They come together with a signaling primitive, a handler search algorithm and a handler invocation mechanism that take into account the execution history and, when possible, work asynchronously to improve object reactivity.

Many other systems for asynchronous programming use "future" objects [21,?,25]. Futures are response holders that are immediately returned to client entities when they asynchronously request a service to a server AAO. When a client needs to use the value of a response, it tries to access the value of the corresponding future. If no value is yet bound, the client AAO can perform a blocking wait. When an exception is bound to a future instead of a standard value, it is signaled to a client when the future is read. The client then usually handles it with some classical built-in *try-catch* like constructs. The main advantages of such a solution are its simplicity and its ability to be seamlessly integrated to existing programming languages. Its drawback is that it does not cope with complex situations. For example, when requests are sent concurrently to different servers, it is difficult to foresee the best order in which futures should be read in order not to wait for an unbound response while others are yet available and could be treated. This is one of the reasons why we think that reactive AAO models are more interesting for exception handling. With futures, exceptions cannot be treated as soon as possible and can sometimes be simply lost when some futures are not read. In a reactive system, like the one in which we have specified SaGE, exceptions are signaled asynchronously by sending messages and can therefore be treated as soon as they occur. The implementation of SaGE in a future-based context has not been done yet but the resulting system would be more limited than today's one.

Erlang [27] has a sophisticated EHS to deal with exceptions within concurrent processes and also proposes an asynchronous message sending based solution to signal process termination exceptions from one process to another. In Erlang, messages that contain exceptions cannot be distinguished from others and, as a consequence, the handling of asynchronous exceptions can only be *ad hoc*. On the contrary, SaGE [37,33,38] carry exceptions with messages that, when received,

trigger a full-fledged EHS. Finally, to cope with concurrent exceptions, [39] also suggests the introduction of future groups in order to gather the exception of a set of futures and apply a resolution function to them. But this solution requires the writing of a lot of code to explicitly deal with future groups. With SaGE, the support for exception resolution is directly integrated in the EHS. Provided that corresponding resolution functions are defined, concurrent exception management does not require any extra programming.

References

1. Briot, J.P., Guerraoui, R.: A classification of various approaches for object-based parallel and distributed programming. In Padget, J.A., ed.: Collaboration between Human and Artificial Societies. Number 1624 in Lecture Notes in Artificial Intelligence. Springer-Verlag (1999) 3–29 Invited conference.
2. Lavender, R.G., Schmidt, D.C.: Active object: An object behavioral pattern for concurrent programming. In Coplien, Vlissides, Kerth, eds.: Pattern Languages of Program Design. Addison-Wesley Reading (1996)
3. FIPA: Foundation For Intelligent Physical Agents : Request Interaction Protocol Specification. (2002)
4. Ferber, J.: Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence. Addison-Wesley Pub Co; 1st edition (February 25, 1999) (2005)
5. Clement Jonquet, S.A.C.: The strobe model: Dynamic service generation on the grid. Applied Artificial Intelligence Journal Special issue on Learning Grid Services **19**(9-10) (2005) 967–1013
6. Dedecker, J., Cutsem, T.V., Mostinckx, S., D'Hondt, T., Meuter, W.D.: Ambient-oriented programming in ambienttalk. In: Proceedings ECOOP'06 (European Conference on Object-Oriented Programming), Springer-Verlag (2006) To appear.
7. Goodenough, J.B.: Exception handling: Issues and a proposed notation. Commun. ACM **18**(12) (1975) 683–696
8. Levin, R.: Program structures for exceptinal condition handling. Phd dissertation, Dept. Comput. Sci., Carnegie-Mellon University Pittsburg (1977)
9. Liskov, B., Snyder, A.: Exception handling in CLU. IEEE Transactions on Software Engineering **5**(6) (1979) 546–558
10. Yemini, S., Berry, D.M.: A modular verifiable exception-handling mechanism. ACM Transactions on Programming Languages and Systems **7**(2) (1985) 214–243
11. Mitchell, J., Maybury, W., Sweet, R.: Mesa language manual, version 5.0. Csl-79-3, Xerox Palo Alto Research Centre (1979)
12. al, I..: Rationale for the design of the ada programming language. ACM SIGPLAN Notices **14**(6A) (1979) 1–139
13. K.Pitman: Error/condition handling. Technical report, Contribution to WG16. Revision 18.Propositions for ISO-LISP. AFNOR, ISO/IEC JTC1/SC 22/WG 16N15 (April 1988)
14. Pitman, K.M.: Condition handling in the lisp language family. In: Advances in Exception Handling Techniques. LNCS (Lecture Notes in Computer Science) 2022. Springer-Verlag (2001) 39–59
15. Meyer, B.: Disciplined exceptions. Technical report tr-ei-22/ex, Interactive Software Engineering, Goleta, CA (1988)

16. Dony, C.: Exception handling and object-oriented programming : towards a synthesis. ACM SIGPLAN Notices **25**(10) (1990) 322–330 *OOPSLA ECOOP '90 Proceedings*, N. Meyrowitz (editor).
17. Koenig, A., Stroustrup, B.: Exception handling for C++. In USENIX, ed.: C++ Conference Proceedings, April 9–11, 1990. San Francisco, CA, Berkeley, CA, USA, USENIX (1990) 149–176
18. Romanovsky, A.B., Kienzle, J.: Action-oriented exception handling in cooperative and competitive concurrent object-oriented systems. In: Advances in Exception Handling Techniques. (2000) 147–164
19. Dony, C., Knudsen, J.L., Romanovsky, A.B., Tripathi, A., eds.: Advances Topics in Exception Handling Techniques. Volume 4119 of Lecture Notes in Computer Science. Springer (2006)
20. Theriault, D.: A primer for the Act-1 language. Technical Report AI Memo 672, MIT Artificial Intelligence Laboratory (1982)
21. Halstead, R., Loaiza, J.: Exception handling in multilisp. In: 1985 Int'l. Conf. on Parallel Processing. (1985) 822–830
22. Campbell, R., Randell, B.: Error recovery in asynchronous systems. IEEE Transactions on Software Engineering (SE) **SE-12 number 8**(8) (1986) 811–826
23. Gärtner, F.C.: Fundamentals of fault tolerant distributed computing in asynchronous environments. ACMCS **31**(1) (1999) 1–26
24. Keen, A.W., Olsson, R.A.: Exception handling during asynchronous method invocation. In Monien, B., Feldmann, R., eds.: Proceedings of Euro-Par 2002 Parallel Processing. Lecture Notes in Computer Science. Springer-Verlag (2002) 656–660
25. Van Roy, P.: On the separation of concerns in distributed programming: Application to distribution structure and fault tolerance in Mozart. In: Fourth International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA 99), Tohoku University, Sendai, Japan, World Scientific (1999)
26. Campéas, A., Dony, C., Urtado, C., Vauttier, S.: Distributed exception handling : ideas, lessons and issues with recent exception handling systems. In: Proceedings of RISE'04 : First International Workshop on Rapid Integration of Software Engineering techniques, Luxembourg (2004) 82–92
27. Carlsson, R., Gustavsson, B., Nyblom, P.: Erlang: Exception handling revisited. In: Proceedings of the Third ACM SIGPLAN Erlang Workshop. (2004)
28. Iliasov, A., Romanovsky, A.: Exception handling in coordination-based mobile environments. In: Proceedings of 29th IEEE International Computer Software and Applications Conference (COMPSAC 2005), 25-28 July, Edinburgh, Scotland, UK. (2005) 341–350
29. Klein, M., Dellarocas, C.: Exception handling in agent systems. In Etzioni, O., Müller, J.P., Bradshaw, J.M., eds.: Proceedings of the Third Annual Conference on Autonomous Agents (AGENTS-99), New York, ACM Press (1999) 62–68
30. Klein, M., Dellarocas, C.: Towards a systematic repository of knowledge about managing multi-agent system exceptions, ases working paper ases-wp-2000-01 (2000)
31. Tripathi, A., Miller, R.: Exception handling in agent oriented systems. In: Advances in Exception Handling Techniques. LNCS (Lecture Notes in Computer Science) 2022. Springer-Verlag (2001) 128–146
32. Miller, R., Tripathi, A.: The guardian model and primitives for exception handling in distributed systems. IEEE Trans. Software Eng. **30**(12) (2004) 1008–1022
33. Souchon, F., Dony, C., Urtado, C., Vauttier, S.: Improving exception handling in multi-agent systems. Lecture Notes in Computer Science (2003)

34. Dony, C., Urtado, C., Vauttier, S.: Exception handling and asynchronous active objects : Issues and proposal. [19] chapter 5 81 – 101
35. Randell, B., Romanovsky, A., Stroud, R.J., Xu, J., Zorzo, A.F.: Coordinated Atomic Actions: from Concept to Implementation. Technical Report 595, Department of Computing Science, University of Newcastle upon Tyne (1997)
36. Miller, R., Tripathi, A.: Primitives and mechanisms of the guardian model for exception handling in distributed systems. In: Exception Handling in Object Oriented Systems: towards Emerging Application Areas and New Programming Paradigms Workshop (at ECOOP'03 international conference) proceedings. (2003)
37. Souchon, F., Dony, C., Urtado, C., Vauttier, S.: A proposition for exception handling in multi-agent systems. In: SELMAS'03 International Worskshop proceedings. (2003)
38. Souchon, F., Urtado, C., Vauttier, S., Dony, C.: Exception handling in component-based systems: a first study. In: Exception Handling in Object Oriented Systems: towards Emerging Application Areas and New Programming Paradigms Workshop (at ECOOP'03 international conference) proceedings. (2003) 84–91
39. Rintala, M.: Handling multiple concurrent exceptions in C++ using futures, kokoelmassa romanovsky. In: Developing Systems that Handle Exceptions, Proceedings of ECOOP 2005 Workshop on Exception Handling in Object Oriented Systems, ACM Press (2005)