Combining Tasking and Transactions

Jörg Kienzle
Software Engineering Laboratory
Swiss Federal Institute of Technology
CH–1015 Lausanne, Switzerland
Joerg.Kienzle@epfl.ch

Abstract

This position paper discusses the issues in design and development of a transaction support for Ada 95. Transactions and other fault tolerance mechanisms are reviewed, and their applicability in a concurrent programming language is analyzed. Possible ways of integration are presented, and implementation problems are discussed.

Keywords: *Transactions, Concurrency, Fault Tolerance, Object-Oriented Programming, Ada 95*

1 Introduction

The importance of fault tolerance in computing systems is constantly increasing. More and more software systems show a growing demand for increased reliability. Fault tolerance techniques for coping with the occurrence and effects of hardware failures form a vital part of any reliable computing system. Software fault tolerance — the ability of software to cope with faults, especially design faults — and fault-tolerant software — the ability of software to mask the failures of underlying hardware — are active fields of research [2].

Transactions are a mechanism that addresses both, software fault tolerance and fault-tolerant software, even for distributed systems. The basic, sequential transaction model has been used for many years to provide consistent access to databases. It is yet unclear how to use transaction mechanisms in the presence of cooperative concurrency offered by concurrent programming languages such as Ada 95.

Ada has a strong reputation for its error-detection qualities, such as strong typing, modularity, and separate compilation; it has been extensively used for the development of mission-critical and safety-critical software. It is not surprising that there has also been various research in the fault-tolerant area, such as providing recovery blocks for Ada [3], replication for Ada 83 [4], and recently even replication of Ada 95 partitions [5, 6]. Support for transactions in Ada 95 would be a powerful tool for a programmer of a fault-tolerant application.

This paper will study the issues in design and development of a transaction support for Ada 95. In section 2, transactions and other existing fault tolerance mechanisms are reviewed; their applicability to concurrent programming languages is presented in section 3. Possible ways of integrating transaction support into concurrent programming languages are discussed in section 4. Ada 95 specific implementation problems and ways of solving them are mentioned in section 5, and the last section draws some conclusions and presents future work.

2 Fault Tolerance Mechanisms

Transactions

The notion of *transaction* has first been introduced in database systems in order to correctly handle concurrent updates and to provide fault tolerance with respect to hardware failures. An overview of transactions and their properties can be found in [7]. Basically, a transaction groups an arbitrary number of simple actions together, making the whole appear indivisible as far as the application is concerned and with respect to other concurrent transactions. At any time during the execution of the transaction it can *abort*, which means that the state of the system will be restored to the state at the beginning of the transaction (also called *roll back*). Once a transaction has completed successfully (is *committed*), the effects become permanent. The properties of transactions are referred to as the ACID properties: *Atomicity*, *Consistency*, *Isolation* and *Durability.*

The basic transaction model, also called *flat transactions*, has been extended in order to provide more flexible support for concurrency and recovery. *Nested transactions* [8] allow transactions to start *subtransactions*, thus creating a tree of transactions. A subtransaction can either commit or roll back; its commit will not take effect (will not be visible to the outside world), though, unless the parent transaction commits. The rollback of a transaction anywhere in the tree causes all its subtransactions to roll back.

Recovery Blocks

Recovery blocks [9] have been introduced as a software structuring mechanism providing software fault tolerance based on the concept of design diversity and backward error recovery. A recovery block consists of one or more alternatives implementing functionality, coupled with an acceptance test that determines whether an alternative has functioned correctly or not. Under normal conditions, only the first alternative is executed. Only if an error is detected by the acceptance test, other alternatives are tried successively. The typical syntax for expressing recovery blocks is as follows:

```
ensure <acceptance test>
   by <primary alternative>
else by <second alternative>
...
else by <nth alternative>
else error
```

It is important to note that prior to the execution of the primary alternate, a snapshot of the current state of the component is taken. When an alternative fails to pass the acceptance test, this state is restored, and the next alternative is tried. This is repeated until either an alternative succeeds, or there are no more alternatives available. In that case, the execution of the component is considered a failure and an error is propagated.

Conversations

Whereas transactions can be used to handle competitive concurrency in distributed systems, the *conversation* concept has been introduced in [17] in order to deal with error recovery in systems that use cooperative concurrency. This is how it works:

Processes enter a conversation asynchronously; a recovery point is established in each of them. They freely exchange information within the conversation but cannot communicate with any outside process (violations of this rule are called *information smuggling*). When all participating processes have come to the end of the conversation, their acceptance tests are to be checked. If all tests have been satisfied, the processes leave the conversation. Otherwise, they restore their states from the recovery points and may try and execute a different *alternative*. The occurrence of an error in a process inside a conversation requires the rollback of all (and only) the processes in the conversation to the checkpoint established upon entering the conversation. Conversations may be nested freely, meaning that any subset of the processes involved in a conversation of nesting level i may enter a conversation of nesting level $i + 1$. A good general description of the conversation model can also be found in [10].

Atomic Actions

Atomic actions are essentially conversations with additional forward error recovery and exception resolution. This means that an exception that has been raised in a process that is part of an atomic action will be propagated to all other participating processes of that action. They have been mentioned for the first time in [11]. The essential AA characteristics and a way of achieving them in Ada 95 are summarized in [12]:

- Well-defined boundaries: each action has a start, and end, and side boundaries.
- Isolation: information exchange between participants and the outside world is strictly forbidden. Processes are not allowed to leave the atomic action until all processes are willing and able to leave.
- Nesting: AAs can be nested, but they may not overlap.
- Concurrency: Different AAs can be executed concurrently.
- Recovery: AAs form the basis of damage confinement in a system. Backward and forward error recovery should be possible.

An Ada 95 framework for atomic actions (programming guidelines and techniques) are presented in [13], and a complete discussion is given in [14].

Coordinated Atomic Actions

The term *Coordinated Atomic Action* (or *CA action*) has been introduced in [15]. A CA action characterizes an activity between a group of interacting objects that combines some properties of both conversations and transactions and integrates exception handling. They can also be seen as an extension to atomic actions. Basically, they allow processes that are involved in a CA action to access *external* objects. Updates to these objects have transactional semantics with respect to other CA actions. Nesting of CA actions is permitted.

3 Transactions in Concurrent Languages

When thinking of providing transaction support in a concurrent language like for instance Ada 95 or Java, the traditional, sequential transaction model is not flexible enough. The concurrent programming features of Ada 95 are very powerful and well integrated into the language, and hence application programmers make ubiquitous use of them. It would be very annoying to restrict the use of tasking inside a transaction.

Some existing transaction systems (Argus [16], Camelot [17]) allow nested transactions to execute concurrently, which can indeed improve performance. But nested transactions are isolated from each other, and thus only provide

competitive concurrency. To make cooperative concurrency possible, a different scheme must be used, namely a mixture between conversations and transactions.

CA actions are such a scheme. Processes participating in a CA action can cooperate and exchange data, and when they access external objects they do so with transactional semantics. CA actions are a relatively new concept and research is still in progress. [18, 19] show that they can also be used as design concepts to structure fault-tolerant distributed systems. According to [20], a CA action is defined by:

- A list of roles of the CA action and corresponding program code. Each role will be executed concurrently.
- A list of interface exceptions that the CA action can signal to an enclosing CA action.
- A list of external objects used by the CA action.
- A list of internal objects.
- A list of exception handlers, one for each role.
- An exception resolution graph.

When looking at CA actions from a concurrent programming language point of view, where the creation of tasks is possible at any time, the CA action concept might be too rigid, since the dynamic creation of roles for a CA action is forbidden. On the other hand, exception resolution between exceptions raised in different roles might not be necessary from a strict transaction point of view and therefore the overhead in design and implementation complexity can be saved.

Classification of Fault-Tolerant Systems

[21] partitions fault-tolerant systems incorporating backward error recovery into two broad classes. The authors describe an *object and action model*, which is based on the use of transactions controlling operations on objects, and the *process and conversation model*, which consists of concurrent processes communicating via messages. They claim that the two models are duals of each other, and present a mapping between the two models. Using that mapping, they identify techniques that have been developed within the context of one model, and finally turned out to have interesting and unexplored duals in the other model. Understanding this duality concepts will help to identify a transaction scheme that can be nicely combined with the concurrency features of concurrent programming languages.

4 Integration of Transactions

There are many ways to integrate transaction support into a programming language. The first approach would be to extend the programming language with new keywords, similar to [22]. Unfortunately, extending the language means modifying the compiler, which counteracts portability. Precompilers that translate a source file written in the extended language into code that consists only of pure language constructs can help to ease this problem, but then other problems arise when using tools that work with the source code and the object code, i.e. debuggers.

Another approach is to introduce new compiler directives, like in the approach described in [3]. The authors of that paper provide support for recovery blocks in Ada 95 using pragmas of the form:

```
pragma Recovery_Block (
   Acceptance_Test,
   Alternative_1,
   ...
   Alternative_N);
```

where `Acceptance_Test` and `Alternative_I` are the names of subprograms implementing the alternatives and their acceptance test, respectively. Here again, supporting new compiler directives results in compiler changes and hence is not portable. To get a portable scheme, only pure language constructs must be used.

From a programming language point of view, integration can be achieved in a very elegant way following an object-oriented approach. To what extent a concurrent transaction scheme can be encapsulated in objects is not easy to answer. A trade-off must be made between:

- Structure / Readability: Centralize all information of a transaction in one place.
- Enforcement of Transaction Rules: Guarantee the proper use of transactions, prohibit information smuggling.
- Reuse and Ease of Use: Keep the work that a user must do to create application transactions small.
- Hide implementation issues.
- Support for distribution: Cooperating tasks of a transaction can execute on different processing nodes.

[13] compares different approaches of programming atomic actions in Ada 95, one based on packages and the other on tasks. Extensibility of atomic actions through type extension is also investigated. Further research is needed to determine the best way of integrating transactions with cooperative concurrency into concurrent programming languages.

Taking a bottom-up approach, one can identify the need for backward error recovery in all of the transaction schemes presented in section 2. Backward error recovery can be achieved in an object-oriented manner by requiring the user of the transaction support to encapsulate the application state in one or several *recoverable objects*[1]. These

1. This is also the way the Object Transaction Service of CORBA [1] is structured. The objects providing state restoration operations are called *Resources*.

objects must provide state restoration operations that allow the transaction support to automatically save the state on transaction begin, and restore the state when a transaction aborts. To provide durability, these objects can be extended to write their state to stable storage (*persistent* recoverable objects).

5 Implementation Considerations

Regardless of how the different problems mentioned in the previous sections are dealt with, several implementation problems must be addressed when considering providing transactions in Ada 95. The following subsections will highlight some of these problems and reflect on possible solutions.

Backward Error Recovery

When using a transaction-like fault tolerance mechanism, it is often necessary to be able to undo the effects of a faulty computation, that is restoring the state of the application to a point where it was known to be correct. Objects offering such restoration capabilities in a concurrent environment are called *shared recoverable objects*. Such shared recoverable objects have been implemented already, and details about the implementation and how they must be used can be found in [23] and are summarized below:

Saving the state of a recoverable object is sometimes also referred to as *taking/establishing a checkpoint/snapshot*. Once saved, the state of the object can be restored at any time. It is possible to take several snapshots of an object, which means that it is possible to have multiple *versions* of the state of an object to which one can revert to.

Shared recoverable objects can be used by multiple tasks concurrently. Such tasks are called *participant tasks*. Note that a shared recoverable object does neither provide nor impose any concurrency control between tasks accessing it. Consistent access to the data encapsulated in the object must be ensured by the application programmer.

The implementation shown in [23] provides a tagged type `Recoverable_Object` that must be extended in order to obtain a recoverable object holding application data. Operations such as `New_State`, `Join_State`, `Commit_State` and `Discard_State` are provided.

When implementing recoverable objects, one has to choose between two different strategies of updating the state of the object: *in-place* update and *deferred* update. It is shown that deferred update — all versions of the object are visible to the user — should be used when objects are shared between multiple tasks. The versions are handled by reference, and therefore multiple tasks can have access to the same version at a time. The `New_State` operation allows a task to create a new version of the recoverable object. Other tasks that want to work with the new version must explicitly call `Join_State`. Each participating task must either commit its changes to a version of the object, or inform the object that it wants to discard the version, before a final decision can be taken. If all participants have committed their changes, the contents of the version are copied back to the previous version. If on the other hand one or several participants want to discard their changes, the complete contents of the version are discarded.

Deserter tasks, i.e. tasks that have been aborted due to explicit abort statements or Asynchronous Transfer of Control, or just terminated without committing or discarding the version, are automatically detected and their vote is ignored.

Transaction Contexts

Once a task has started a transaction, the transaction context should be tightly linked to the task. The context of a transaction in a concurrent environment would includes things like the number and identity of other participants, the external objects that have been manipulated, and, in the case of a distributed transaction system, a list of remote participants that are also part of the transaction. One possible way to achieve this link between a task and its transaction is to store a reference to the transaction descriptor in a task attribute.

Deserter Tasks

In Ada 95, tasks can be aborted by other tasks. This can potentially lead to "dangling" transactions, meaning transactions that have been started, but not properly committed or aborted. These situations should be avoided. Transactions that don't have any tasks working on their behalf anymore must therefore be aborted.

A possible solution to this deserter problem is using a controlled task attribute which contains a reference to the transaction descriptor. The `Finalize` procedure of the controlled type will be called automatically if the task is aborted. It is then possible to inform the transaction of the abortion of the task.

Creating New Tasks

If new tasks can be created during a transaction, they should automatically be included as participants of the transaction of the parent task. Using a controlled task attribute will not help in this case, since task attributes are not required to exist from the moment a task is created, but only when they are associated a value different than the default value. This implies that even if a task attribute is a controlled type, the `Initialize` procedure will not automatically be called upon every task creation.

6 Conclusion and Future Work

This paper has highlighted problems that arise when designing, integrating and implementing some form of transaction

support for concurrent programming languages, in particular Ada 95. Obviously, a transaction scheme that supports cooperative concurrency must be a mixture between conversations and transactions. Integration into the language should be done in a portable way, as object-oriented as possible.

A class providing shared recoverable objects has already been implemented. It provides operations for state restoration, thus making backward error recovery possible. Future work consists of extending this class to provide durability as required by the ACID properties of transactions. It should be possible to write the state of such a persistent recoverable object to stable storage when necessary, and restore the state in case of failures. Further plans are to implement a transaction support on top of these recoverable objects.

7 References

[1] Object Management Group, Inc.: *Object Transaction Service*, August 1994.

[2] Lee, P. A.; Anderson, T.: "Fault Tolerance - Principles and Practice". In *Dependable Computing and Fault-Tolerant Systems*, volume 3, Springer Verlag, 2nd ed., 1990.

[3] Kermarrec, Y.; Nana, L.; Pautet, L.: "Providing fault-tolerant services to distributed Ada 95 applications". In *TRI-Ada'96 conference*, pp. 39 – 47, ACM Press, December 1996.

[4] Miranda, J.; Alvarez, A.; Arévalo, S.; Guerra, F.: "Drago: An Ada Extension to Program Fault-Tolerant Distributed Applications". In *Reliable Software Technologies - Ada-Europe'96*, volume 1088 of *Lecutre Notes in Computer Science*, pp. 235 – 246, Springer Verlag, 1996.

[5] Wolf, T.: "Fault Tolerance in Distributed Ada 95". *Proceedings of the 8th International Real-Time Ada Workshop **XVII**(5)*, pp. 106 – 110, September 1997.

[6] de las Heras-Quirós, P.; González-Barahona, J. M.: "Programming Distributed Fault Tolerant Systems: The replicAda Approach". In *TRI-Ada'97 Conference*, pp. 21 – 29, St. Louis, MO, November 1997, ACM Press.

[7] Gray, J.; Reuter, A.: *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, California, 1993.

[8] Moss, J. E. B.: *Nested Transactions, An Approach to Reliable Computing*. Ph.D. Thesis, MIT, Cambridge, April 1981.

[9] Randell, B.: "System structure for software fault tolerance". *IEEE Transactions on Software Engineering **1**(2)*, pp. 220 – 232, 1975.

[10] Strigini, L.; Giandomenico, F. D.; Romanovsky, A.: "Coordinated backward recovery between client processes and data servers". *IEE Proceedings - Software Engineering **144**(2)*, pp. 134 – 146, April 1997.

[11] Jalote, P.; Campbell, R. H.: "Atomic Actions in Concurrent Systems". In *5th International Conference on Distributed Computing Systems*, pp. 184 – 191, Denver, 1985.

[12] Burns, A.; Wellings, A. J.: "Ada 95: An Effective Concurrent Programming Language". In *Reliable Software Technologies - Ada-Europe'96*, volume 1088 of *Lecture Notes in Computer Science*, pp. 58 – 77, Springer Verlag, 1996.

[13] Romanovsky, A.; Mitchell, S. E.; Wellings, A. J.: "On Programming Atomic Actions in Ada 95". In *Reliable Software Technologies - Ada-Europe '97*, volume 1251 of *Lecture Notes in Computer Science*, pp. 254 – 265, Springer Verlag, June 1997.

[14] Romanovsky, A.: "A Study of Atomic Action Schemes Intended for Standard Ada". *Technical Report 600*, University of Newcastle upon Tyne, June 1997.

[15] Xu, J.; Randell, B.; Romanovsky, A.; Rubira, C. M. F.; Stroud, R. J.; Wu, Z.: "Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery". In *FTCS-25: 25th International Symposium on Fault Tolerant Computing*, pp. 499 – 509, Pasadena, California, 1995.

[16] Liskov, B.: "Distributed Programming in Argus". *Communications of the ACM **31**(3)*, pp. 300 – 312, March 1988.

[17] Spector, A. Z.; Pausch, R. F.; Bruell, G.: "Camelot: A Flexible, Distributed Transaction Processing System". In *Proceedings of the 33rd IEEE Computer Society International Conference (Spring COMPCON 88)*, pp. 432 – 437, San Francisco CA, USA, March 1988, IEEE Computer Society Press.

[18] Zorzo, A. F.; Romanovsky, A.; Xu, J.; Stroud, R. J.; Welch, I. S.: "Using Coordinated Atomic Actions to Design Complex Safety-Critical Systems: The Production Cell Case Study". *Technical report*, Esprit Long Term Research Project DeVa - Second Year Report, December 1997.

[19] Zorzo, A. F.; Romanovsky, A.; Xu, J.; Stroud, R. J.; Welch, I. S.: "Using Coordinated Atomic Actions to Design Dependable Distributed Object Systems". *Technical report*, Esprit Long Term Research Project DeVa - Second Year Report, December 1997.

[20] Vachon, J.; Buchs, D.; Buffo, M.; Randell, B.; Romanovsky, S.; Xu, J.: "COALA - A Formal Language for Coordinated Atomic Actions". *Technical report*, Published in DeVa third year report, December 1998.

[21] Shrivastava, S. K.; Mancini, L. V.; Randell, B.: "The Duality of Fault-tolerant System Structures". *Software-Practice and Experience **23**(7)*, pp. 773 – 798, July 1993.

[22] Patino-Martinez, M.; Jimenez-Peris, R.; Arevalo, S.: "Integrating Groups and Transactions: A Fault-Tolerant Extension of Ada". In *Reliable Software Technologies - Ada-Europe'96*, volume 1411 of *Lecture Notes in Computer Science*, pp. 78 – 89, 1998.

[23] Kienzle, J.; Strohmeier, A.: "Shared Recoverable Objects". submitted to *Reliable Software Technologies - Ada-Europe'99*.