Implementing Transactions using Ada Exceptions: Which Features are Missing?*

M. Patiño-Martínez†, R. Jiménez-Peris†, S. Arévalo⋆

† Universidad Politécnica de Madrid
Facultad de Informática, Boadilla del Monte, 28660, Madrid, Spain
{mpatino,rjimenez}@fi.upm.es
⋆ Universidad Rey Juan Carlos
Escuela de Ciencias Experimentales, Móstoles, 28933, Madrid, Spain
s.arevalo@escet.urjc.es

Abstract

Transactional Drago programming language is an *Ada* extension that provides transaction processing capabilities. Exceptions have been integrated with transactions in *Transactional Drago*; exceptions are used to notify transaction aborts and any unhandled exception aborts a transaction. Transactions can be multithreaded in *Transactional Drago*, and therefore, concurrent exceptions can be raised. In that case a single exception must be chosen to notify the transaction abort. *Transactional Drago* provides exception resolution within transactions to tackle such a situation. In this paper we describe the transaction model of *Transactional Drago* focusing on how the *Ada* exception model can be used to implement *Transactional Drago* semantics. In the paper we identify which features of *Ada* were useful, as well as some weaknesses we have found in the language. We point out some missing basic mechanisms for the *Ada* standard that can be of great help for developing applications with strong exception handling requirements like transactional frameworks and many other applications.

Keywords: exceptions, concurrent exception resolution, transactions, Ada 95.

1 Introduction

Transactional Drago [PJA98] is an *Ada* extension that incorporates constructs for exception handling, transaction management and group communication. Behind the programming primitives, a sophisticated model unifying exceptions, transactions and group communication allows for concise reasoning and intuitive understanding of the behavior of the programs developed using such primitives. This makes it the ideal vehicle for developing large fault-tolerant distributed software. Programming languages incorporate primitives to deal with exceptions while transactions and process groups usually require dedicated middleware like transaction processing (TP) monitors and group communication systems. As a result, combining exception handling, transactions and process groups is often quite complex. *Transactional Drago* was developed to address this problem.

Nowadays, transactions are becoming a common mechanism in general purpose programming languages (Enterprise Java Beans [Mic00]) and systems (CORBA [OMG95b]). Transaction importance has grown with the advent of new distributed applications (e.g. e-commerce) and they are not just confined to databases. In this paper we describe the transaction processing facilities provided by *Transactional Drago* and how exceptions have been integrated in this context.

Transactional Drago is translated into *Ada* and *TransLib* [JPAB00, KJRP01], an object oriented framework to program distributed transactional systems in *Ada*. Another contribution of this paper is how the *Ada* exception mechanism can be used for the implementation of transactions. We have found several missing features in *Ada* that would

*This work has been partially funded by the Spanish Research Council (*CICYT*), contract number *TIC98-1032-C03-01* and the Madrid Regional Research Council (*CAM*), contract number *CAM-07T/0012/1998*.

have been useful for implementing transactions. In this paper we identify such features and propose some basic mechanisms for *Ada* that would be helpful for any application with a complex exception handling semantics like transactional systems.

The paper is organized as follows. Section 2 presents related work. Transactions and their implementation are presented in Section 3. Section 4 deals with multithreaded transactions. Section 5 discusses how exception resolution for concurrent exceptions is performed in *Transactional Drago* and its implementation. Finally, we present our conclusions in Section 6.

2 Related Work

One of the earliest papers on the integration of exceptions with backward recovery is [JC86]. In this paper backward recovery provided by conversations (distributed recovery blocks) is integrated with forward recovery provided by exceptions. However, the goals of conversations (to provide software fault-tolerance) and transactions (data consistency) are different.

More related to our work is Argus [Lis88], a programming language that provides nested transactions, as well as exceptions. In Argus it is possible to commit or abort a transaction, both normally or exceptionally. It is possible to propagate an unhandled exception outside of a transaction, without aborting it [ea95], which can be misleading.

Arjuna [SDP91] is an object oriented system written in C++ that provides nested atomic transactions. Although C++ provides exceptions, no semantics is given for exceptions crossing transaction boundaries.

Transactional-C [Tra95], the programming language provided by the Encina TP-monitor, provides multithreaded transactions. Any exception going out of a transaction aborts the transaction. Transaction threads can raise concurrent exceptions, but no exception resolution mechanism is provided.

Transactions have also been incorporated in functional programming languages [HKG$^+$94]. In this extension of SML, transactions can be multithreaded and exceptions are used to propagate aborts. However, the semantics for transactions with concurrent exceptions is not defined.

Some initial work exists on how to implement basic atomic actions with *Ada* [WB97, RMW97], but without addressing exceptions concurrently raised nor recovery. The activity of a group of components constitutes an atomic action if there are no interactions between that group and the rest of the system [AL81]. In an atomic action, participants are processes that asynchronously join the action to collaborate. Atomic actions were initially proposed for software fault-tolerance. Therefore, the consistency of data is not guaranteed in case of node crashes.

Coordinated atomic actions [XRea95] provide a more general setting. They integrate conversations, transactions and exceptions to tolerate hardware and software design faults. Coordinated atomic actions finish producing a normal outcome, an exception outcome, an abort exception, or a failure exception [XRea99]. An exception in a coordinated action is always propagated to all participants and in case of concurrent exceptions a graph is used for resolution [XRR98].

In the open multithreaded transaction model [KJRP01, Kie00], independent threads decide to join a transaction. An exception propagated out of a transaction also aborts the transaction. If several threads raise an exception, each one propagates the raised exception to the enclosing scope. No exception resolution is contemplated as threads are independent.

The Object Transaction Service (OTS) [OMG95a] of CORBA [OMG95b] provides support for transactions. Support for nested transactions is not mandatory. There is no integration among exceptions and transactions, and therefore, no semantics is given for exceptions crossing a transaction boundary.

[Iss91] proposes an exception handling model for parallel programming. An operation can be requested to a group of objects. The group runs this operation in parallel. If the group objects raise several exceptions, exception resolution is performed and the resulting exception is propagated to the caller. No transactional properties are provided.

Exceptions have been integrated with transactions in Enterprise Java Beans (EJB) [Mic00]. The EJB model of transactions is flat and there is no concurrency inside a transaction. EJB uses the Java exception model, where exceptions belong to a class. Exceptions are classified in system and application exceptions. Each type of exception belongs to a different class. Application exceptions can be handled within a transaction and therefore, there is no need for rollback. However, when an exception is raised in a client as a result of a bean method invocation, the client does not know if the transaction is going to abort or not. A method is provided to query about the transaction status. System exceptions cannot be handled and they abort the transaction.

3 Transactions

Transactions provide data consistency in the presence of concurrent activities and system crashes [GR93]. A transaction either finishes successfully (commits) and its effects become durable (they are not lost even in the advent of crashes), or it fails (aborts) and its effects are undone. Transaction ACID (*A*tomicity, *C*onsistency, *I*solation and *D*urability) properties [HR83] are useful for constructing reliable distributed applications.

Transactions can be nested [Mos85]. This is useful for two reasons: first, they allow additional concurrency within a transaction by running concurrently nested transactions (subtransactions). Second, the abort of a subtransaction does not force the parent transaction to abort, providing in this manner damage confinement against failures (especially in a distributed system where partial failures can be more likely). Subtransaction commit is conditioned to the commit of its ancestors. If a transaction aborts, all its descendant subtransactions will abort.

Transactions in *Transactional Drago* are defined using the *transactional block* statement. The structure of a transactional block is similar to the one of an *Ada* block statement. Transactional blocks introduce a new scope; they have a declarative section, a body and can also have exception handlers.

```
transactional-block::=
  transaction
     [declare declarative-part]
     begin
        sequence-of-statements
     [exception
        exception-handlers]
  end transaction;
```

Nesting of transactional blocks is allowed yielding to nested transactions. All data declared inside a transaction are subject to concurrency control (in particular, locks) and are recoverable. *Transactional Drago* provides read/write locking. That is, two transactions accessing the same data item conflict if at least one of them performs a write operation. Locks are implicit, i.e., they are automatically set on each data item when it is accessed. This frees the programmer from the error prone task of having to acquire and release locks. When there is a failure, any changes introduced by uncommitted transactions are removed by the recovery procedure.

All data declared in a transactional block are volatile. Non-volatile (persistent) data are declared in the outermost scope of a program. Persistent data is prefixed with the `persistent` keyword in its declaration. For instance, if an array of elements of type `telement` is persistent, it will be declared as:

```
vector : persistent array (1..MaxElements)
         of telement;
```

3.1 Aborts and Exceptions

Exceptions are used to signal abnormal situations. In *Transactional Drago* any exception that is propagated outside the scope of a transaction causes the abort of the transaction [PJA01]. If the transaction had been able to handle the exception internally, it would mean that forward recovery (exception handling) was successfully applied within the transaction. However, if the error could not be handled or was not anticipated (there is no exception handler for it), backward recovery (undoing the transaction) is automatically performed. In this way, transactions act as firewalls confining damage produced by unhandled errors and exceptions are used as notification mechanism. If the transaction commits, it will not raise any exception.

Moreover, transaction aborts have been integrated with the exception mechanism. There is no `abort` statement in *Transactional Drago*. Aborts are always notified by means of an exception. Since programmers can define their own exceptions, exceptions provide more information about the cause of an abort than a simple abort statement.

The following example illustrates a transaction to withdraw money from an ATM. The transaction connects to a server (`Branch_Server`) where the accounts are held. If the balance on the account is less than the money requested, the transaction will abort and the `not_enough_funds` exception will be raised. The server could not be available during the `GetAccountBalance` service invocation. If this happens, the `branch_server_not_available` exception will be raised. The transaction provides a handler for that exception. Thus, instead of aborting the transaction, forward recovery is performed. In this situation the ATM gives a maximum fixed amount of money (`fixed_amount`). If the amount requested is less than this maximum, the ATM gives the money and the transaction commits. Otherwise, the transaction is aborted and the `limited_funds_availability` exception is raised. Thus, the exception reports that there is a limit on the funds that can be requested and that this limit has been exceeded.

```
transaction
begin
    GetAccountBalance(branch_server, account, amount);
    if requested_amount > amount then
       raise not_enough_funds;
    else
       amount:= amount - requested_amount;
    end if;
exception
   when branch_server_not_available =>
       if requested_amount > fixed_amount then
            -- Report to the user that the branch
            -- server is unavailable and the max.
            -- available amount is fixed_amount
            raise limited_funds_availability;
       else
            Credit(account, requested_amount);
       end if;
end transaction;
```

3.2 Implementing Transactions

Transactional blocks are translated into *Ada* block statements and *TransLib* invocations. However, *TransLib* can be used as a library for those programmers not willing to use an *Ada* extension. This means that the way a transaction is implemented must be such that is easy to write manually or automatically (using a preprocessor).

The events of interest for the runtime system concerning a transaction are its beginning, completition and outcome. Transactional systems use transaction identifiers (tids) to keep track of the transactions. Depending on whether the tid handling is performed transparently or not, there are several approaches to implement transaction bracketing, explicitly or implicitly. From the four possible combinations there is one we do not consider due to its lack of interest that is implicit bracketing and explicit tid handling. The following sections discuss the different approaches.

3.2.1 Explicit Bracketing and Tid Handling

Explicit bracketing is performed by invoking operations to begin and end (commit or abort) a transaction. Explicit bracketing implies to bracket the transaction with begin and end operations.

An *Ada* block is used to represent the transactional scope. Since the programmer can perform its own exception handling, an additional nested block is needed in the implementation of a transactional block to prevent interference between the programmer exception handling, and the automatic exception handling in charge of aborting the transaction in case of an unhandled exception propagation. The inner block corresponds to the programmer code including exception handling. The outer block performs transaction bracketing invoking *TransLib* operations (`T_Begin`, `T_End` and `T_Abort`) and default exception handling:

```
declare
   tid : Trans_Id;
begin
   tid:= T_Begin;
   declare
      -- transaction local declarations
   begin
      -- transaction code
   exception
      -- programmer defined exception handling
   end;
   T_End(tid);
exception
   when e: others =>
      T_Abort(tid, e);
end;
```

The `T_Begin` operation starts a new transaction and associates a tid to it. This tid is then used to notify the underlying system about which transaction is committing or aborting. If an exception is not handled in the transaction or is raised in the transaction exception handlers or by the `T_End` operation, the transaction must be aborted. That exception is handled by the `others` handler in the outer block. That handler calls the `T_Abort` operation with the corresponding exception occurrence as a parameter. `T_Abort` aborts the current transaction and propagates the exception to the enclosing scope. As it is shown an exception handler is used catch any unhandled exception and call the `T_Abort` operation. An exception occurrence is passed on to the `T_Abort` operation, in order to store it and possibly reraise it later.

Instead of using a parameter in the `T_Abort` operation, the exception could be raised immediately after calling T_Abort. However, if the transaction has several threads, this implementation would not be correct. Section 4 explains this situation.

It might be argued that the outer block statement is not needed and just use a single block statement. Using the following scheme:

```
declare
   tid : Trans_Id;
   -- transaction local declarations
begin
   tid=T_Begin;
   -- transaction code
exception
   when excep1 =>
      exception handler1
      T_End(tid);
   when excep2  =>
      exception handler2
      T_End(tid);
   when e: others=>
      T_Abort(tid, e);
end;
```

There are two reasons for having the outer block statement. First, the transaction might have declared data and if an exception is raised during the elaboration of the declarations, the transaction must abort, and the actions already performed must be undone. However, if the previous code is used, an exception raised during the elaboration of the transaction declarations will be propagated to the enclosing scope without executing the T_Abort operation. Although, the data declared in the transaction is volatile, some of the actions performed during the elaboration of these data could modify some external data (in an outer scope, persistent or not). What it is more, using the above code, the transaction would be unknown at elaboration time (the T_Begin operation has not been executed) and therefore, no undo is possible. This situation cannot be distinguished in the enclosing scope from the one in which the transaction has been executed and properly aborted (the T_Abort operation has been executed).

The same problem arises if the T_Begin invocation is out of the block statement and there is just a block statement, despite the transaction is known when the transaction declarations are elaborated. An exception in the declarations elaboration, will be propagated to the enclosing scope, which could not be transactional (the transaction is not nested) and therefore, the transaction cannot be aborted. If the transaction is nested and the exception is handled in the others handler of the enclosing transaction, two transactions must abort, the inner one and the outer one (if there is no programmer-defined handler for that exception). However, the T_Abort operation just aborts the inner transaction and the outer one is left inconsistent (it is neither committed nor aborted).

The second reason for having two block statements is the presence of programmer-defined exception handlers in the transaction. If an exception is implicitly raised in one of the handlers, the exception will be propagated to the enclosing scope, which fulfills the *Transactional Drago* semantics, but the corresponding T_Abort operation is not executed. Therefore, the situations previously described arise again.

Therefore, given the structure of a transactional block, the actions needed to commit and abort a transaction, and how exceptions are propagated in *Ada*, two block statements are needed to translate a transactional block. What it is more, the use of two block statements ensures that the handlers provided by the programmer will not collide with the one needed to perform the transaction abort.

3.2.2 Explicit Bracketing and Implicit Tid Handling

Tid handling is an error-prone task, for instance, when there are nested transactions. There is a very interesting feature in the *Ada* Systems Programming Annex, that allows to associate state to tasks. This feature enables implicit tid handling in the following way. The state of tasks is extended to keep a stack of active transactions (initially empty). In this way the operations to begin and end transactions just access this state to push the tid of a new transaction and to pop it, respectively. The transactional runtime system always knows which is the current transaction by just looking at the top of the stack. The resulting code for transaction bracketing is:

```
begin
   T_Begin;
   declare
      -- transaction local declarations
   begin
      -- transaction code
   exception
      -- programmer-defined exception handlers
   end;
   T_End;
exception
   when e: others =>
```

```
        T_Abort(e);
end;
```

The T_Begin operation starts a new transaction and associates a transaction identifier (*tid*) to the task that executes that operation. The new transaction can be either a top level transaction (if the task was not executing any transaction, i.e, if its stack was empty), or a subtransaction of the transaction currently executing. The T_End and T_Abort operations commit and abort the current transaction (i.e., the transaction on top of the stack), respectively. With this scheme, it has been possible to hide the tid handling from the programmer, preventing thus the errors to which is subject the explicit handling.

3.2.3 Implicit Bracketing and Tid Handling

Explicit bracketing is also subject to errors, since it is possible to forget writing any of the begin/end operations. The use of controlled types guarantees that bracketing is performed implicitly at the initialization and finalization of the corresponding scope. A transaction can be a controlled type, and a transactional scope just needs to declare a variable of this type. However, as the following piece of code shows, fully implicit bracketing is not possible with *Ada* controlled types:

```
declare
   t : Transaction;
begin
   declare
      -- transaction local declarations
   begin
      -- transaction code
   exception
      -- programmer-defined exception handlers
   end;
exception
   when e: others =>
      T_Abort(e);
end;
```

The transaction is a controlled type declared in the outer block whose Initialize procedure calls T_Begin. The Finalize checks if the transaction was aborted, and if not, it calls the T_End operation to commit. Therefore, there are no T_Begin and T_End calls in the translation. However, as the Finalize operation does not provide any information about how the scope has terminated nor about the exception that caused the termination (in case of exceptional termination), it prevents fully automatic bracketing, and forces an explicit invocation of the abort operation.

The *Ada* finalization operation of controlled types could be overloaded with an exceptional finalization operation that would have as an argument the exception occurrence that caused the finalization of the scope. An additional advantage of the exceptional Finalize is that it would save one of the two blocks needed to bracket a transaction. The automatic transaction abort needed in case of the propagation of an unhandled exception can be performed by the exceptional Finalize. The code that could be written with an exceptional Finalize would be:

```
declare
   t : Transaction;
   -- transaction local declarations
begin
    -- transaction code
exception
    -- programmer-defined exception handlers
end;
```

The exceptional Finalize with the exception occurrence as argument, besides being useful for building a transactional system, it would also be useful for automatic logging of exceptions, debugging, or administration purposes. Currently, this has to be performed manually writing ad-hoc exception handlers to keep track of the exception propagation. There are many frameworks where the actions to be performed during Finalize depend on whether the finalization is normal or exceptional. In this kind of applications fully automatic bracketing with controlled types is not possible without an exceptional Finalize.

4 Multithreaded Transactions

Tasks can be declared inside a transaction in *Transactional Drago*, hence a transaction can have auxiliary threads. *Multithreaded transactions* [PJA00] allow taking advantage of multiprocessor and multiprogramming capabilities, for instance a thread can perform input, another output, and a third one work on the data. A transaction finishes its execution when it reaches the end of the transactional block statement and all its threads have finished. This model is consistent with the *Ada* one, where blocks synchronize their termination with the tasks declared (allocated) in their scope.

Since locking is only intended for inter-transactional concurrency, that is, to guarantee the isolation of concurrent transactions (logical consistency), latches [MHL+98] are used to provide data consistency (physical consistency) in the presence of concurrent accesses from the same transaction (intra-transactional concurrency). As locks, latches are also automatically set, which simplifies the programming task.

Any of the threads of a multithreaded transaction can start a new transaction. In that case a subtransaction and its parent transaction will run concurrently, and therefore they can compete for the data. For this reason, concurrency control is used for all data used within a transaction, even if they are volatile. This is known as parent-child concurrency [HR93]. The locking algorithm used in *Transactional Drago* guarantees that subtransactions started by a transaction thread are atomic and isolated from all other threads of the parent transaction. Once a subtransaction acquires a lock on a data item, its parent transaction (or any ancestor) cannot access that data item until the subtransaction finishes. Latches also ensure that a lock on a data item is not granted to a subtransaction while its parent (or any ancestor) is accessing that data item.

4.1 Implementing Multithreaded Transactions

Transaction threads are modeled by *Ada* tasks. These tasks are declared within the declarative section of a transactional block or within an inner scope. Transaction threads are written with a different scheme than the one used for single threaded transactions, despite both are basically blocks. This difference stems from the fact that the underlying transactional system needs to associate tasks to their corresponding transactions, but this cannot be performed transparently. Despite being possible to identify *Ada* tasks (using the facility of the Programming Systems Annex), it is not possible to get the identifier of the master task (the task who created task under consideration). This lack in the Programming Systems Annex prevents automatic tid handling. Transactional tasks must be explicitly informed about the master task identifier or the tid of the transaction they belong to. In what follows the approach currently taken to implement multithreaded transactions, and how the code could be simplified if the facility to get the master task were available.

4.1.1 Explicit Bracketing and Tid Handling

In this approach the task body and its declarations are enclosed in a block statement. The block is enclosed between the `T_Begin_Thread` and `T_End` operations. For instance, if a transaction contains two thread types (e.g., ThreadType1 and ThreadType2), it can be written as follows:

```
declare
   tid : Trans_Id;
begin
   tid = T_Begin;
   declare
      -- transaction local declarations

   task  type ThreadType1
        (my_tid : TransactionIdentifier) is
      -- task interface
   end ThreadType1;

   task body ThreadType1 is
   begin
      T_Begin_Thread(my_tid);
      declare
         -- thread declarations
      begin
         -- thread code
      exception
         -- thread exception handlers
      end;
      T_End(my_tid);
   exception
      when e: others =>
         T_Abort(my_tid, e);
   end ThreadType1;

   task type ThreadType2
         (my_tid : TransactionIdentifier) is
      -- task interface
   end ThreadType2;

   task body ThreadType2 is
   begin
      T_Begin_Thread(my_tid);
      declare
         -- thread declarations
      begin
         -- thread code
      exception
         -- thread exception handlers
      end;
      T_End(my_tid);
   exception
      when e: others =>
          T_Abort(my_tid, e);
   end ThreadType2;

      Thread1 : ThreadType1(CurrentTID);
      Thread2 : ThreadType2(CurrentTID);
  begin
```

```
      -- transaction code
   exception
      -- transaction exception handlers
   end;
   T_End(tid);
exception
   when e: others =>
      T_Abort(tid, e);
end;
```

The inner block statement is used to ensure that any action performed by a thread is done on behalf of a transaction. If the block statement is not used and the `T_Begin_Thread` operation is at the beginning of the task body, the actions performed during the declarations elaboration are not associated to a transaction and therefore, they cannot be undone if an exception is raised and therefore, the transaction aborts.

Since it is not possible to know the master task (i.e., the creator task) of a task in *Ada*, the tid of the transaction is passed as a task discriminant (`my_tid` in the above example) in the task declaration. The `T_Begin_Thread` operation has the tid as parameter. This allows to associate the work performed by the thread to a transaction. The runtime system will apply the concurrency control policy previously described and keep track of all the modifications performed to be undone in case of an abort.

An exception raised during the elaboration of a thread declarations will finish the block statement and will be handled in the `others` handler of the task. The handler will call the `T_Abort` operation. Since variables declared within a thread are not implemented as variables declared at the outer task scope, but at an inner scope, the *Tasking_Error* exception cannot be raised due to an exception in the declarative part of a task body.

An unhandled exception in the thread body or an exception raised in a thread handler will also be handled by the task `others` handler. Since exceptions are not propagated out of a task in *Ada*, if any task executes the `T_Abort` operation, *TransLib* stores the exception in the transaction state. Otherwise, the exception would be missed. The main task (the one executing the transactional block) waits for the transaction threads completion in the `T_End` operation. The main task knows how many threads there are because they call the `T_Begin_Thread` operation, which increments the number of threads in the transaction state (accessible through the tid). If a thread aborts, the `T_End` operation in the main task will raise the exception that the thread raised. If the main task raises an exception and it is not handled, the `others` handler will handle it. The `T_Abort` operation will wait for the completion of the rest of the threads to undo transaction updates and reraise the exception.

4.1.2 Implicit Bracketing and Tid Handling

If an enhanced *Ada* with exceptional `Finalize` and an additional facility in the Programming Systems Annex to get the identifier of the master task, the code for multithreaded transactions could be simplified to the following:

```
declare
   trans : Transaction;
begin
   declare
      -- transaction local declarations

      task  type ThreadType1 is
         -- task interface
      end ThreadType1;

      task body ThreadType1 is
         trans : Transaction;
         -- thread declarations
      begin
         -- thread code
      exception
         -- thread exception handlers
      end ThreadType1;

      task type ThreadType2 is
         -- task interface
      end ThreadType2;

      task body ThreadType2 is
         trans : Transaction;
         -- thread declarations
      begin
         -- thread code
      exception
         -- thread exception handlers
      end ThreadType2;

   Thread1 : ThreadType1;
   Thread2 : ThreadType2;
begin
   -- transaction code
exception
   -- transaction exception handlers
end;
```

It can be seen how the code could be greatly simplified with the pointed out missing features in *Ada*. As before, an exceptional `Finalize` would enable implicit bracketing of the outer transaction. On the other hand, the facility to get the master task would allow to perform implicit tid handling within the transactional threads, as the initialization in the controlled type could find out the identifier of the master task, and thus access its transactional state to associate the new thread to its inner transaction.

5 Concurrent Exceptions

When a transaction aborts in *Transactional Drago*, a single exception is propagated to inform about the cause of the abort. If more than one thread of a transaction finishes with an unhandled exception, exception resolution is performed in *Transactional Drago* in order to propagate a single exception. Exception resolution [CR86] allows to choose an exception that represents all the exceptions that have been concurrently raised. *Transactional Drago* provides a default resolution scheme that is applied when concurrent exceptions are raised (in contrast with *Ada*, where no exception resolution scheme is provided and exceptions resulting in task termination are just lost). This default resolution scheme propagates the *several_exceptions* predefined exception.

Exception resolution is usually performed using an exception tree. When two concurrent exceptions are raised, the oldest common ancestor in that tree is the result of the resolution. In *Transactional Drago* programmers can define their own exception resolution scheme providing an exception resolution function. This function is defined in the declaration section of a transactional block by means of the following attribute-definition clause:

```
exception-resolution-clause::=
   for localresolution use name;
```

We have used an attribute-definition clause because the programmer exception resolution function overrides the by default resolution function of *Transactional Drago*. This behavior is similar to the one of attribute-definition clauses used for *Ada* streams.

name is the identifier of the programmer provided exception resolution function. A resolution function receives two exception identities as arguments and returns an exception identity (the result of the resolution of the two exceptions). That function implements the exception resolution tree and is called by the runtime system $n-1$ times (being n the number of concurrent unhandled exceptions raised in the transaction) to obtain a single exception. The following piece of code shows how the exception resolution function should be programmed.

```
transaction
declare
function WithdrawResolution
 (exception1, exception2: Exception_ID) re-
turn Exception_ID is
begin
   ...
   if exception1 = unknown_account'Identity then
      return exception1;
   ...
   end if;
end WithdrawResolution;

for localresolution use WithdrawResolution;
begin
   -- Perform concurrent work
end transaction;
```

Exceptions cannot be arguments of a subprogram in *Ada*, but it is possible to obtain the *identity* attribute of an exception, which can be used for this purpose. For instance, to pass the `not_enough_funds` exception as an argument, the real argument will be `not_enough_funds'identity`. The resolution function would be simpler, if exceptions were a type, removing the hassle of getting the identities of exceptions.

5.1 Exception Hierarchy

Exceptions are classified in *Transactional Drago* according to their origin as *system* and *application exceptions*. The system can implicitly abort a transaction (without an application request) in several circumstances such as lack of resources to commit the transaction (e.g., log space), or deadlocks. If the system aborts a transaction, the transaction will be undone and an exception will be propagated to the enclosing transaction (therefore, handlers for that exception in the transaction cannot capture those exceptions). Inòsome cases it is not worth to retry the transaction until a certain period of time has elapsed, since those situations can persist some time. In case of a deadlock, it makes sense to retry the transaction immediately with the hope that the conflicting transaction has already finished. The `abort_error` and `abort_retry_error` predefined system exceptions are used to distinguish these two situations.

On the other hand, application exceptions (predefined or user-defined) can be raised when an operation precondition does not hold or an error is found in the code. In the former case, the enclosing scope can be interested in knowing the reason of the error

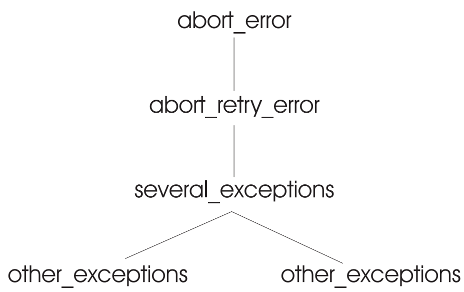

Figure 1: Exception resolution tree

and, perhaps, in handling it and retrying the transaction with different data (e.g., in a withdraw operation if the `not_enough_funds` exception has been raised, the client can try to withdraw less money or money from a different account). Whether it is meaningful to retry or not after an application exception is application dependent. In the case of an error in the code, the transaction enclosing scope will be interested in knowing that the transaction has been aborted, and that there are few possibilities of success if it is retried. For instance, if a `constraint_error` is raised due to an error in the code, it may not be meaningful to retry the transaction. The exception can be renamed to make it more understandable to the enclosing scope.

The nature of concurrent exceptions can be very different; for instance, the system can raise an exception indicating a deadlock and the transaction itself can raise an application exception (there are not enough funds to withdraw). A consistent resolution must be applied to raise a single exception that captures the maximum amount of information.

Transactional Drago has a predefined exception tree for exception resolution (Fig. 1). At the root of the hierarchy is the `abort_error` exception. Whenever this exception is raised, that one will be propagated. Unhandled application exceptions are only propagated when no system exception has been raised during the execution of a transaction. If only an application exception is raised, that one will be propagated notifying the transaction abort. However, if concurrent application exceptions are raised, by default, the predefined exception `several_exceptions` is raised. The programmer can override this default behavior, providing an exception resolution function. Recall that this resolution is only applied for application exceptions. If `abort_error` or `abort_retry_error` are raised concurrently with application exceptions, the system exception will be chosen as the final exception. Exception resolution takes place during transaction termination.

5.2 Implementation of Transaction Termination

Traditional transactions are single-threaded. In this situation transaction termination can be trivially determined. However, this is not the case for multithreaded transactions. When a thread (including the main task) of a multithreaded transaction completes, it is not yet known whether the transaction has terminated or not, neither how it has terminated.

The termination of multithreaded transactions requires a termination protocol that synchronizes the termination of all the participating threads. The main thread (the one that initiates the transaction) will coordinate the transaction termination. When the main thread completes, it must synchronize with the completion of the rest of the threads (a similar problem is dealt in *Ada* [Ada95] tasks). All the threads of a transaction must report to the main thread how they have terminated. Even if the main thread decides to commit, if any thread terminates raising an exception then the transaction will abort.

The protocol is started either in the `T_End` or `T_Abort` operation of the main thread, depending on whether the main thread finishes successfully or raising an exception. The first step in the termination protocol is waiting for the completion of all the threads. When a thread completes its execution, it records its outcome in the transaction state. The transaction state is accessible to all the transaction threads through the tid. If all the threads complete successfully, then the main task commits the transaction; otherwise, if one or more threads terminate exceptionally, it aborts the transaction. If several threads raise exceptions, resolution is applied in order to raise a single exception as the outcome of the transaction. It may seem that exception resolution is a time consuming task that requires extra synchronization among the transaction threads. However, the termination protocol is a synchronization point among all the transaction threads needed to decide the outcome of a transaction. If exception resolution is performed at this point no extra overheads are paid.

Since threads of a transaction do not survive the transaction scope (they finish with the transaction), there is no need to inform them about the transaction outcome. All the activities needed to abort or commit the transaction are performed in the next step of the protocol. Even if all the threads finish successfully, the transaction could abort. It could be the case that when the data changes are propagated to disk, the disk is full and the transaction cannot commit. In such a situation the termination protocol will abort the transaction and raise the `abort_error` exception. For this reason exceptions are raised at this final stage.

6 Conclusions

Transactions are becoming a general programming mechanism available in most systems, like Corba and Enterprise Java Beans. The integration of transactions in a programming language needs to be orthogonal to other language mechanisms and precisely defined.

Transactional Drago is an *Ada* extension that integrates exceptions with transactions. In this paper we have described the semantics of exceptions in that context and how exceptions can be used to implement that semantics. To summarize, exceptions are used to notify transaction aborts in *Transactional Drago* and also to implement aborts through the translation of transactions into block statements, which always have an exception handler that is in charge of the abort. Additionally, in multithreaded transactions concurrent exceptions are resolved, and exceptions raised at a thread are not lost as it happens in *Ada* with unhandled exceptions in tasks.

We have found some missing features in the *Ada* exception model that were needed for providing full implicit bracketing and tid handling.

- Finalize cannot distinguish between an exceptional and a normal finalization. This forces to include an implicit termination for the exceptional case.

- It is not possible to find out the master of a task (the task who created a particular task). This forces to pass in a task discriminant an identifier relating the child task to its master.

Two basic mechanisms have been proposed to overcome these lacks:

- An additional Finalize can be provided for controlled types that would be invoked in case of exceptional finalization. This Finalize would have as argument the exception occurrence that triggered the finalization.

- The Programming Systems Annex can be enriched with an additional function that given the identifier of a task returns the identifier of the master task.

An additional feature would also be helpful. Exceptions are not currently types in *Ada*. This implies that exception identities should be extracted from them or from exception occurrences in order to compare exceptions during exception resolution. A cleaner code could result if exceptions were types.

References

[Ada95] *Ada 95 Reference Manual, ISO/8652-1995*. Intermetrics, 1995.

[AL81] T. Anderson and P. A. Lee. *Fault Tolerance: Principles and Practice*. Prentice Hall, NJ, 1981.

[CR86] R. H. Campbell and B. Randell. Error Recovery in Asynchronous Systems. *IEEE TSE*, 12(8):811–826, August 1986.

[ea95] B. Liskov et al. *Argus Reference Manual*. MIT, CS Lab, 1995.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[HKG+94] N. Haines, D. Kindred, J. Gregory Morrisett, S. m. Nettles, and J. M. Wing. Composing First-Class Transactions. *ACM TOPLAS*, 16(6):1719–1736, 1994.

[HR83] T. Haerder and A. Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computer Surveys*, 15(4):287–317, 1983.

[HR93] T. Haerder and K. Rothermel. Concurrency Control Issues in Nested Transactions. *Very Large Databases Journal*, 2(1):39–74, 1993.

[Iss91] V. Issarny. An Exception Handling Model for Parallel Programming and its Verification. In *Proc. of ACM Conf. on Software for Critical Systems*, pages 92–100, 1991.

[JC86] P. Jalote and R. H. Campbell. Atomic Actions for Fault-Tolerance Using CSP. *IEEE TSE*, 12(1):59–68, January 1986.

[JPAB00] R. Jiménez Peris, M. Patiño Martínez, S. Arévalo, and F.J. Ballesteros. TransLib: An Ada 95 Object Oriented Framework for Building Dependable Applications. *Int. Journal of Computer Systems: Science & Engineering*, 15(1):113–125, January 2000.

[Kie00] J. Kienzle. Exception Handling in Open Multithreaded Transactions. In *ECOOP. Proc. of Exception Handling in Object-Oriented Systems*, Nice, France, June 2000.

[KJRP01] J. Kienzle, R. Jiménez Peris, Alexander Romanovsky, and Marta Patiño Martínez. Transaction Support for Ada. In *Proc. of Int. Conf. on Reliable Software Technologies*, volume LNCS-2043, pages 290–304. Springer, May 2001.

[Lis88] B. Liskov. Distributed Programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[MHL+98] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. In *Recovery Mechanisms in Database Systems*, pages 145–218. Prentice Hall, 1998.

[Mic00] Sun Microsystems. *Enterprise Java Beans Specification*. 2000.

[Mos85] J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. MIT Press, 1985.

[OMG95a] OMG. *CORBA Services: Common Object Services Specification*. OMG, 1995.

[OMG95b] OMG. *The Common Object Request Broker: Architecture and Specification*. OMG, 1995.

[PJA98] M. Patiño Martínez, R. Jiménez Peris, and S. Arévalo. Integrating Groups and Transactions: A Fault-Tolerant Extension of Ada. In *Proc. of Int. Conf. on Reliable Software Technologies. LNCS 1411*, pages 78–89. Springer, June 1998.

[PJA00] M. Patiño Martínez, R. Jiménez Peris, and S. Arévalo. Group Transactions: An Integrated Approach toTransactions and Group Communication. In *Europ. Work. on Dependable Computing, EWDC-11*, Budapest, Hungary, 2000.

[PJA01] M. Patiño Martínez, R. Jiménez Peris, and S. Arévalo. Exception Handling in Transactional Object Groups. *LNCS-2022*, 2001.

[RMW97] A. Romanovsky, S.E. Mitchell, and A.J. Wellings. On Programming Atomic Actions in Ada 95. In *Proc. Conf. on Rel. Soft. Tech.*, pages 254–265, June 1997.

[SDP91] S. K. Shrivastava, G. N. Dixon, and G. D. Parrington. An Overview of Arjuna. *IEEE Software*, 8(1):63–73, January 1991.

[Tra95] TransArc Corporation, Pittsburgh, PA 15219. *Encina Transactional-C Programmers Guide and Reference*, 1995.

[WB97] A. Wellings and A. Burns. Implementing Atomic Actions in Ada 95. *IEEE TSE*, 23(2):107–123, feb. 1997.

[XRea95] J. Xu, B. Randell, and A. Romanovsky et al. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *Proc. of FTCS-25*, pages 499–509, 1995.

[XRea99] J. Xu, B. Randell, and A. Romanovsky et al. Rigorous Development of a Safety-Critical System Based on Coordinated Atomic Actions. In *Proc. of FTCS-29*, pages 68–75, 1999.

[XRR98] J. Xu, A. Romanovsky, and B. Randell. Coordinated Exception Handling in Distributed Object Systems. In *Proc. of ICDCS-18*, pages 12–21, May 1998.