Implementation of Conversations in the Conventional Ada

A. Romanovsky and L. Strigini

TECHNICAL REPORT SERIES

No. 475 June, 1994
Implementation of Conversations in the Conventional Ada

A. Romanovsky and L. Strigini

Abstract

The purpose of this paper is to give an approach for using backward error recovery in Ada. We are not going to consider the shortcomings of Ada or to propose new language constructions and new run-time algorithms for Ada but we shall try to offer an approach to how software diversity could be used within this language. To use our approach, application programmers are to develop all redundant software in accordance with our recommendations. We believe that Ada has enough facilities to allow using software diversity while developing fault-tolerant systems. Two of the main advantages of this scheme are its functioning within the widely used conventional industrial language and its suitability for real time systems of an iterative type and with time constraints. We hope these will allow to use our ideas in real practice in the near future.
Bibliographical details

ROMANOVSKY, Alexander

Implementation of Conversations in the Conventional Ada


(University of Newcastle upon Tyne, Computing Science, Technical Report Series, no. 475)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE.
STRIGINI, Lorenzo.

Abstract

The purpose of this paper is to give an approach for using backward error recovery in Ada. We are not going to consider the shortcomings of Ada or to propose new language constructions and new run-time algorithms for Ada but we shall try to offer an approach to how software diversity could be used within this language. To use our approach, application programmers are to develop all redundant software in accordance with our recommendations. We believe that Ada has enough facilities to allow using software diversity while developing fault-tolerant systems. Two of the main advantages of this scheme are its functioning within the widely used conventional industrial language and its suitability for real time systems of an iterative type and with time constraints. We hope these will allow to use our ideas in real practice in the near future.

About the author

A. Romanovsky is a visiting research fellow in the Department of Computing Science at the University of Newcastle upon Tyne, on leave from the Department of Applied Mathematics at St. Petersburg State Technical University, Russia.

Lorenzo Strigini is a "Ricercatore" at Instituto di Elaborazione della Informazione, CNR, Pisa, Italy.

Suggested keywords

BACKWARD ERROR RECOVERY
CONVERSATIONS
SOFTWARE DIVERSITY
SOFTWARE FAULT TOLERANCE

Suggested classmarks (primary classmark underlined)
Dewey (18th): 001.6425 001.64404
U.D.C. 681.322.06 519.687
Implementation of Conversations in the Conventional Ada
A. Romanovsky*, L. Strigini**

* Computing Department, University of Newcastle upon Tyne, NE1 7RU, UK. E-mail: alexander.romanovsky@newcastle.ac.uk
** IEE-CNR, via Santa Maria, 46 Pisa, 56126 Italy. E-mail: strigini@iei.pi.cnr.it

1. Abstract
The purpose of this paper is to give an approach for using backward error recovery in Ada. We are not going to consider the shortcomings of Ada or to propose new language constructions and new run-time algorithms for Ada but we shall try to offer an approach to how software diversity could be used within this language. To use our approach, application programmers are to develop all redundant software in accordance with our recommendations. We believe that Ada has enough facilities to allow using software diversity while developing fault-tolerant systems. Two of the main advantages of this scheme are its functioning within the widely used conventional industrial language and its suitability for real time systems of an iterative type and with time constraints. We hope these will allow to use our ideas in real practice in the near future.

2. Introduction
The paper deals with the problems of recovery based on rolling a system back to the previous state preceding the error occurrence. This type of recovery is termed backward error recovery as opposed to the forward error recovery which is based on correcting the system after detecting an error and putting it into a certain known correct state which it could have reached but for the errors in its operation [1]. The most essential merit of the backward error recovery is that general recovery tools can be created to facilitate the work of programmers developing fault tolerant systems. This is due to the fact that this kind of recovery relies on a unified reaction to the detected error of any type: it is the system rollback.

Developing tools for fault tolerance is much more complicated when the system is a group of communicating processes. It is obvious that, in a system like that, it is by no means sufficient to roll back a faulty process if this has interacted with other processes.

The problem of recovery in communicating process systems has been dealt with in a great number of papers, e.g. [2-10]. The classical one [2] describes the general conversation scheme which has become the basis from which
all further proposals proceed. All recovery schemes for parallel process systems fall into static and dynamic ones. The former (e.g. the colloquy scheme in [3], the S-conversation scheme in [4], four schemes for the language with monitors in [5] or the exchange scheme in [6]) are based on a static description of the rollback region by means of special language constructions of concurrent programming languages. These schemes always include using software diversity by developing several alternates and acceptance tests and by executing alternates successively till acceptance tests are satisfied. The dynamic scheme operation [9, 10] is transparent for (unplanned by) programmers and relies on processing the information about the events occurring in processes that are relevant for determining the rollback region. We are going to consider only the static approach.

As we have mentioned before, several particular schemes have been proposed for different languages on the basis of the initial conversation scheme. As a rule, they consider an extension of a conventional concurrent language. We wanted to find an appropriate approach which could be used within the standard Ada. After consideration all known schemes, we chose two of them as a basis: Kim’s concurrent recovery block scheme [5] and Anderson’s and Knight’s exchange scheme [6].

3. Conversations in Ada
In our scheme, a fault-tolerant unit of software (a subprogram in Ada) is to be built as a set of alternates. Each of these alternates has the same interface (formal parameters) as the unit has. These alternates are to be considered as different implementations (with different algorithms, by different programmers, etc.) of the required unit. There are several ways to specify and to check the local acceptance test in every alternate. The specification of the acceptance test includes as a minimum that no exceptions are raised during the execution of the alternate (without being treated in the alternate itself). Local tests may be different for different alternates. After completing the alternates, the global acceptance test is to be checked. This should probably be developed by an independently working programmer and check the correctness of the results (by checking some invariants or some required specifications) of each alternate. If an error is detected either by the local tests or by the global test, the execution of the next alternate is initiated.

All concurrency in this approach occurs within each alternate by spawning tasks in its body. These tasks communicate by rendezvous or by data sharing within an alternate. There can be different numbers of tasks in alternates. An alternate is completed when all of its tasks terminate.
Our conversation can be nested, which means that any task can call a subprogram developed as a conversation.

Besides, we introduce a deadline mechanism: the programmer can restrict the execution of each alternate and therefore the execution of the whole conversation. With the limitation of the timing primitive of the language (timed entry call in Ada), it is guaranteed that the user of the subprogram will have the answer within the given deadline (it can be either the correct result or the signal of a failure). This mechanism is vital for using software diversity in real-time applications.

This scheme is designed in this way because it is quite natural to use it in Ada. We tried to find the most suitable scheme for this language having in mind its concurrency tools and its structure of units.

One research we want to mention especially is the approach devised by Clematis and Gianuzzi [8] to building a conversation scheme within the conventional Ada language. We completely agree with the authors' idea of using the conventional Ada language to software fault-tolerance by giving to application programmers a template and conventions for writing conversations. We give a comparison of this and our approaches in the last section.

4. How to write conversations in Ada
Because of the peculiarities of our approach, we do not introduce any new language constructs. That is why we give the skeleton of our conversation in the conventional Ada and describe the set of conventions for application programmers developing conversations. A fault tolerant procedure (Sort in our example) can be developed with software diversity (two alternates Sort1 and Sort2 in our example). The skeleton of our conversation has the following structure:

```ada
-- conversation example
package NiceSort is
    FAILURE: exception;
    procedure Sort(list in params; list out params);
end NiceSort;

package body NiceSort is
    -- real-time constraint for entire procedure Sort:
    T: constant := 10.0;
```
procedure Sort1(list in params; list out params;
    localtest: in out BOOLEAN);

procedure Sort2(list in params; list out params;
    localtest: in out BOOLEAN);

function GlobalTest(list out params) return BOOLEAN;

procedure Sort(list in params; list out params) is
    type ALTERNATE_RANGE is range 1 .. 2;
    alternate: ALTERNATE_RANGE:=1;
    localtest: BOOLEAN:=TRUE; -- as a default
begin
    loop
        case alternate is
            when 1 =>
                Sort1(list in params, list out params, localtest);
            when 2 =>
                Sort2(list in params, list out params, localtest);
            when others => raise FAILURE;
        end case;
    exit when localtest
    and then GlobalTest(list of out param);
    localtest:=TRUE;
    alternate:=alternate+1;
    end loop;
end Sort;

procedure Sort1(list in params; list out params;
    localtest: in out BOOLEAN) is
    task T11;
    .... -- N tasks form the first alternate
    task T1N;

    -- special internal task for conversation
    -- and real-time constraint control:
    task WATCHDOG is
        entry ACTION;
        entry OKAY;
    end WATCHDOG;

    -- application task bodies:
    task body T11 is
    begin
        -- .......
        WATCHDOG.OKAY;
    end T11;

    task body T1N is
    begin
        -- .......
    end T1N;

    task body WATCHDOG is
begin
select
  accept ACTION;
  abort T11, ..., T1N;
  localtest := FALSE;
or
  accept OKAY;
  localtest := TRUE;
or
-- ALT1*T is constraint for the first alternate:
  delay ALT1*T;
  abort T11, ..., T1N;
  localtest := FALSE;
end select;
end WATCHDOG;

begin -- Sort1 body
  null;
  exception
    when others =>
      abort T11, ..., T1N, WATCHDOG;
      localtest := FALSE;
end Sort1;

procedure Sort2(list in params; list out params;
                  localtest: in out BOOLEAN) is
  task T21;
  .... -- M tasks form the second alternate
  task T2M;
  task WATCHDOG is
    entry ACTION;
    entry OKAY;
  end WATCHDOG;

  -- application task bodies:
  task body T21 is
    begin
      -- ........
      WATCHDOG.OKAY;
    end T21;
  task body T2M is
    begin
      -- ........
    end T2M;
  task body WATCHDOG is
    begin
      select
        accept ACTION;
        abort T21, ..., T2M;
        localtest := FALSE;
or
        accept OKAY;
    end select;
end WATCHDOG;

localtest:=TRUE;

or
-- (1-ALT1)*T is constraint for the second alternate:
delay (1-ALT1)*T;
abort T21,..., T2M;
localtest:=FALSE;
end select;
end WATCHDOG;
begin --Sort2 body
null;
exception
when others =>
 abort T21,..., T2M, WATCHDOG;
localtest:=FALSE;
end Sort2;

function GlobalTest(list out params) return BOOLEAN is
begin
-- ...return ...
end GlobalTest;
end NiceSort;

The user of this package can call procedure Sort without knowing of the diversity used in developing this procedure. Besides, the concurrency in executing this procedure is hidden from the user as well. The only way to know about a non-tolerable error during the procedure execution is analysing exception FAILURE.

In our scheme, a conversation can have several alternates (two in the example: subprograms Sort1 and Sort2) each of which must be developed as a procedure (we shall use the term "alternate-procedure") with a set of inner tasks. These tasks must be written so that they should only interact within their own alternate. Any task can check its own local acceptance test and, besides, after termination of all alternate tasks the global acceptance test can be checked. If no error occurred during the execution of any alternate and both the local tests and the global test have been satisfied, we consider that this conversation is completed and the result of its execution can be returned. Otherwise the next alternate starts.

Now let us enumerate all important conventions for the application programmer to follow. Only in this case we could guarantee the proper functioning of the conversation scheme.

The developer of the fault-tolerant procedure Sort is to develop several alternates (two in our example) as different procedures (alternate-procedures) with the same formal parameter list and with one additional parameter (localtest in our example) of the BOOLEAN type (this is a sign
of detecting an error while executing a given alternate: it can be signalled by any task - participant of alternate or by catching any exception during the alternate execution; see the bodies of Sort1 and Sort2).

The following set of conventions prevents an alternate-procedure from producing side effects (outside its local state), and thus obviates the need for the run-time support to provide a roll-back operation. We require that the developer of the alternate-procedure does not use in out parameters at all, that there is no direct operation with the global data external to the alternate-procedure, that the global variables and the data are in procedure parameters, that all the local variables are to be initialised in the beginning of the alternate-procedure's body, that the pointers are not to be used as parameters. All results are to be returned as the parameters of the procedure. In case the alternate-procedure calls a procedure from another package, this latter procedure must not produce any side effects just as the alternate as a whole (no changing in the global data of any package and no operating with the outer world). We will discuss later how such conventions can be enforced. All this provided, aborting all alternate tasks guarantees the absence of any influence on other parts of system. Note that these restrictions are similar to the well-known structured programming conventions of working with data and parameters in procedures [11].

The internal structure of an alternate is as follows. It consists of a set of tasks. These tasks start when the execution of the alternate body is started. The tasks interact by rendezvous. The result of their running is the set of out parameters of the procedure-alternate. For controlling the execution of the task sets, service task WATCHDOG is to be introduced in each alternate. This task knows about the deadline given to this alternate (ALT1*T for first alternate and (1-ALT1)*T for second one in our example - allocation of these time-outs is under control of programmer) and after this deadline it deletes all tasks and signals an error. Besides, the WATCHDOG task has two entries. The ACTION entry is used by any task when it checks some local test and decides that its execution is not correct. In this case this task is to call the ACTION entry. Besides, one of the tasks from this set is to call the OKAY entry (and this call must be its last statement) when it has made sure that all the local tests have been checked and the tasks in the alternate have terminated. The WATCHDOG task initiates the recovery in two cases: no task has called the OKAY entry within a time constraint; one of the tasks has called the ACTION entry. In these situations the WATCHDOG task aborts all application tasks and signals an error for starting the next alternate. Note that we use the Ada abort operation only in the exceptional case when an error has been detected.
This is actually one of two solutions that we have considered for coordinating the termination of the cooperating tasks and the watchdog. In this solution, simpler but probably less generally useful, it is the responsibility of the tasks to guarantee that when one of them calls the OKAY entry all other tasks in the alternate-procedure have been completed: unless this is required by the function to be performed, the application programmers have to develop some additional termination protocol. In the Appendix we give the other solution, where each of the tasks in the alternate-procedure has to call the DONE entry, and the WATCHDOG task, knowing the number of tasks in the alternate-procedure, can be assured of the completion of all other tasks.

The body of the (fault-tolerant) procedure called by the user is as follows (see procedure Sort in our example). It consists of calling alternates successively and checking the local test and the global test. In our example, there are two alternates Sort1 and Sort2. We consider that they have time constraints $\text{ALT1} \cdot T$ and $(1 - \text{ALT1}) \cdot T$ respectively; where $T$ is the time constraint for executing the entire procedure Sort. This allocation of time to the alternates is the responsibility of the application designer, who has to derive it from the timing requirements of the application and the execution times of the alternates after the first one (if another conversation is nested in one task of the alternate, a proper fraction of the time allotted to the alternate shall be given as the whole time allowed for the whole nested conversation). Exception FAILURE can be raised only within the body of this procedure when all alternates have been exhausted. In case of a nested conversation, this exception should be used to give a signal from the nested conversation about a non-tolerable fault in its execution.

One additional convention to avoid information smuggling from conversations is that programmers must avoid any output operations with files, controlled devices, operator, etc. during alternate execution; all these operations are to be executed before or after the call to the fault-tolerant procedure.

Let us briefly sum up the conventions for the developer of an alternate-procedure:

1. no in out parameters, no in pointers;
2. no input/output operations;
3. no modifying operations with global data;
4. tasks started within alternate may only rendezvous between themselves;
5. for all procedures, called by alternate-procedure, conventions 2-4 apply.
Note that within this approach every alternate has a guaranteed time of execution: the result or the signal of error is given within a guaranteed time constraint. It is important when a task uses a nested conversation because it is sure to get an answer within a given interval.

5. Discussion of the scheme proposed
Let us first discuss the scheme proposed without any regard for its implementation.

Unlike the scheme proposed in this paper, Kim's "concurrent recovery block" scheme in [5] is intended for extended Concurrent Pascal and it does not allow a designer to impose any deadline constraint. The exchange scheme in [6] has been discussed on a rather different concept level and neither the language instantiation nor an opportunity to nest conversations have been proposed. Besides, it is not described specifically how alternates can be used in this scheme within a given time constraint.

Rather serious problems are known to be involved in introducing the planned backward error recovery schemes into parallel programming languages (see [2, 5, 7]). Those which are most discussed include the possibility of a deadlock and the "deserter process" problem.

It is obvious that with our approach the deserter process is impossible. Besides, provided all software has been developed in accordance with our conventions, the domino effect is impossible as well.

The structuring of concurrency through processes and messages (where the state of a computation is the set of states of all its components, as opposed to the dual style of objects and actions [12], where the long-term state is kept in specialised data objects) is typical of, among others, real-time control applications. Our approach within the Ada language can be easily used to structure precisely real time systems of an iterative type. Of course, Ada has known problem in setting hard time-outs: a delay statement "suspends further execution ... for at least the duration specified" [13]. This is a problem that we cannot overcome, and it is for Ada implementors and for the developers of the new versions of the Ada standard to solve [14]. More generally, time-outs, however "soft", are necessary in most practical application, to avoid waste of resources and the propagation of errors from "runaway" processes. Note that our approach (like those in [5, 6]) differs from the original one [2] in one respect. In our scheme, all concurrent tasks are initiated as soon as the alternate is started but in [2] and in several other approaches they can enter the conversation
at different times. This is an example of the simplification intended to make this scheme usable in practice and this is full in accordance with the opinion of other authors [6, 12] of how conversations should be used in practice.

A disadvantage of this approach could be seen in the need for spawning tasks when each alternate starts. There are two counter-arguments.

First, this is one of the well-known approaches to designing concurrent software (e.g. fork-join, cobegin-coend) and its suitability depends on the application area. As for the cost, the time overhead of process creation is one of the important characteristics of all modern operating and run time systems; it is decreasing and it is always possible to know this time in advance and to decide whether the scheme proposed can be used. And, lastly, this style in designing concurrent software is in accordance with top-down, structured engineering and with information hiding.

It is here that the main difference between our scheme and the colloquy scheme in [3] lies. The authors of the colloquy scheme consider that the entire system consists of a set of long-living processes which can join one discuss (alternate in our scheme) but are not supposed to participate in the next one (after a fault has been detected in the previous one).

The main disadvantages in using/implementing our scheme are precisely the reverse side of its advantages. As long as a developer is limited to a standard Ada tool set, we can not check the accordance of software to our conventions during either the compiling or the run time. If the developer breaks any of our conventions there can be serious consequences without any warning. On the other hand, two solutions exist, using: a) a post-processor to check that these conventions are respected, or b) a pre-processor which takes special added constructs and expands them into convention-abiding conversation. Both solutions of course are equivalent to imposing a dialect or a subset of Ada, but result in pure Ada source code: similar conventions have been proposed for applying Ada in safety-critical projects. Actually, such tools will by necessity only accept programs whose conformance with the conventions can be checked statically, thus being more restrictive than a programmer applying the conventions "manually" with a knowledge of the run-time behaviour of his/her program. We regard this as an acceptable restriction.

Note that within the approach proposed there is no need to develop a recovery point tool.
The Ada conversation scheme proposed in [8], which we quoted before as part of an approach similar to ours, is based on introducing a service task - the conversation manager - for every conversation. This task has a special structure; it monitors and synchronizes entry to conversation, alternate execution, the acceptance test check for all processes-interlocutors. This scheme does not allow to impose any deadline constraints, it operates under an assumption that the same set of processes takes part in all alternates. Besides, the "deserter process" can stop the operating of the entire system both in the entry to a conversation and in the acceptance test check. Programmers have to develop their own recovery point tools and take care of not only saving, discarding or restoring information but of recovery point nesting as well.

We hope that our approach is quite natural for Ada programming where concurrency can be hidden in procedures in a natural way; all tasks (including a service task for their control) can be spawned when the procedure is called; and where a natural procedure nesting can be considered with some restrictions as a recovery point nesting.

Our approach is also similar to the one in [15], where the conventional Ada has been used for developing atomic actions and for structuring concurrent application software as atomic actions. This approach has been used for introducing forward error recovery in communicating tasks (on the basis of a simultaneous spreading of exceptions in all tasks involved in an atomic action).

We hope that the main advantage of our scheme is its functioning within a conventional wide-spread industrial language and that this will allow the practical use of our ideas in the near future.

Acknowledgements.
This work has been supported by ESPRIT Basic Research Action 6362 "Predictably Dependable Computing Systems (PDSC2)".

References:


Appendix. The second approach to implement **WATCHDOG** service task

This service task has two entries: DONE and ACTION. It initiates recovery in two cases: not all of the tasks have called the DONE entry within a time constraint; one of the tasks has called the ACTION entry.

```pascal
task body WATCHDOG is
  -- task number in CONV1
  TaskNumber1: constant:=N;
  -- constraint for 1st alternate:
  TimeConstr1: constant:=ALT1*T;
  localtst: BOOLEAN:=TRUE;
  task_count: INTEGER:=0;
  last_event_time: TIME;
  constr_part, timedelta: DURATION;
  begin
  last_event_time:=CLOCK;
  constr_part:=TimeConstr1;
  loop
    select
      accept ACTION;
      abort T11, T1N;
      localtst:=FALSE; exit;
    or
      -- one task end:
      accept DONE;
      timedelta:=CLOCK-last_event_time;
      constr_part:=constr_part-timedelta;
      last_event_time:=last_event_time+timedelta;
      task_count:=task_count+1;
      if task_count=TaskNumber1 then exit;
    end if;
    or
      -- time constraint
      delay constr_part;
      abort T11, T1N;
      localtst:=FALSE; exit;
    end select;
  end loop;
end WATCHDOG;
```