
Further information on publisher website:
http://www.ietdl.org

Publisher’s copyright statement:
This paper is a postprint of a paper submitted to and accepted for publication in Software Engineering Journal and is subject to Institution of Engineering and Technology Copyright (http://www.ietdl.org/journals/doc/IEEDRL-home/info/support/copyinf.jsp). The copy of record is available at IET Digital Library.

Always use the definitive version when citing.

Use Policy:
The full-text may be used and/or reproduced and given to third parties in any format or medium, without prior permission or charge, for personal research or study, educational, or not for profit purposes provided that:

- A full bibliographic reference is made to the original source
- A link is made to the metadata record in Newcastle E-prints
- The full text is not changed in any way.

The full-text must not be sold in any format or medium without the formal permission of the copyright holders.
Backward error recovery via conversations in Ada

by A. Romanovsky* and L. Strigini**

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

This paper proposes an approach for using backward error recovery in Ada. We do not discuss advantages and disadvantages of Ada, nor propose new run-time algorithms for Ada, but we try to offer practical method for using backward recovery and software diversity within this language. We believe that Ada has sufficient facilities to allow using software diversity in developing fault-tolerant systems. However, previous researchers have noted problems in attempting to use this possibility, and restrictive rules are necessary to avoid these problems. We consider "conversations" for coordinated backward recovery of concurrent processes and propose: i) a restricted scheme similar to Kim's "concurrent recovery block", but providing for deadlines on the execution of the diverse modules; ii) programming rules for applying this scheme to Ada procedures; and iii) a way for automatically enforcing these rules through a source code pre-processor. Two of the main advantages of this scheme are its functioning within this widely used conventional industrial language and its suitability for real-time systems of an iterative type and with time constraints.

1 Introduction

This paper deals with the problems of error 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 "forward error recovery" which is based on correcting the system state after detecting an error, moving the system to a certain known correct state which it could have reached but for the errors in its operation [1]. The most essential merit of 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, i.e. system rollback, to
detected errors of any type.

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

[2] proposed the use of a programming construct, called a *recovery block*, which
combined checkpointing and backward recovery with retry by a diverse variant of the
code. This allowed redundancy and design diversity to be hidden inside program blocks:

```
ensure Acceptance Test
by Alternate1
else by Alternate2
    . . . . . . .
else by Alternate_n
else error.
```

The $Alternate_i$ modules are diverse implementations of the function specified for
the whole recovery block, so that if an execution (by $Alternate_1$, for instance) fails
the *Acceptance Test*, the following retry, by, for instance, $Alternate_2$, may not
repeat the same error. Each retry executes a different $Alternate_i$ (and is subject to
the same *Acceptance Test*) and exchanges messages that may differ (in their
sequence, or in their contents) from those exchanged by the previous alternate. So, co-
ordinated roll-back is necessary. To this end, the same paper proposed *conversations*. A
*conversation* (Fig. 1) can be described as a multi-process recovery block: when two or
more processes enter a conversation, each must checkpoint its state, and they may only
leave the conversation ("committing" the results computed during the conversation, and
dISCARDING their checkpoints) by consensus that all their acceptance tests are satisfied.
Processes can asynchronously enter a conversation, but all must leave it at the same
time. During the conversation, they must not communicate with any process outside the
conversation itself (violations of this rule are called *information smuggling*). So, the
occurrence of an error in a process inside a conversation requires the roll-back of all and only the processes in the conversation, each 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 $i+1$.

Paper [2] did not specify an implementation or language construct for the conversation scheme. Several proposals have later appeared of language constructs and implementations, differing in the resulting semantics of conversations. 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 constructs of concurrent programming languages. These schemes always include using software diversity, obtained by developing several alternates and acceptance tests and by executing alternates successively till acceptance tests are satisfied. The operation of the dynamic schemes [7, 8] is transparent for (unplanned by) programmers and relies on processing the information (about the events occurring in processes) that is relevant for determining the rollback region. We are going to consider only the static approach.

Most implementations proposed for the conversation approach [2], for different languages, propose an extension of conventional concurrent languages. We wanted to find an approach that could be used within the standard Ada language [9]. After considering all known schemes, we chose a restrictive scheme very similar to Kim’s *concurrent recovery block* scheme [5] and, to some degree, to Anderson’s and Knight’s *exchange* scheme [6]. The restriction consists in that conversations are only allowed among a set of processes spawned together (Fig. 2): a process cannot freely form conversations with arbitrarily chosen other processes in the system. Hence the name "concurrent recovery block": from outside, an alternate is indistinguishable from a sequential block of execution, like an alternate in a sequential recovery block.
The approach we propose relies on the programs being structured according to special, restrictive rules. To automatically enforce these rules, we propose that programmers use a dialect of Ada (including a few special constructs for building conversations), which can be translated into standard Ada by a simple source code pre-processor.

In the rest of the paper, we describe the structure of our restricted Ada conversations (Section 2), and introduce our dialect in Section 3. In Section 4 we show how a conversation is coded in Ada, and in Section 5 we specify the pre-processor which produces this Ada code from the dialect used by programmers. Section 6 discusses the pros and cons of our proposal, comparisons with other proposals and possible developments.

2 Conversations in Ada

In our scheme, a fault-tolerant unit of software is an Ada procedure (a similar scheme could be realised for functions), and will be called a "fault tolerant procedure", or ft-procedure. It is built as a set of alternates, each a procedure with the same interface (formal parameters) as the whole ft-procedure. The caller of the subprogram does not need to be aware of either its fault-tolerant implementation or its internal concurrency. Any concurrency occurs within an alternate, where multiple tasks can be spawned; these tasks communicate, by rendezvous or by data sharing, within the alternate. Different alternates may have different numbers of tasks. An alternate is completed when all of its tasks have terminated. Each task contains "local" acceptance tests, checking variables accessible to that task, and a "global" acceptance test may be specified to be performed after the spawned tasks have terminated but before returning from the fault-tolerant subprogram (i.e., committing the conversation). If any test fails, the next alternate is executed. If all alternates fail, a FAILURE exception is raised. Conversations can be nested, which means that any task can call a subprogram which is in its turn structured as a conversation.

Each task can use two special statements to terminate an alternate: one to signal that the local acceptance test (any check internal to the task) is not satisfied and the other that the
task completed correctly. Tasks which do not call either statement are eventually aborted via a time-out. Local acceptance tests do not require any special operator or construct; they may be composed of sub-tests (checking different necessary conditions for correctness) spread out in time along the execution of the alternate to obtain early detection of errors. The specification of the acceptance tests is of course to be decided based on the specification and details of the ft-procedure. However it includes, as a minimum, that all exceptions raised during the execution of an alternate must be treated without propagating outside the ft-procedure; our scheme includes a catch-all handler for this purpose, to abort the alternate upon unforeseen exceptions (this covers the case of a hardware error, if it has been treated properly by the Ada run-time system and caused the raising of the appropriate, standard exception). The treatment of exceptions obviously requires some caution. In each alternate, each task is allowed to have its own handlers for its exceptions and for the FAILURE exceptions arising from its own calls to (nested) ft-procedures: all these are local problems of the given task. If the task fails to process an exception, it will be aborted in due time.

In addition, we introduce a deadline mechanism: the programmer can bound the execution time of each alternate and therefore of the whole conversation. Within the limitations of the timing primitive of the language (the specification of the timed entry call in Ada does not provide for an actual real-time deadline mechanism), the caller of the fault-tolerant subprogram is thus guaranteed to receive either the correct result, or the notification of a failure, by the time the run-time support signals the expiration of the deadline. This is especially important when a task in a conversation uses another, nested conversation, as the duration of the calling task, and hence of the alternate where it belongs, can still be bounded.

We shall specify rules guaranteeing that the fault-tolerant subprogram is side effect-free. Otherwise, "information smuggling" could not be prevented with an ordinary Ada runtime support. As a bonus, the absence of side-effects also makes a checkpointing mechanism unnecessary.
3 Ada dialect for conversations

As explained before, we specify a dialect of Ada, obtained by adding special constructs for conversations and imposing some restrictions on the allowable uses of the standard parts of the language. This dialect is meant to be paired with an appropriate pre-processor, which translates a program written in the dialect into conventional Ada. In Section 4 we will discuss how conversations could be programmed in standard Ada (this will be described as a set of conventions and as a template for programmers to follow) and therefore the peculiarities of our pre-processor (which is then specified in Section 5).

Our dialect has statements similar to those often used in the literature to describe conversation schemes (and especially for concurrent recovery blocks). Ada is a complex, flexible language which gives a designer many choices and opportunities. Our dialect does not preserve all these opportunities because that would mean an extremely complex pre-processor.

Thus, we propose a special way of developing a ft-procedure. The declaration of a ft-procedure should be given as follows (we use **bold italic** for our new constructs, **bold** for Ada keywords):

```
ft_procedure Name( declaration of parameters);
```

The caller of a ft-procedure can receive either a result in the **out** parameters or the predefined exception FAILURE raised within this procedure. The latter signals a non-tolerated error during the execution of the ft-procedure.

The body of this ft-procedure is specified in our dialect as follows:

```
ft_procedure Name( declaration of parameters) is
  ...   -- declarative part **
  ensure Test( list of actual parameters)
  by Altern1
  else by Altern2
  ...
  end Name;
```
The global test for the given conversation, an essential part of the conversation design, is represented by function Test. The list of actual parameters to the Test function call can be a subset of the list of formal parameters given to the ft-procedure. This function must have no side effect and all its parameters should be of in mode. We allow these to include out parameters of the ft-procedure. This is necessary, as the acceptance test is meant to check precisely that the results of the ft-procedure are as intended, although it is an apparent exception to Ada rules, which forbid a called procedure from ever seeing (or making visible to the subprograms it calls) the contents of its out parameters. Sections 4 and 5 describe how this special exception can be implemented without violating the general Ada rule.

The specifications of the Altern1, Altern2, ..., and Test subprograms and their bodies can be written either in the declarative part of the ft-procedure (position marked by comment ** above) or outside the ft-procedure. These subprograms should be declared as follows:

```plaintext
alternate Altern1(declaration of parameters of ft-procedure);
alternate Altern2(declaration of parameters of ft-procedure);
...
function Test(declaration of some of parameters of ft-procedure)
    return BOOLEAN;
```

In this dialect each alternate body looks as follows:

```plaintext
alternate Altern1(declaration of parameters of ft-procedure)
within time_out1 is
... -- declaration of shared variables
... -- and common types for application tasks (if any)
task T11 is
    entry ...;
    ...
end T11;
...
task T1N is
    entry ...;
    ...
```
end T1N;
task body T11 is
begin
...
[alternate_error;] -- zero or more of these calls, depending
-- on which checks of necessary properties can
-- be included in this task
...
task_done; -- normal termination of task
end T11;
...
task body T1N is
begin
...
[alternate_error;]
...
task_done;
end T1N;
end Altern1;

where

- T11, ..., T1N are application tasks, and their concurrent joint execution
  represents an execution of an alternate Altern1;

- time_out1 is a time-out for the execution of the alternate Altern1 (if the
time-out expires, all tasks are terminated, this alternate is considered unable to
ensure Test and the next available alternate, if any, is executed);

- the statement task_done is to be executed in the body of each task when it
  completes execution successfully;

- the statement alternate_error is to be executed in a task when it decides
  that it is not able to complete its execution successfully; its effect is to abort the
  alternate, similarly to the effect of a time-out expiring.

These last two statements can be used after checking some test (invariant, condition)
local for the task or for the set of tasks.
4 How to implement conversations in Ada

We now describe the implementation of a conversation in pure Ada, for the time being in the form of a set of conventions for application programmers developing conversations. This implementation should also be seen, on the other hand, as an output text produced by a pre-processor of the dialect described above. The structure of a ft-procedure is fixed, and consists of calling the alternates successively and checking their acceptance tests, until one alternate passes all tests or alternates are exhausted.

We now list the conventions that our hypothetical application programmer has to follow in order to develop a conversation in Ada using this approach.

The internal structure of an alternate (alternate-procedure), given in the preceding section, is implemented as follows. The alternate consists of a set of tasks, which start when the execution of the alternate body is started. The result of their execution is the set of out parameters of the alternate-procedure. For controlling the execution of the task set in an alternate, an auxiliary task, WATCHDOG, has to be added. This task has a co-ordination role: i) it knows about the deadline assigned (by the programmer) to this alternate and after this deadline it deletes all the other tasks and signals an error; ii) it accepts from the other tasks, through its entry TASK_FAILURE, signals of detected errors in their executions; iii) it accepts from the other tasks (through the TASK_DONE entry) their signals of successful termination: calling TASK_DONE must be the last statement in each of these tasks, after running the local acceptance test. The WATCHDOG task thus waits until all the other tasks have completed successfully, and then terminates. The WATCHDOG task initiates recovery in two cases: i) not all tasks have called the TASK_DONE entry within the deadline; ii) one of the tasks has called the TASK_FAILURE entry. In these situations the WATCHDOG task aborts all application tasks and signals an error for starting the next alternate. We only use the Ada abort statement in this exceptional case, i.e., when an error has been detected.

Each alternate-procedure must have the same formal parameter list as the ft-procedure, with one additional Boolean parameter, alternatesuccess, for signalling errors.
detected while executing that alternate. Such errors may be detected in the tasks and explicitly signalled to the WATCHDOG task, or the signalling may consist in raising an exception.

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. The alternate-procedure must be side-effect free, returning all its results through \texttt{out} parameters. So, it must:

- have no \texttt{in out} parameters;
- have no parameters of type pointer ("access", in Ada terminology);
- contain no assignment to global variables external to it;
- perform no output operations to files, controlled devices, or the operator;
- contain no tasks which rendezvous with tasks outside the alternate-procedure.

If the alternate-procedure calls a procedure from another package, this latter procedure must have no side effects, just as the alternate as a whole (no changes in the global data of any package and no operating with the outer world). All this provided, aborting all tasks in an alternate guarantees no effect of their partial execution on other parts of system. Note that these restrictions amount to a strict adherence to well-known structured programming conventions for working with data and parameters in procedures [10].

The restrictions imposed by Ada on the modes of parameters cause some further complications. Let us suppose that a programmer wishes to use an \texttt{out} parameter of the \texttt{ft}-procedure as an input to the \texttt{Test} function (as will normally be the case: the Test function is supposed to implement an "acceptance test" on the results of the alternates). This is not allowed by the Ada rules. A seemingly simple solution would be for the preprocessor to transform the declarations (in the \texttt{ft}-procedure) of such parameters from \texttt{out} to \texttt{in out}. This would solve the immediate problem, but would have undesirable consequences: for instance, a cascade of procedure calls which seems legal in the dialect
(passing an `out` parameter along the tree of calls) could become illegal once preprocessed. A better solution, detailed in Section 5, is: the pre-processor creates temporary variables in the body of the ft-procedure for each `out` parameter that needs to be used as `in` for the `Test` function; it uses these as actual parameters when calling the alternate-procedures; and it copies the values of these variables to the corresponding `out` parameter of the ft-procedure before the latter returns.

Our conversation scheme allows only a "flat" set of tasks as a set of participant processes, in the sense that each alternate-procedure must:

- contain no calls to procedures in which an internal task starts (unless such procedures are ft-procedures);
- contain no internal tasks declared in the tasks which form the alternate itself.

We want to mention some additional restrictions caused solely by our desire to have a clear syntax for the dialect, a simplified template for the implementation of ft-procedures and a simpler pre-processor. These are: an alternate-procedure must contain no tasks defined as objects of task type, must not execute an allocator to create a new task, and must have `null` as a procedure body. It is obvious that our general scheme allows more complex alternatives as well.

In the Appendix we give a complete example, written in standard Ada, which demonstrates the use of these conventions and the recommended structure of an application program with conversation.

5 Pre-processor for Ada dialect with conversations

The purpose of this Section is to give a complete description of the automatic translation provided by the pre-processor. For this purpose we give strict rules of correspondence between the conversational dialect and conventional Ada constructions and discuss peculiarities of the pre-processor which translates a program written in the dialect into the corresponding conventional Ada program. Even without the pre-processing, these
rules can help a programmer to write conversations without errors in standard Ada as well, serving as a template or a set of rigorous conventions to follow (we discussed these conventions in details in the previous Section).

For each piece of code in the dialect, in the left column, we give the corresponding piece of Ada code in the right column. The declaration of a ft-procedure should be translated in the following way:

<table>
<thead>
<tr>
<th>ft_procedure Name( declaration of parameters);</th>
<th>FAILURE: exception; procedural Name( declaration of parameters);</th>
</tr>
</thead>
</table>

We adhere to the following discipline for exception handling and raising in our dialect. When creating the Ada code, the pre-processor inserts the default catch-all handler (when others => raise FAILURE) into the ft-procedure body. This catches all exceptions and in its turn raises only the FAILURE exception to be interpreted, by the caller of the ft-procedure, as the notification of an error that could not be tolerated within the ft-procedure.

The body of this ft-procedure must be translated as follows:

```
ft_procedure Name( declaration of parameters) is
  ...
ensure  Test(
    list of actual parameters)
by
  Altern1
else by
  Altern2
...
end Name;
```

```
procedure Name( declaration of parameters) is
  ...
type ALTERNATE_RANGE is range 1..M;
alternate: ALTERNATE_RANGE := 1;
alternatesuccess: BOOLEAN;
<temporary replicas of out parameters>
begin
  loop
    case alternate is
```
The body of the ft-procedure is translated into the body of the corresponding Ada procedure by the following steps:

- the number of alternates is calculated and put into the declaration of type ALTERNATE_RANGE as a constant M; the declaration of the type ALTERNATE_RANGE is added;

- the declarations of the variables alternate, alternatesuccess, are added;

- where the place holder <temporary replicas of out parameters> appears, declarations of internal variables are added, one for each out parameter of the ft-procedure that is also an in formal parameter for the function Test;

- the body of the corresponding Ada procedure consists of only one loop, the code copying the values of replicas to out parameters and an exception handler;
- within this loop, the case statement calls successively each alternate, given in the declaration of the ft-procedure after the keywords by or else by;

- each alternate call has the same list of actual parameters as the ft-procedure, with two changes:
  - for those out parameters of the ft-procedure for which a "temporary replica" variable has been created, the latter is passed as an actual parameter to the alternate-procedure instead of the out parameters of the ft-procedure;
  - the additional parameter alternatesuccess is added;

- the function Test given after the word ensure in the Ada dialect is called to check a global test; again, its actual parameters include the "temporary replica" variables instead of actual out parameters of the ft-procedure;

- the exit statement is in this loop after the case statement; it checks alternatesuccess and Test;

- before ending the loop, the variable alternate is increased by one;

- before successfully completing the execution of the ft-procedure body, the contents of the "temporary replica" variables are copied to the corresponding out parameters of the ft-procedure.

Note that only one ft-procedure can be in a declarative part, because it returns the predefined exception FAILURE; we do not concentrate on this problem but our dialect could be extended to allow several ft-procedures to be declared in one block. This can be solved by complicating the pre-processor to have it declare the FAILURE exception only once, or, in a more complex way, by changing the dialect and giving a developer the opportunity to name this exception.

Each alternate declaration should be translated in the following way:
Each alternate body should be translated in the following way:

<table>
<thead>
<tr>
<th>alternate</th>
<th>Altern1(declaration of parameters of ft-procedure);</th>
<th>procedure</th>
<th>Altern1(declaration of parameters of ft-procedure; alternatesuccess: out BOOLEAN);</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>alternatenechnology of parameters of ft-procedure;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>is</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>within time_out1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>task T11 is</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>end T11;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>task T1N is</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>end T1N;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>task body T11 is</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>begin</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>task_done;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>end T11;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>task body T1N is</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>begin</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>alternate_error;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>task_done;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>end T1N;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>end Altern1;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>task T1N is</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>end T1N;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>task body WATCHDOG is</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>begin</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-- task number in alternate 1:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TaskNumber: constant:=N;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-- constraint for 1st alternate:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>TimeConstr: constant:=time_out1;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>task_count: INTEGER:=0;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>this_alt_deadline: TIME;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>begin</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>this_alt_deadline:=CLOCK+TimeConstr;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>loop</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>select</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>accept TASK_FAILURE;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>abort T11, ..., T1N;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>alternatesuccess:=FALSE; exit;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>or</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>-- one task ends:</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Each alternate written in the dialect is processed in accordance with the following rules:

- first, the list of the names of all application tasks declared in the alternate (that is T11, ..., T1N) is collected;
- each statement \textit{task\_done} in the bodies of these tasks is translated into the entry call \texttt{WATCHDOG\_TASK\_DONE};

- each statement \textit{alternate\_error} in the bodies of these tasks is translated into the entry call \texttt{WATCHDOG\_TASK\_FAILURE};

- the specification and the body of the special \texttt{WATCHDOG} task are created in Ada text after the specifications of the all application tasks: i) the list of all tasks is to be used in \texttt{abort} statement; ii) the simple expression \texttt{time\_out1} given with the standard word \texttt{within} is used in the declaration of the constant \texttt{TimeConstr}; iii) the number of tasks is calculated during pre-processing and put in the constant \texttt{TaskNumber}; iv) the default catch-all handler is inserted into the \texttt{ft-procedure} body.

Note that for added robustness, the pre-processor could insert in the \texttt{WATCHDOG} task a separate \texttt{TASK\_DONE} entry for each task: instead of counting the calls, \texttt{WATCHDOG} would mark the successfully completed tasks in an array of \texttt{BOOLEAN} flags.

\section*{6 Discussion and conclusions}

We have proposed a detailed implementation of the conversation scheme, in a restricted form, for the Ada language. We first discuss this scheme in terms of its usability and usefulness, without regard for its implementation.

Before comparing it with other realisations of the conversation approach, it is appropriate to consider the possible criticism that our scheme is a very restrictive form of conversation, in particular because a programmer must limit co-ordinated recovery to tasks that are spawned together. We believe that our scheme is a practical and reasonable simplification of the general conversation concept (see the discussion in Section 2). It is difficult to expect that any practical mapping of a general concept and this concept itself would be equivalent. [11] proves this by discussing the problems and the variety of the implementations of conversations. In particular, a practical scheme should take into consideration the peculiarities of the applications for which it is intended, and our
scheme does this for real-time applications of an iterative type and with time constraints. So, our purpose was similar to the purpose of [6], that is, to deliberately simplify the use of conversations and "to take advantages of the natural synchronisation inherent in real time". In this light, we have considered that the need for co-ordinated recovery for a set of co-spawned tasks seems more likely to arise than for unrelated processes, and our proposal deals precisely with such tasks. The design style based on spawning sets of "sibling" tasks (described e.g. by the fork-join, cobegin-coend constructs) is a popular one for concurrent software, and one that is in accordance with principles of structured engineering and information hiding. Its suitability depends on the application area: this paper does not mean to argue in favour of this design style against others, but to offer a recovery method for those applications where this design style is appropriate. Last, we believe that, as argued in [11], only the simpler forms of conversations, based on static membership and restricted communication mechanisms, have clear realistic applications. The more complex forms have unclear usefulness and clear reliability-related problems of less controllable execution and added complexity in the run-time support. For applications where the pattern of inter-process interactions is flexible and largely decided at run-time, other structuring principles, e.g. atomic transactions, may be more appropriate.

Rather serious problems are known to be involved in introducing the planned backward error recovery schemes into concurrent programming languages (see [11] [5] [12]). Those which are most discussed include the possibility of information smuggling, the "capture effect" and the "deserter process" problem whereby a process may block all its intended partners in a conversation by not joining it.

Information smuggling is avoided by our set of programming rules. To avoid accidental violations of these rules, an automatic code inspection tool can be realised. Two considerations apply here. First, there are many ways for an Ada programmer to make a subprogram produce side-effects, so that such a tool may have to be rather complex. It can be made simpler by making the rules stricter than in our description in Section 4, and such restrictions may well be acceptable, for instance, in a software project using a
so-called "safe subset" of Ada. Second, such a tool would need to check all the source code in an application: when separately compiled units are incorporated, the tool must re-inspect either their code or some appropriate condensed information previously extracted from it. This would not be a serious problem in projects (like safety-critical projects) where configuration and version control has to be ensured by automatic tools; for projects where such full source-level inspectability is considered too onerous, it is likely that enforcement of the rules via visual inspection and non-automated administrative procedures will be considered sufficient.

As for the "deserter process" problem, it is solved by the "concurrent recovery block" structure which implies that all tasks automatically take part in the conversation. Tasks failing to complete an alternate are detected via the time-out facility.

The "concurrent recovery block" structure also avoids the "capture effect", which may limit the parallelism in the system. The "capture effect" (so defined in [12]) occurs if a process needs to enter a conversation just in case it may be required to interact with the participants in that conversation, even during executions when this interaction does not actually take place.

In our scheme, there is no need to develop a checkpointing mechanism. This advantage, and the corresponding limitations to the behaviour of alternate-procedures, are not shared by other implementations of the concurrent recovery block.

We now compare our scheme with others proposed in the literature, starting with the more restrictive schemes which we have taken as our models. Kim's "concurrent recovery block" scheme in [5] is intended for extended Concurrent Pascal and it does not allow a designer to impose deadlines. The exchange scheme in [6] is discussed on a more abstract level and neither the language instantiation nor an opportunity to nest conversations have been proposed.

As correctly pointed out in [12] Ada is an extremely complex language for realising a consistent concept of concurrent backward error recovery which would incorporate all opportunities of the language. The authors of [12] enumerated several problems and
unclear situations (process manipulation, shared objects) which prevent the direct use of the conversation concept in Ada. Our approach to solving these problems is in thoroughly specifying a restrictive but powerful enough subset of the language on which a clear conversational semantics can be imposed.

It is here that the main difference between our scheme and the colloquy scheme in [3] lies. The authors of [3] consider that the entire system consists of a set of long-lived processes which can join one "discuss" (alternate in our scheme) but are not necessarily expected to participate in the next one (after a fault has been detected in the previous one). This has obvious advantages of flexibility, but disadvantages in allowing extreme complexity in application design.

The Ada conversation scheme proposed in [13] is based on introducing a service task - the conversation manager - for every conversation. This task has a special structure; it monitors and synchronises entry to a conversations, execution of alternates and the acceptance test check for all participant processes. This scheme does not allow for deadline constraints, and assumes that the same set of processes takes part in all alternates (in this, our scheme is intermediate between [13] and [3]). A "deserter process" can stop the operation 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 consider it quite restrictive to require that each designer of the participants of any nested conversation knows the level of nesting, and in what outermost conversation this participant is going to take part. This essentially restricts the independent implementation of the parts of the entire system.

Our approach is somewhat similar to the one in [14], where the conventional Ada language is used for developing atomic actions and for structuring concurrent application software as atomic actions. This approach is 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).
In terms of implementation, a disadvantage of our approach could be seen in the need for spawning tasks when each alternate starts. There are two counter-arguments. First, as mentioned above, this is a matter of design style, which should be dictated by the needs of the application. 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. Note also that Ada tasks are normally implemented as light-weight processes.

An important advantage of our approach is that code units to be separately compiled can be separately pre-processed. The declarations of a ft-procedure and of the corresponding alternate-procedures may be spread through separately compiled units. The cost is of course that a programmer has to repeat small parts of the code to ensure cross-checks and cross-binding between separately compiled units, but the advantage is a simpler, 'local' one-pass pre-processor and having most errors detected by the Ada compiler itself.

Copying into temporary variables those out parameters that are used in the function Test could be costly for small ft-procedures. However, it may be desirable to extend it to all out parameters, to obtain an "all or nothing" semantics for the ft-procedure: the ft-procedure either terminates returning correct results, or, if it fails with a FAILURE exception, it produces no visible effect.

In conclusion, 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, WATCHDOG, for their control) can be spawned when the ft-procedure is called; and a natural nesting of procedure calls can be considered, with some restrictions, as a nesting of recovery points.

In terms of possible developments, we notice that our conversation scheme can be quite easily ported to Ada9X [15] because of two reasons: the upward compatibility of the latter with Ada and the new features which simplify the implementation of WATCHDOG
task. In particular, a new delay until statement allows one not to re-calculate the deadline time after each call of the TASK_DONE entry and 'Monotonic' time feature allows to use an actual real-time 'delay' semantics.

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

Acknowledgements

This research was sponsored by the ESPRIT Basic Research Project PDCS2. Part of this research has been done during Dr. Romanovsky's postdoctoral fellowship, sponsored by the Royal Society. The work described has benefited greatly from discussions with colleagues in the PDCS2 project. We appreciate comments by Dr. Franco Mazzanti and Dr. Felicita Di Giandomenico from IEI of CNR.

References


Appendix: Code for conversation in Ada

We give an example of conversation in Ada, developed as described in Sections 2, 4. Ft-procedure Max finds the maximum element in the large matrix A(3*M). For this example, let us assume that the matrix elements are scattered and stored in a distributed fashion, that the search is to be completed in a limited time, that the locations of the matrix elements are not known in advance (the search can take unpredictably long), and the allocation of tasks is also unknown. With these assumptions, this search is organised in the following way. Ft-procedure Max has two alternates: procedures MaxRow and MaxSeq which are called from the body of Max. In the first alternate, MaxRow, each of 3 tasks (called T11, T12, T13) looks for the maximum element in one row, and afterwards each of them (except the third one) sends the element found to task T13 which calculates the maximum element in the entire matrix. If an error occurs or the time-out expires, this alternate is considered to have failed, and the second one, procedure MaxSeq, is started. It has one task which searches for the maximum element sequentially in the entire matrix. Time constraint 40, given for the entire procedure Max, is split between the two alternates: 10 for the first and 30 for the second. Function TestResult checks whether the element found falls within a statically known range (minlimit, maxlimit).

package MaxElem is
  FAILURE: exception;
  procedure Max(A: in big_matrix; elem: out emtype);
end MaxElem;

package body MaxElem is
  procedure MaxRow(A: in big_matrix; elem: out elemtype;
    alternatesuccess: out BOOLEAN);
  procedure MaxSeq(A: in big_matrix; elem: out elemtype;
    alternatesuccess: out BOOLEAN);
  function TestResult(elem: in elemtype) return BOOLEAN;
  procedure Max(A: in big_matrix; elem: out elemtype) is
    type ALTERNATE_RANGE is range 1 .. 2;
    alternate: ALTERNATE_RANGE:=1;
    alternatesuccess: BOOLEAN;
temp_elem: elemtype;
begin
loop
case alternate is
  when 1 => MaxRow(A, temp_elem, alternatesuccess);
  when 2 => MaxSeq(A, temp_elem, alternatesuccess);
  when others => raise FAILURE;
end case;
exit when alternatesuccess
  and then TestResult(temp_elem);
  alternate:=alternate+1;
end loop;
elem:=temp_elem;
exception
  when others => raise FAILURE;
end Max;
procedure MaxRow(A: in big_matrix; elem: out elemtype;
  alternatesuccess: out BOOLEAN) is
  task T11;
  task T12;
  task T13 is
    entry NEWMAX (newelem: in elemtype);
  end T13;
  task WATCHDOG is
    entry TASK_FAILURE;
    entry TASK_DONE;
  end WATCHDOG;
  task body T11 is
    max_tmp: elemtype;
    begin
      max_tmp:=A(1, columns_interval'first);
      for K in columns_interval loop
        if max_tmp<A(1,K) then max_tmp:=A(1,K); end if;
      end loop;
      T13.NEWMAX(max_tmp);
      WATCHDOG.TASK_DONE;
    end T11;
  task body T12 is
    max_tmp: elemtype;
    begin

max_tmp := A(2, columns_interval'first);
for K in columns_interval loop
    if max_tmp < A(2, K)
        then max_tmp := A(2, K); end if;
end loop;
T13.NEWMAX(max_tmp);
WATCHDOG.TASK_DONE;
end T12;
task body T13 is
new_max : elemtype;
begin
    new_max := A(3, columns_interval'first);
    for K in columns_interval loop
        if new_max < A(3, K)
            then new_max := A(3, K); end if;
    end loop;
    for K in 2 .. maxrow_number loop
        accept NEWMAX(newelem: in elemtype) do
            if newelem > new_max
                then new_max := newelem; end if;
        end NEWMAX;
    end loop;
    elem := new_max;
    WATCHDOG.TASK_DONE;
end T13;
task body WATCHDOG is
    TaskNumber : constant INTEGER := maxrow_number;
    TimeConstr : constant := 10.0;
    task_count : INTEGER := 0;
    this_alt_deadline : TIME;
begin
    this_alt_deadline := CLOCK + TimeConstr;
loop
    select
        accept TASK_FAILURE;
        abort T11, T12, T13;
            alternatesuccess := FALSE; exit;
    or -- one task ends:
        accept TASK_DONE;
        task_count := task_count + 1;
        if task_count = TaskNumber
then alternatesuccess:=TRUE;
   exit; end if;

or -- time constraint
delay (this_alt_deadline-CLOCK);
abort T11, T12, T13;
   alternatesuccess:=FALSE; exit;
end select;
end loop;

exception
   when others => abort T11, T12, T13;
   alternatesuccess:=FALSE;
end WATCHDOG;
begin  --MaxRow body
   null;
end MaxRow;

procedure MaxSeq(A: in big_matrix; elem: out elemtype;
            alternatesuccess: out BOOLEAN) is

  task T21;
  task WATCHDOG is
      entry TASK_FAILURE;
      entry TASK_DONE;
  end WATCHDOG;
  task body T21 is
      new_max: elemtype;
      begin
          new_max:=A(rows_interval'first,
               columns_interval'first);
          for K in rows_interval loop
              for J in columns_interval loop
                 if new_max<A(K,J) then new_max:=A(K,J); end if;
              end loop;
          end loop;
          elem:=new_max;
          WATCHDOG.TASK_DONE;
  end T21;
  task body WATCHDOG is
      TaskNumber: constant:=1;
      TimeConstr: constant:=30.0; -- for the second alt
      task_count: INTEGER:=0;
      this_alt_deadline: TIME;
      begin


this_alt_deadline:=CLOCK+TimeConstr;

loop
select
  accept TASK_FAILURE;
  abort T21;
  alternatesuccess:=FALSE; exit;
or  -- one task ends:
  accept TASK_DONE;
  task_count:=task_count+1;
  if task_count=TaskNumber
     then alternatesuccess:=TRUE;
        exit; end if;
or  -- time constraint
  delay (this_alt_deadline-CLOCK);
  abort T21;
  alternatesuccess:=FALSE; exit;
end select;
end loop;
exception
  when others => abort T21;
  alternatesuccess:=FALSE;
end WATCHDOG;
begin  -- MaxSeq body
  null;
end MaxSeq;
function TestResult(elem: in elemtype) return BOOLEAN is
  minlimit: constant:= -10000;
  maxlimit: constant:= 10000;
begin
  return elem>minlimit and elem<maxlimit;
end TestResult;
end MaxElem;
Captions to illustrations:

Fig. 1. Conversations

Fig. 2. The execution structure of a concurrent recovery block, or ft-procedure
Fig. 1. Conversations
Fig. 2. The execution structure of a concurrent recovery block, or ft-procedure