THE CONVERSATION DEADLOCK PROBLEM IN CLIENT-SERVER MODEL

Andrea Clematis[•] and Vittoria Gianuzzi[⋆]

[•] Istituto per la Matematica Applicata del C.N.R.
Via L.B. Alberti, 4 - 16132 Genova, Italy

[⋆] Dipartimento di Matematica dell'Università
Via L.B. Alberti, 4 - 16132 Genova, Italy

Different problems have to be solved for an effective use of conversations as a mean to improve the overall reliabilty of a concurrent program. In this paper we address the so called *Deserter Process Problem*. We investigate the causes which generate deserter processes, distinguishing between the case in which a process does not enter a conversation and the case in which a process does not reach the acceptance test line. The first case is related with a particular type of deadlocks. A model of concurrent programs with conversations, and conditions, which are useful to detect deadlocks in these programs, are presented.

1. INTRODUCTION

Conversations have been proposed as a linguistic construct to provide concurrent programs with backward error recovery facilities[1], avoiding the domino effect. We briefly describe the semantic of such construct. The processes partecipating in the conversation enter it, not necessarily synchronously, establish a recovery line and cooperate in error detection by executing a synchronous acceptance test. If any of the processes fails its acceptance test, recovery is achieved by rolling back all the processes in the conversation to their respective recovery points. Processes participating in a conversation are prevented from communicating with processes not participating in it, to avoid information smuggling (figure 1).

Processes

P1 P2 P3

recovery line

side wall

acceptance test line

figure 1. The conversation structure

Even if various studies exist about applications of conversations, some open problems still remain to be solved to allow their widespread and effective use. For instance, conversations cannot solve all kinds of error arising during a computation: in fact, certains errors are revealed by the acceptance test, others result in a blocking of the conversation itself. This latter problem, known as the *deserter process problem*, is present in almost all the conversation implementation schemes proposed in the literature. In spite of its importance, only few papers deal explicitely with it, and no one performs a through analysis enlightening the reasons of its rising.

Here, we present a comprehensive analysis of the desertion problem, which investigates its causes and indicates how to handle it.

This paper can be intended as the completion of the work[2] presented by the authors at the same Conference, dealing with problems arising from an effective use of conversations in typical concurrent applications. In that work, we referred to an implementation[3] of conversation in Ada* and discussed some extentions to the conversation construct, in order to improve its usefulness in applications following the client-server model. Thus, we advise the reader to refer to those papers for a deeper discussion about the mechanisms proposed in the literature for implementing conversations.

2. THE DESERTER PROCESS PROBLEM

In the client-server model, a distinction is made between active processes, or clients, which act in a deterministic way, and passive processes, or servers, which act in non-deterministic way, since they cannot decide deterministically which request to answer (among those which can be satisfied). In such a model, a conversation occurs among clients and servers. The first ones decide deterministically to enter it, while the second ones are asked to participate in it, and decide in a non-deterministic way which conversation to join. Thus, the system can deadlock if one (or more) process will never join a conversation. This can be due to two main reasons.

First, the final synchronization required by the conversation may introduce a particular form of contention on resources, that is, some conversations cannot evolve because of a circular wait for servers. This case shall not be considered as a real program error because it is the consequence of an incorrect scheduling of the underlying operating system. In such a case, the deadlock must not be handled inside the conversation, by means of restart and retry of another alternative. Indeed, we may have a degraded response, at least in time, also in presence of a basically correct program, and remark that the restarting of an

---

* Ada is a registered trademark of the U.S. Government.

alternate try block cannot guarantee the breaking of a cyclic restoration of deadlock situations.

The implementation proposed by the authors[1] requires the presence of an additional server process, that is the *Conversation Manager* which locks the servers needed in a conversation, allows the synchronization among processes and the exclusion among conversations. This process could also provide mechanisms for avoiding resource deadlock, for instance, by acquiring the lock of the servers following some order imposed on the same servers.

Second, the decision of an active process, which is expected to participate in a conversation, to follow an alternative path, may introduce a deadlock in the form of an endless wait. In this case, deadlock cannot be avoided at run time, since missing participation in a conversation can be caused by a fault of the process. Thus, to deal with deadlock, it is necessary to use an adequate computational model and the related suitable tools to perform static analysis of concurrent programs.

3. A COMPUTATIONAL MODEL OF ADA PROGRM WITH CONVERSATIONS

We can derive a computational model by suitably modifying the model proposed by Taylor[4] which deals with verification problems arising in concurrent software programming. Taylor aims at constructing a tool to ensure the designer of a concurrent program (in particular Ada programs) that no undesiderable parallelism is present, or that her/his system is free from deadlock.

In that paper, an Ada program is represented as a set of directed rooted flowgraphs $(G_1...G_U)$, where each $G_b = (N_b, A_b, r_b)$ correspond to an uniquely identified program unit of a task. A node (called *state node*) $c_i \in N_b$ represents any of the following statements: entry call, accept, select, select-else, delay, abort, task begin, task end, subprogram begin, subprogram end, subprogram call, block begin and block end. They represent the tasking activities which are relevant for the analysis to be performed. Each arc $a_i \in A_b$ represents possible control flow, and $r_i \in N_b$ is the root of the flowgraph.

From our point of view, the relevant tasking activities are at a higher level than those in Taylor's model. Rather than in the single communication, we are interested in the relationships among conversation sections. Each conversation can be considered as a single communication involving a set of conversation sections and a set of resources, thus becoming the only synchronization points in the program.

We have to modify also the representative set of the nodes. In fact, active tasks can contain specialized blocks, the *conversation sections*, that constitute the local parts of a conversation. a rendez vous, can be performed only within a conversation.

Thus, in this modified representation of an Ada program, a state node for an active task can represent one of the following items: *abort, task begin, task end, subprogram begin, subprogram end, subprogram call, block begin, block end, conversation section*; while a task representing a reusable resource is represented by the following flowgraph:

$$\text{begin} \;\rightarrow\; \text{body} \;\rightarrow \text{end}$$

These graphs are not sufficient to represent information such as the set of conversations and, for each conversation, the set of conversation sections and resources composing it.

To simplify this presentation, we can follow the example of figure 2, in which a program is represented by means of 6 flowgraphs (the nodes labeled as Si represent the conversation sections, whereas R1 and R2 are tasks considered as resources).

main	**subr**	**task R1**
	declare tasks	
begin	T1, T2, R1 and R2	begin
$\downarrow$		$\downarrow$
subr;	begin	R1 body
$\downarrow$	$\downarrow$	$\downarrow$
end	S1	end
	$\downarrow$	
	S2	
	$\downarrow$	
	end	

task T1	**task T2**	**task R2**
begin		begin
$\downarrow$	begin	$\downarrow$
S3	$\downarrow$	R2 body
$\downarrow$	S5	$\downarrow$
S4	$\downarrow$	end
$\downarrow$	end	
end		

The conversations are so defined:

C1 among S3, R1, R2
C2 among S1, S5, R1, R2
C3 among S2, S4, R1

figure 2: an Ada program constitutes of 6 program units and 3 conversations.

We observe that non determinism is maintained, because of the possibility of a resource to select what conversation to enter.

If at any moment we observe the evolution of the program, we see that each task appears to be in one of the above listed states. This set of states can thus be represented by means of a N-tuple, $X = (c_1, c_2, ..., c_N)$ called *concurrency state*, where N is the number of active tasks. Each state c_i may assume one of the values stated above for our case, plus the value *inactive*.

Hereafter, resources whall not be represented in our scheme, since they are not responsible of *structural deadlocks*, that is, unrecoverable wait-forevers, due to a software fault. However, they will be reconsidered later in the analysis, to state rules allowing the user to test the complete deadlock-freeness of his/her program. For lack of space, we do not describe the algorithm to be applied in order to find all the different concurrency states. In a few words, starting from the initial states of each task, we obtain all the other states by following the control flow of the task or by applying any possible synchronization due to the completion of a conversation.

For the example above, we have the concurrency states, with $N = 3$, presented in figure 3 (the set is not complete).

States	(Main task, task T1, task T2)
X_0	(begin main, *inactive*, *inactive*,)
X_1	(subr, *inactive*, *inactive*)
X_2	(begin subr, begin T1, begin T2)
X_3	(S1, S3, S5)
X_4	(S1, S4, S5)
X_5	(S2, S3, S5)
X_6	(S2, S4, S5)
$\vdots$	$\vdots$

figure 3: concurrency states

We can also represent the *graph of the histories* of the program which can be obtained by linking the states togheter by means of edges representing the possible state transitions. Since we are interested in the conversation analysis, we will only consider those states which are relevant, that is, the states containing conversation sections. Moreover, when a transition is due to the completion of a conversation, the corresponding edge is labeled with the identifier of such a transition. For the above example we have the state transition graph of figure 4.

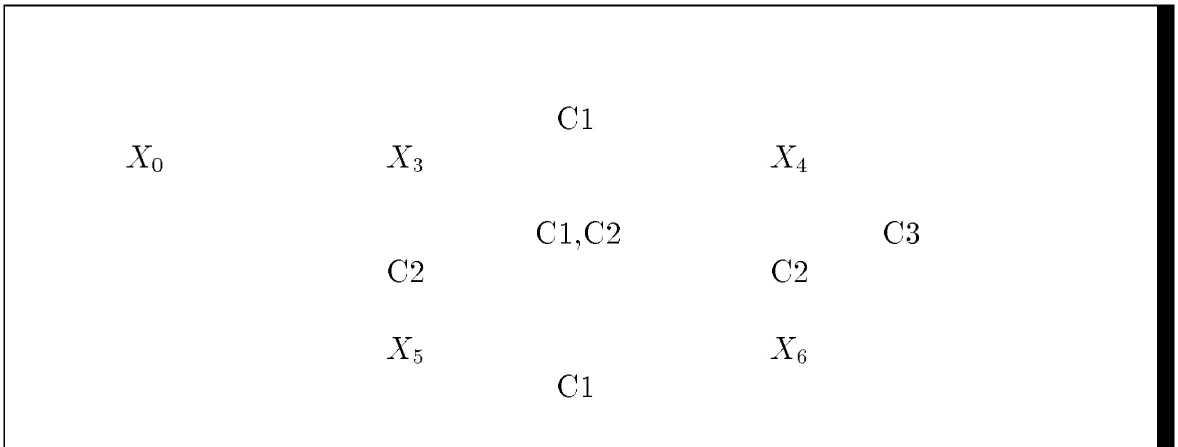

figure 4: a state transition graph

4. DEADLOCK ANALYSIS

Starting from this computational model of an Ada program with conversations, we obtain the information needed to analyze the deadlock problem by considering, for every concurrency state, called *Conversation Execution System (CES)*. A CES is a multigraph containing both directed and undirected edges, derived from a concurrency state $X = (c_1, ..., c_N)$ in this way:

- a node is a conversation C such that at least one of its sections is present in X;
- a directed edge (C2,C1) exists if there exist $c_i,\ c_j,\ c_k$ such that $c_i \in$ C1; $c_j,\ c_k \in$ C2 (C1 $\neq$ C2), $c_i,\ c_j \in X$ and a path from c_i to c_k exists in the initial flowgraphs. This edge means that the conversation C2 needs for its completion the section c_k which can be viewed at as a resource held by C1;
- an undirected edge {C1,C2,R} exists, if the resource R is required by both the conversations C1 and C2. Such an edge is labeled with the resource identifier.

Again on the above example, we obtain, for each concurrecy state containing some conversation sections, the $CESs$ presented in figure 5:

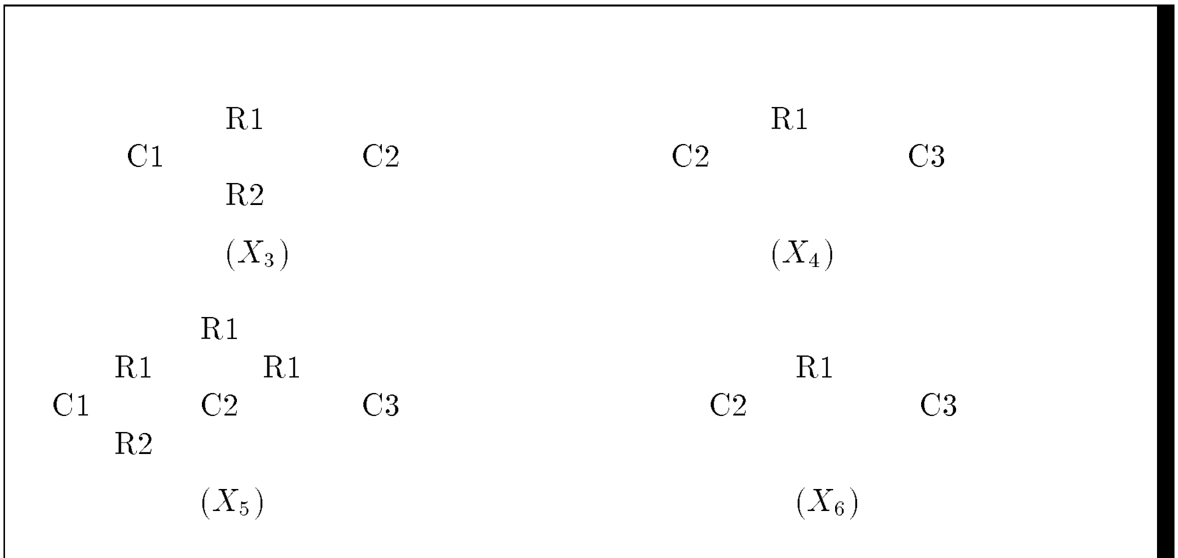

figure 5: Conversation Execution Systems

An undirected edge represents a dependency which can be explicited only at run-time. The relation

{C2,C3,R1} in the CES related to the state X_4 can become (C2,C3) if resource R1 is acquired by C3, so that C2 is waiting for it from C3, and vice versa.

As an example, let us see in more details state X_4 = (S1, S4, S5). Both conversations C2 and C3 are represented by a section, so resource R1 can select what conversation enter between them. Depending on its choice, the system could enter a deadlock or not. If R1 enters conversation C2, such a conversation complete, and R1 can participate in another conversation. This case corresponds to the transition labeled C2 between X_4 and X_6 on the graph of the histories. On the contrary, if R1 enters conversation C3, we shall obtain a deadlock. In fact, conversation C3 needs for its completion also the section S2 which cannot be executed until the end of section S1. But section S1 needs the resource R1 to exit conversation C2. This leads to a circular waiting situation, that is, deadlock. These possibilities are both represented by a unique CES, ascribing a direction to the undirected edge {C2,C3,R1}. We can also observe that the deadlock could not be detected run-time without the knowledge of the relations among sections, tasks and conversations.

Thus, we can state the following:

Theorem: Given a Conversation Execution System the following facts hold:

1. a structural deadlock arises if there exists a cycle in the CES, obtained by following only directed edges. A structural deadlock is a program error for which no legal schedule exists.
2. the system is deadlock-free if and only if no cycles exist among the conversations, obtained by following both directed and undirected edges.
3. a Conversation Execution System which satisfies constraint 1 but not constraint 2 admits both legal schedules, which lead to the execution termination, and illegal ones, which lead to deadlock situations.

Circular waiting are not, however, the only case in which a deadlock can arise. Let us consider, for example, the state X_7 = (end, end, S5), not listed in figure 4, but which deems possible, considering the initial flowgraphs. It is indeed a deadlock state, even if it is not due to a cycle in the corresponding CES

but to an infinite wait. In fact conversation C2 cannot complete because the other needed section, that is S1, never executes. To be sure that the state X_7 will never actually occur, it is necessary to analyze the boolean expressions which cause the choice between, for example, the end of the task or the iteration of another section. These controls must be performed for all the concurrency states from which, in the concurrency history graph, it is not possible to reach the terminal states, that is, the ones containing the markers *end* or *inactive* for all the tasks.

5. CONCLUSIONS

In this paper we propose an analysis of deadlock problem in conversation.

The theorem we present in the last section can be used to design an algorithm and can be applied in different ways. It can be used to test the conversation definitions for avoiding structural deadlocks or to prove the deadlock freeness of a program. If any state is found to be deadlock prone, a deadlock avoidance algorithm can be applied (for example an initial synchronization among conversation sections and a suitable protocol for the resource locking) restricted to that state. In such a way it is possible to avoid the performance decrease consequent to the application of this kind of algorithm to all conversations.

Some consideration are necessary about the analysis of concurrent programs we propose in order to collect information necessary to avoid deadlocks. With respect to the methodology described by Taylor[4], we observe that the modularization induced by the conversations on the whole In fact, the first step is the analysis with respect to the relationships among conversations, while the possible second step, in order to complete the verification of the program, is the analysis of the concurrency inside each conversation, following Taylor's scheme.

As like all the other tools performing concurrency analysis, also our method has an exponential time. However, the hierarchical approach results in an improvement in the efficiency of the method, since the number of tasks and of the synchronizations to be considerd for each phase decreases with respect to the analysis performed on the whole system at a time.

The methodology we proposed is oriented to Ada programming. However, a similar approach can be used also starting from different formalisms. For instance, the class of distributed systems which can be expressed as a set of communicating sequential processes, can take advantage in the concurrency analysis, of the Petri nets formalism. An example of Petri net use can be found the work of Tyrrell[5], where a method is presented for the "a priori" definition of the conversation boundaries in an occam program, starting from its reachability tree.

REFERENCES

[1] Randell B., "System structure for software fault tolerance", *IEEE Trans. Software Eng.,* Vol.SE-1, pp.220-232, 1975.

[2] Clematis A. and Gianuzzi V., "Software Fault Tolerance in Concurrent Ada Programs", in *Proc. Euromicro 91 Conference*, Vienna, Sept.2-5, 1991.

[3] Clematis A. and Gianuzzi V., "A Conversation Structure for Remote Procedure Call Oriented Languages", in *Proc. Fault Tolerant Computing Systems*, Springer Verlag, pp. 163-173, 1989.

[4] Taylor R.N., "A General-purpose Algorithm for Analyzing Concurrent Programs", *Comm. ACM*, Vol.26, No.5, 1983.

[5] Tyrrell A.M. and Holding D.J., "Design of reliable software in distributed systems using the conversation scheme", IEEE Trans. Softw. Eng., Vol.SE-12, no.9, pp.921-927, 1986.