Deadlock Avoidance, Non-Linearity and Games

Raymond Devillers
Département d’Informatique, CP212
Université Libre de Bruxelles
B-1050 Bruxelles, Belgium
rdevil@ulb.ac.be

Deadlocks lead to interesting questions and techniques, more subtle than usually thought; that led to some mistakes or difficulties that we shall exhibit. In particular, we shall show how the introduction of an asymmetric game allows to correct a flaw in the deadlock avoidance field.

Deadlocks, detection, prevention, avoidance, game interpretation

1. INTRODUCTION

Deadlocks occurred in computer science in the sixties, when manufacturers started to develop operating systems managing more than one process simultaneously. Initially, the goal was to get a better usage of the resources, especially the CPU, through a background task or through a few independent jobs; later occurred the possibility to have cooperating processes. Anyway, in all cases, problems occurred when it came to share non-preemptible resources. At that time, the most striking example of such resources was probably the tape drives.

Let us assume that there are three drives\(^1\), like in the figure 1, one of them being reserved by the OS for its ancillary tasks, a foreground process F and a background one B. Let us also assume that F requests a drive (and a tape to be mounted on it): since two of them are available, the OS may grant one. Now, if F is suspended (waiting for instance for an action of the user), B will be activated. If B also requests a drive, since one is still available, the OS may grant it. If F becomes ready again, it will receive the control; now, if F requests a second drive to terminate its job, it will be suspended until the drive attributed to B is released. B is then resumed, but if it requests itself a second drive to terminate its job, it will be suspended until the drive attributed to F is released. A deadlock (also called deadly embrace, or interlock, ...) has now occurred: F and B block each other in a way that will not come to an end. This is of course a very nasty situation since F and B are blocked forever, the two attributed drives are lost for the system, and if the OS is not devised to allow for more than one foreground and one background processes, the whole system is dead. This kind of situation is represented in figure 2.

Various kinds of techniques have been devised in order to cope with this type of situation (and much more complicated ones of the same kind), that we will evoke in the next sections. Those techniques were initially meant to be applied by the OS, which anyway is usually in charge of managing the resources of the system. Here, we considered independent processes, interacting

\(^1\)Of course, the problem we are about to describe could also occur with more modern resources, like USB slots or CD-ROM drives.
implicitly through the physical resources of the system, but soon it was discovered that it was possible to develop cooperating processes, interacting more explicitly through logical resources only known by the processes themselves, like the access to (possibly imbricated) critical sections, and that those processes could be run on separate platforms, each one with its own operating system, interacting through a network. The problem then tended to shift to one or more (human and/or automated) (centralised or distributed) resource managers. We shall not elaborate more on who will eventually manage the problem, but on how this may be done.

2. WAIT AND SEE

A first approach to the deadlock problem is to let the system evolve without any constraint, except that, from time to time or when a process requests resources which are not available, one checks for the apparition of a deadlock and then one takes drastic actions.

Deadlock detection is not a simple problem in general systems; a detailed analysis has been conducted in Holt (1971, 1972), where one allows to request simultaneously many resources, resources are grouped by types, and besides reusable resources (which are recovered by the system when the process ends, unless they are released before, like the tape drives above) one also manages consumable resources (which may be created and absorbed, like semaphores or slots in a queue). Holt defines a reduction operation, which devises the more favourable evolution which may occur before a process ends, and searches for a complete reduction sequence. In general, one may need to perform some form of backtracking so that the check is intrinsically exponential. Fortunately, when there are only reusable resources, backtracking is not necessary and the check is in $O(P^2T)$, where $P$ is the number of processes and $T$ the number of resource types. Moreover, Holt noticed that, with adequate data structures, one may avoid to uselessly consider a process in the search of a complete reduction sequence. The procedure (to be performed when one wants to know if a deadlock emerged) looks like the following:

1. for each resource type, one creates a list of the processes, sorted by increasing number of requested resources of that type, keeping only the processes which are presently blocked due to this resource type, hence which request more than the number of resources of that type which are presently free; for each process, one keeps a variable counting the number of lists it is in (hence the level of blocking); and one creates a set of the processes with a null counter (which are not presently blocked).

2. while the set is not empty, extract one process in it, free (virtually) all its previously obtained resources, adjust the sorted lists, the counters and the set;

3. if all the lists are empty (or all the counters are null), no deadlock occurred; otherwise, the processes with a nonnull counter are involved in a deadlock.

Since one only progresses in the structures, Holt claims that this procedure is in $O(PT)$, hence is linear in the number of processes. This is essentially true for the (main) step 2; step 3 is still quicker; but step 1 is more problematic. Indeed, one has to create sorted lists, each one having a size which may reach $P$. As a consequence, with a naive sorting procedure in $O(P^2)$, nothing is gained. Better procedures (like the quicksort or the heapsort) are in $O(P \log P)$ and may thus be a bit more interesting (except that for small $P$ this may be slower than a naive method). But since, for each list, the number of possible sorting keys is the number $I$ of instances in the corresponding type of resources, one may also use sorts in $O(P + I)$, like the counting sort, so that, if we assume that all types have the same $I$, we finally get a complexity in $O((P + I)T)$, hence indeed linear in the number of processes. However, we may also observe that the procedure has only to consider processes which already acquired some resources (otherwise they may not hamper the other processes and cause a deadlock, at least if they are not autoblocking, i.e., need more resources of some type than they are available in the whole system,
which always leads to a problem but is easy to check). Hence $P \leq TI$, which is good news, but this also means that in order to increase infinitely $P$, one also has to increase correspondingly $TI$, so that the complexity may no longer be linear.

Now, if a deadlock occurred, one still has to get out of it. This may be done for instance by killing one or more processes involved in the deadlock and restart them later. Another possibility is to apply some preemption on one or more resources causing the deadlock; indeed, it may happen that some resource types are in fact preemptible but the cost of the preemption is so high that one prefers to manage them as non-preemptible, until a problem occurs (like a deadlock) needing to finally apply the preemption. This is the case for tape drives for instance, since it may be possible to suspend the user process, record the exact position of the tape mounted on the drive, rewind it and unmount it, and when we (the OS or the resource manager) want to resume the process, one has to remount the correct tape on the drive and unwind it on the correct position.

Coffman, Elphick and Shoshani (1971) devised branch and bound algorithms in order to find the optimal way of getting out of a deadlock situation, but they were not often used because they are intrinsically exponential, so that it may be possible to lose more time in searching for the optimal solution than to use a simple not too bad (but non-optimal) heuristic argument. Moreover, it is necessary to have at hand a precise cost associated to the killing of a process or to the unexpected preemption, while we often only have a vague estimation of it. Notice that it may also happen that it is impossible to restart a process after killing it, if it has modified a file and if it is not possible to get a correct execution from its present shape. But this is the clue that the application was not correctly developed, since it may always happen that the system crashes at any point, leading the process exactly in the very same situation. This is why transaction systems, with commits, checkponts and unwindings have been developed for data base managements and similar applications.

Wait and see strategies are interesting if resources are numerous enough with respect to the number of concurrent processes and their needs, so that deadlocks are rare enough, and if the cost of a recovery is not too high. Otherwise it may be preferable to apply a prevention or an avoidance strategy\(^2\).

\(^2\)Curiously, in the literature, a terminology switch occurred in the seventies: avoidance strategies became prevention ones, and conversely.

3. PREVENTION

Prevention strategies amount to impose restrictions on the way processes may request their resources, in such a way that a deadlock never can occur. Havender made a thorough study of those strategies in Havender (1968), at least when there are only reusable non-preemptible resources since consumable resources are more delicate to handle and specific ad hoc strategies must be developed for each case. Those strategies essentially consist in preventing the formation of a cycle of requests/attrition, which is known to be a necessary condition to have a deadlock.

For instance, one may impose each process asks for all the resources it may need in a single request, unless all the previously acquired resources have been released meanwhile. This is a bit constraining however, since it may happen that some of those resources requested together will only be used at the very end of the process work, and they will so be neutralised uselessly (but for the prevention of deadlocks) for a possibly long time.

A more efficient solution is thus the so called Havender’s hierarchical strategy (also developed in Havender (1968)), which amounts to order the resource types (or groups of them) and to strictly follow this order in the requests, i.e., a process already having some resources may only request additional resources which are strictly further in the order. The role of the resource manager is then to enforce respecting this rule (but it often remains implicit, assuming the users, through their processes, will always follow the specified order, which by the way they fixed themselves). Of course, as usual, there is a price to pay to prevent deadlocks in this way, but it is usually lower than for the the previous kind of strategy. For instance, if a process needs two resources of the same type, it will need to request them simultaneously, even if it only needs the second one at the very end of its job, thus neutralising again uselessly this resource during a possibly long time.

Another kind of strategies falling in this category is to fix a model of behaviour of the processes and to perform some kind of model checking to verify that no deadlock can ever occur.

4. AVOIDANCE: THE BANKER’S ALGORITHM

Avoidance strategies assume that some information is known in advance about the future needs of the various processes in terms of non-preemptible resources. Then, the resource manager determines unsafe situations, i.e., dangerous ones, from which a deadlock may occur without being able to avoid
it. When a process requests some resources, even if they are available, if this would lead to an unsafe situation, the request will not be granted, in order to navigate through safe situations only.

The most known case falling in this category is the maximum need model, where each process warns the resource manager (when it does not hold any resource yet) about the maximum amount it will need for each type of (reusable) resource. An algorithm for detecting unsafe situations in this case was published in Dijkstra (1968) when there is a single resource type, and for the general case in Habermann (1967, 1969), under the nickname of the banker’s algorithm. This name arises from an analogy with a banker (the OS or resource manager) granting loans (resources) to clients (processes) in such a way that it is sure each client will be able to perform his investments and himself will recover all his money (single resource type), but curiously without any interest! The algorithm essentially assumes that each process, from the current state to be checked, immediately asks to reach the maximum amount it announced, and checks if we reach a deadlock situation (with the aid of a nonextendable sequence of reductions, like in Holt’s algorithm); as such, its complexity is in $O(P^2T)$.

However, Habermann made three interesting remarks on Dijkstra’s analysis. First, the “worstcase” considered by Dijkstra is a bit “too worst”. Indeed, if a process is suspended on a resource request in the present situation, it is not licit to assume that it immediately asks to reach its maximum need: it should be allowed to obtain its current request first, and only then it may ask for its maximum. Fortunately, Habermann showed that this does not modify the set of (un)safe states. Next, he noticed that it was possible to speed up the procedure, in a way different but similar to Holt’s one (and previously to the latter). Finally, he noticed that the banker’s algorithm does not address the right problem. Indeed, instead of checking if some situation is safe or not, one should check if, starting from a safe situation, when some process $p$ requests some resources and they are granted to it, does the system stay in a safe situation? He then showed that, to solve this redefined problem, one does not need to find a complete sequence of reduction, but a partial sequence ending with the considered process $p$. Hence, one may first check if the system, after the tentative grant, allows to directly reduce the process $p$; if this is true (which will often be the case if the system is not overloaded), one may immediately stop and the cost of the check is minimal; otherwise, one may search for a sequence of length 2, then 3, ... ending with $p$.

With some chance, the cost of the check will not be very high.

Of course, there is also a price to pay with this kind of strategies; indeed, it may happen that the resource manager refuses to grant a request while the resources are available, because there is a (possibly small) chance that a deadlock will occur in the future and we do not want to take the feeblest risk; but it may well happen that even if the resources are granted, that deadlock never happens in fact.

5. AVOIDANCE: THE LINEAR MODEL

The kind of information yielded by the maximal need model is a bit poor however. Often, one will have more information about the future needs, and one may hope this will lead to deem safe many situations which would have been considered unsafe by the banker’s algorithm. In this respect, Hebalkar (1970, 1971) considered a very informative “linear model”, where each process specifies the exact sequence of steps it will follow, and the (maximal) need of each of those steps. One will however assume that, when going from one step to the next one, each process will either release resources, or claim for more, but not both (otherwise, it is not clear if we first perform the releases, then the new grants, or the other way round: in any case it is always possible to introduce one or more intermediate steps (without associated computations) specifying how the transition should be made concerning the resource allocation).

Then, a situation is safe if and only if there is a global evolution such that its projection on each process yields the linear history of the process from its current state, and one only visits realizable states, i.e., for each point of the global evolution, the sum of the needs of the last state of each process is lower or equal to the total number of resources known by the system (i.e., its resource manager).

Hebalkar showed that, unfortunately, when there are at least two resource types, the problem of determining if a state is safe or not is intrinsically exponential and needs some form of backtracking. That means that only small systems, with a limited number of processes and not too long histories, will be manageable. On the contrary, if there is a single resource type, the problem is linear in the size of the system, the difference arising from the fact that the order on natural numbers is total while the componentwise order on vectors of naturals is only partial. The idea of the algorithm for a single resource type is schematised in Figure 3. One looks at the successive needs of some process. From the present situation, one looks first at the first local minimum which is lower than the present need
(indicated in the figure by the first upward arrow), and one determines the global maximum (indicated in the figure by the first downward arrow) between the present situation and the found local minimum: if it there are enough free resources to reach this maximum, one may (virtually) progress to the local minimum (which represents a “better” situation) and resume the procedure (indicated in the figure by the subsequent upward and downward arrows). If it is not possible to reach the maximum point, one has to look at another process. If one reaches a point where all the processes are finished, that means that the initial situation was safe. If, on the contrary, one gets stuck because no process allows to progress (while some are unfinished), there is a deadlock.

6. AVOIDANCE: THE BRANCHING MODEL

Then, Hebalkar considered the case where each process specifies a future history which is only linear or finitely branching\(^3\). His idea is that, if one considers all the (finitely many) combinations of linear histories for each process compatible with the specified branching one, and if each of them is recognised as safe by his previous algorithm, then the situation is safe.

Of course, this means checking exponentially many combinations with an exponential algorithm (if there are many resource types), but this is not the most severe objection to this idea. The main one is that it is wrong.

The origin of the problem may be exhibited on a system with just two processes, one of them being linear and the other one having a single binary choice, as shown in Figure 4.

The problem is that it may happen that, when we combine the linear history of the first process with the high history of the second process, Hebalkar’s algorithm says the system is safe, but one needs to first advance in the first process in order to be able to conduct the system to its complete termination. And when we combine the linear history of the first process with the low history of the second process, Hebalkar’s algorithm still says the system is safe, but one needs to first advance in the second process in order to be able to conduct the system to its complete termination. As a consequence, in that case, initially, since we don’t know in advance which branch will be chosen by the second process, we are unable to know if we should first let the first process to progress, or the second one. As a consequence, we should not consider the system is safe since we do not know how to behave to lead it to its end. And this is indeed a situation that can occur, as exhibited by the concrete example in Figure 5, with 4 binary resources (i.e., there is no true resource types: resources are all individual and non-interchangeable).

From state \((s_1, s_2)\), if \(P_2\) follows the high linear history \(s_2^3, s_2^4\), while \(P_1\) follows its unique linear history \(s_1^4\), \(s_1^5\), \(s_1^6\), the sequence (where we denote by end, the fact that process \(i\) reached its end) \(s_2^3, s_2^4, \text{end}_2, s_2^5, s_1^6, \text{end}_1\) is realizable, but there is no complete realizable sequence starting with \(s_1^5\). Indeed, if \(P_1\) progresses first, then it has to continue until reaching the state \((s_1^5, s_2^4)\), then \(P_2\) has to progress, reaching the state \((s_1^5, s_2^2)\), but from that point neither \(P_1\) nor \(P_2\) may progress and the system is stuck. And if \(P_2\) follows the low linear history \(s_2^3\),

\[P_1: s_1^1 \rightarrow s_1^2 \rightarrow s_1^3 \rightarrow s_1^4 \rightarrow s_1^5\]

\[P_2: s_2^1 \rightarrow s_2^2 \rightarrow s_2^3 \rightarrow s_2^4\]

Figure 4: Why Hebalkar’s idea to manage branching histories doesn’t work.

\[P_1: s_1^1(0,0,0,1) \rightarrow s_1^2(0,1,0,1) \rightarrow s_1^3(0,1,0,0) \rightarrow s_1^4(0,1,1,0)\]

\[s_1^5(0,0,1,0) \rightarrow s_1^6(1,0,1,0)\]

\[s_2^1(1,1,1,0)\]

\[P_2: s_2^1(1,0,0,0) \rightarrow s_2^2(1,1,0,0)\]

\[s_2^3(1,1,0,1)\]

Figure 5: A concrete counter-example to Hebalkar’s idea.

\(^3\)He also considered the case of loops, and reduced this case to a finite set of branching systems.
Figure 6: The flowchart model of a process.

$s_2^2$, the sequence $s_1^1$, $s_1^2$, $s_1^3$, $s_2^2$, $s_2^3$, end_2, $s_0^1$, end_1$ is realizable, but there is no complete realizable sequence starting with $s_2^2$. Indeed, if we start with $s_2^2$, we reach the state $(s_1^1, s_2^2)$, from which neither $P_1$ nor $P_2$ may progress and the system is stuck.

7. AVOIDANCE: THE FLOWCHART MODEL

In order to solve this problem, we shall first generalise it. We shall assume that each process specifies a priori its future needs in the form a flowchart representing the set of steps, for which we know the resources they need, and the possible transitions between them. Again we shall assume that steps connected by a transition have comparable needs (either it is a release, a request or do not use any resource).

We shall also assume in a row, hence, for the game, there will be states $\sum_{i=1}^{P} \vec{n}(\varepsilon_i) \leq \vec{r}$, i.e., we only consider realizable states.

P processes in the system, where $\sum_{i=1}^{P} \vec{n}(\varepsilon_i) \leq \vec{r}$, hence, for the game, there will be states $\varepsilon$, where the system is in state $\varepsilon$ and the coalition of processes has the lead.

The moves of each player are as follows:

1. for the resource manager:

- $\varepsilon \rightarrow \varepsilon'$ if $\varepsilon$ and $\varepsilon'$ are realizable, $\varepsilon_i = (s_k^j, s_l^m)$ and $\varepsilon_i' = s_k^i$ for one or more $i$'s, and $\varepsilon_i = \varepsilon_i'$ for the other ones, i.e., one or more requests are granted;
• \( \emptyset \rightarrow \emptyset \) if some processes are in a working state in \( \varepsilon \), i.e., \( \exists i : \varepsilon_i = s_i^j \), or all processes are finished, i.e., \( \forall i : \varepsilon_i = f_i \); hence, waiting moves are not allowed if some processes are not finished and all non-finished processes make a request; indeed, in that case, there is no interest to wait: the resource manager has to grant some requests, or recognise it lost;

2. for the processes:
   • \( \varepsilon \rightarrow \emptyset \) if one or more processes make a request, i.e., \( \varepsilon_i = s_i^j \) and \( \varepsilon'_i = (s_i^j, s_i^k) \) and for all the other processes \( \varepsilon_i = f_i \);
   • \( \emptyset \rightarrow \emptyset \) unless all the processes are finished, i.e., \( \forall i : \varepsilon_i = f_i \); indeed in that case, the processes recognise they have lost and no waiting move may help.

It may be observed that we have a kind of “Nim game”: when a player can make no move, he has lost!

Now, we may apply the following algorithm to construct a set of unsafe situations:

1. let \( \mathcal{U} \) be the set of completely blocked states\(^4\) for the resource manager: \( \mathcal{U} = \{ \emptyset | \exists i, j, k : \varepsilon_i = s_i^j, \varepsilon'_i = (s_i^j, s_i^k) \text{ and all the other processes } \varepsilon_i = f_i \}; \) this is easy to construct when we build the graph of the game;

2. while there exists \( \emptyset \notin \mathcal{U}, \emptyset \in \mathcal{U} : \emptyset \rightarrow \emptyset \), we add \( \emptyset \) to \( \mathcal{U} \) or \( \emptyset \notin \mathcal{U}, \forall \emptyset' : \emptyset \rightarrow \emptyset' \Rightarrow \emptyset' \in \mathcal{U} \), we add \( \emptyset \) to \( \mathcal{U} \).

It may be observed that this procedure terminates since \( \mathcal{U} \) increases at each step of the loop and the graph of the game is finite. Constantly, \( \mathcal{U} \) contains only unsafe configurations. Indeed it is so initially, by construction (it then contains the situations which are winning for the coalition of processes), and we enrich it with situations where either the processes have the lead and there is a move leading to a situation we already know as unsafe, or the resource manager has the lead and whatever it does, this leads to a situation we already know as unsafe. But we also have the following crucial result.

**Theorem 1** When the previous procedure terminates, i.e., when \( \mathcal{U} \) is stabilised, it is not only a set of unsafe situations, but the set of all of them.

---

\(^4\)We could as well start from \( \mathcal{U} = \emptyset \), since then the completely blocked states would be incorporated during the first iterations of while loop.

---

**Proof 1** Let \( S \) be the complement of \( \mathcal{U} \) when the procedure terminates, i.e., the set of all the configurations of the game that were not incorporated in \( \mathcal{U} \). By construction, if \( \emptyset \in S \), there must be some \( \varepsilon' \in S \), too, such that \( \emptyset \rightarrow \varepsilon' \). Indeed, otherwise \( \emptyset \) could have been added to \( \mathcal{U} \) and would not be in \( S \). And if \( \emptyset \in S \), then \( \forall \emptyset \rightarrow \emptyset ', \emptyset ' \in S \), for the very same reason. Hence, if we are in \( S \), the resource manager can manage in order to stay in it, and the processes may not force leaving it. Moreover, from the hypothesis we made about the behaviour of the processes, they may not induce an infinite history, hence the game should terminate at some point. But the only terminating point of the game which is not in \( \mathcal{U} \) is the winning situation for the resource manager. Hence we have, for each situation in \( S \), a winning strategy for the resource manager, i.e. \( S \) is only composed of safe situations. By the way we also proved that, for this kind of game, each situation is either safe or unsafe: there is no configuration where the issue of the game is undefined (for good players).

Interestingly, the procedure we just described does not work the other way round, i.e., by constructing \( S' \) as follows:

1. let \( S' \) be the singleton set containing the winning situation for the resource manager: \( S' = \{(f_1, \ldots, f_i, \ldots, f_P)\} \)

2. while there exists \( \emptyset ' \in S', \exists \emptyset ' \in S' : \emptyset \rightarrow \emptyset ' \), we add \( \emptyset \) to \( S' \) or \( \emptyset ' \in S', \forall \emptyset ' : \emptyset \rightarrow \emptyset ' \Rightarrow \emptyset ' \in S' \), we add \( \emptyset \) to \( S' \).

Again, at each step \( S' \) only contains safe configurations and the procedure terminates, but then it does not in general contain the whole set of safe configurations. Indeed, a closer look at the procedure reveals that \( S' \) only contains safe states from which there is a strategy allowing the resource manager to win in a bounded number of moves (a possible bound being the number of steps needed to stabilise \( S' \)); hence, in case there are loops, even with a single process, the initial state will never be recognised as safe. Stated in other words, if we start from outside the stabilised \( S' \), there is a strategy for the processes allowing them to stay outside \( S' \), thus impeaching the resource manager to win, but in general this strategy will not be licit because it does not induce finite evolutions as required (for instance a process will manage to stay forever in a loop).

Of course, the complexity of the procedure is in general exponential, but this was expected since the flowchart model is more general than the linear...
one, which is already exponential when there are many resource types. Moreover, the problem we solve is not exactly the same as the one solved by Hebalkar. Indeed, Hebalkar's algorithm determines if a specific configuration is safe or not, while here we characterise in one stroke the status of all the possible configurations. If the number of safe or unsafe configurations is not too high, or if they have some form of regularity, and if the system is known in advance, we may then a priori store them in an adequate data structure, and exploit it during the system evolutions.

It is also possible to derive some general properties of the rather general game we defined; the (inductive) proofs are a bit lengthy (there are many subcases to examine), so that we shall not give them here, but they are by no means difficult. For instance, it may be shown that \( \forall \varepsilon : \mathcal{F} \text{ is (un)safe } \iff \varepsilon \text{ is (un)safe, i.e., the status, safe or unsafe, of a configuration does not rely on the player who has the lead, and we may say that } \varepsilon \text{ is safe or not. Also, (as expected, but here it is a proved property and not an assumed one) it is never interesting to delay a request for release. It is also possible to simplify a bit the game without modifying the set of (un)safe configurations, by assuming for instance that the resource manager always waits that all the non-finished processes (if there are some) make their next (non-releasing) request before examining which one to grant, that no multiple requests or grants are performed, ...}

8. CONCLUSION

We have shown that the analysis of concurrent processes is a very delicate subject, i.e., it is very easy to make a mistake, or to miss a problem. We have seen how the introduction of a game to analyse the evolutions of a system may be precious, and it is not surprising that later this technique became so popular in the model checking field. We also saw that breaking a symmetry (cf. Havender’s hierarchical technique, and the difference between the strategies of the resource manager and the ones of the coalition of processes) may be useful.

9. REFERENCES


