Deadlock Avoidance, Non-Linearity and Games

Raymond Devillers

Département d'informatique, CP212
Université Libre de Bruxelles
B-1050 Bruxelles, Belgium
rdevil@ulb.ac.be

Abstract. 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.
Keywords: 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, like in the figure, one of them being reserved by the OS for its ancillary tasks, a forward 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 are available, the OS may grant one; now, if F is supended (waiting for instance 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

Fig. 1. IBM 2401 magnetic tape units.

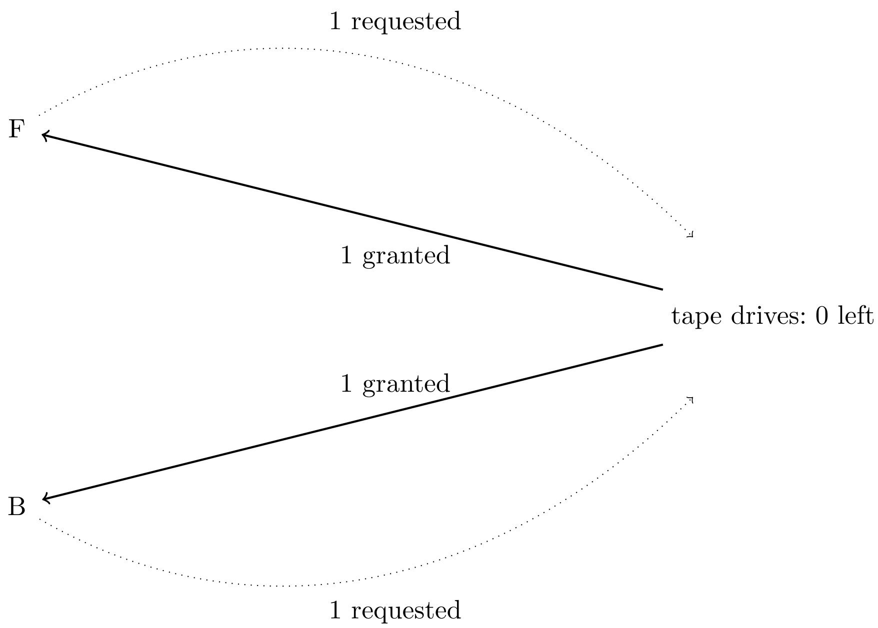

Fig. 2. Representation of a simple deadlock situation.

the OS, which anyway is usually in charge of managing the resources of the system. Here, we considered independent processes, interacting implicitely through the physical resources of the system, but soon it was discovered that it was possible to develop cooperating processes, interacting more explicitely 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 systems, 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) (centralized 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 by Holt in [14,15], 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, backtraking is not necessary and the check is in $\mathcal{O}(P^2T)$, 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 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 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; and one creates a set of the processes with a null counter.
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 $\mathcal{O}(PT)$, 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 $\mathcal{O}(P^2)$, nothing is gained. Better procedures (like the quicksort or the heapsort) are in $\mathcal{O}(P \log P)$ 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 $\mathcal{O}(P + I)$, like the counting sort, so that, if we assume that all types have the same I, we finally get a complexity in $\mathcal{O}((P + I)T)$, 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 there 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 is 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 preemtible 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 et al [5] 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 intrisically 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 happen that it is impossible to restart a process after killing it, if it has modified a file and 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, checkpoints 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 preferrable to apply a prevention or an avoidance strategy.

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 [11], 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/attribution, 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 neutralized 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, 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 algorithms for detecting unsafe situations in this case was published by Dijkstra in [8] when there is a single resource type, and by Haberman in the general case [9,10], 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 he is sure each client will be able to perform his investments and himself will recover all its 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 $\mathcal{O}(P^2T)$.

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, do we 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 [12,13] 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 backtraking. 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 schematized in Figure 3

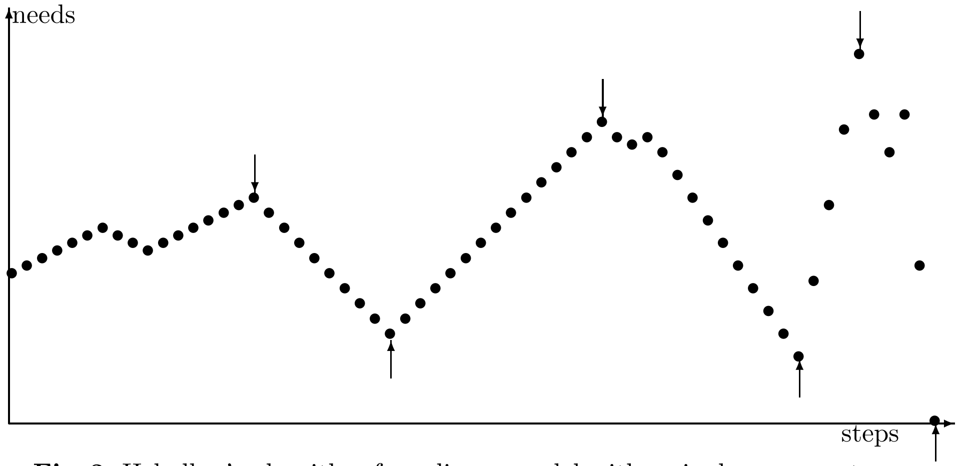

Fig. 3. Hebalkar's algorithm for a linear model with a single resource type.

6 Avoidance: the branching model

Then, Hebalkar considered the case where each process specifies a future history which is only linear or finitely branching[1]. 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 recognized 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.

$$\begin{tikzcd}
P_1 : s_1^1 \arrow[r] & s_1^2 \arrow[r] & s_1^3 \arrow[r] & s_1^4 \arrow[r] & s_1^5 \arrow[r] & s_1^1 \\
& & & s_2^4 & & \\
P_2 : s_2^1 \arrow[r] & s_2^2 \arrow[r] & s_2^3 \arrow[ur] \arrow[dr] & & & \\
& & & s_2^5 & &
\end{tikzcd}$$

Fig. 4. Why Hebalkar's idea to manage branching histories doesn't work.

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 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; so that, 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. 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_1^1, s_2^1), if P_2 follows the high linear history s_2^2, s_2^3, while P_1 follows its unique linear history s_1^2, s_1^3, s_1^4, s_1^5, s_1^6, the sequence s_2^2, s_2^3, end_2, s_1^2, s_1^3, s_1^4,

[1] he also considered the case of loops, and reduced this case to a finite set of branching systems.

$$P_1 : s_1^1(0,0,0,1) \to s_1^2(0,1,0,1) \to s_1^3(0,1,0,0) \to s_1^4(0,1,1,0) \to s_1^5(0,0,1,0) \to s_1^6(1,0,1,0)$$

$$\begin{tikzcd} & & & s_2^3(1,1,1,0) \\ P_2 : & s_2^1(1,0,0,0) \arrow[r] & s_2^2(1,1,0,0) \arrow[ur] \arrow[dr] & \\ & & & s_2^4(1,1,0,1) \end{tikzcd}$$

Fig. 5. A concrete counter-example to Hebalkar's idea.

s_1^5, s_1^6, end_1 is realizable, but there is no complete realizable sequence starting with s_1^2; indeed, if P_1 progresses first, then it has to continue until reaching the state (s_1^5,s_2^1), 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 we are stuck. And if P_2 follows the low linear history s_2^2, s_2^4, the sequence s_1^2, s_1^3, s_1^4, s_1^5, s_2^2, s_2^4, end_2, s_1^6, 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 we are stuck.

7 Avoidance: the flowchart model

In order to solve this problem, we shall first generalize 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 for more resources, or a status quo); this allows to represent branching situations mixed with (possibly imbricated) loops; we shall also assume there is a final step (with null need), that we do not know any bound on the number of times loops are followed, but that, if a process is not blocked, it will eventually head to the final step. This is illustrated in Figure 6, where each step s_i has a (vectorial) need $\overline{n}_i$, the starting step s_0 as well as the final one f do not use any resource, and for instance either $\overline{n}_2 \leq \overline{n}_8$ or $\overline{n}_2 \geq \overline{n}_8$ (hence allowing $\overline{n}_2 = \overline{n}_8$) componentwise.

Usually, and this is indeed the idea behind the banker's algorithm (as well as behind Hebalkar's algorithm for the linear histories), in order to detect if a situation is safe or not, one considers the worst case evolution of each process, and if we can manage in order to get out of the trouble, that means that we are safe, otherwise we are unsafe since the worst case is a possible case anyway. But here, we do not really have in general any "worst case" evolutions. So what could be the way to detect (un)safe situations?

The idea is to consider that the worst case arises from a game between the resource manager, which aims at conducting each process to its end, and

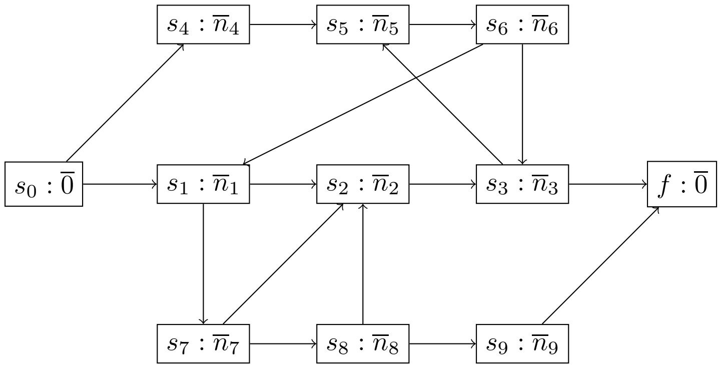

Fig. 6. The flowchart model of a process.

a coalition of crazy processes which aim at blocking the system and creating a deadlock. If, from some configuration, there is a winning strategy for the resource manager, the configuration is safe; if there is a winning situation for the processes willing to commit suicide, the situation is clearly unsafe; in principle, we could also have situations for which there is not winning strategy for any of the players, which could also be considered as unsafe, but we shall see that this is not the case for the game we have at hand.

In order to define more precisely that game, we shall first define the states of the game, then the possible moves of the players. For each process p_i, we shall distinguish three types of local states: a state s_i^j will denote a situation where the process is working in step s_i^j, having acquired the vector (one component for each resource type) $\overline{n}_i^j$ of resources needed by it (we shall as usual assume that $\overline{n}_i^j \leq \overline{r}$, the vector giving for each resource type the number of resources, free or granted, available in the whole system); a state (s_i^j, s_i^k) will denote a situation where the process p_i has finished step s_i^j, has required to enter step s_i^k (with $\overline{n}_i^j \geq \overline{n}_i^k$ or $\overline{n}_i^j \leq \overline{n}_i^k$ or $\overline{n}_i^j = \overline{n}_i^k$) but this has not been granted yet, hence the process still has the vector $\overline{n}_i^j$ of resources previously acquired by it; the state f_i will denote the fact that the process has reach the end, has released all its resources and disappeared. We shall denote generically those three cases by a state ε_i, with acquired resources $\overline{n}(\varepsilon_i)$. Then, a global state of the system will be denoted by a vector $\varepsilon = (\varepsilon_1, \ldots, \varepsilon_i, \ldots, \varepsilon_P)$, assuming there are P processes in the system, where $\sum_{i=1}^{P} \overline{n}(\varepsilon_i) \leq \overline{r}$, i.e., we only consider realizable states.

We shall assume that the game is an alternating one, i.e., that it progresses through an alternation of moves made by each player (but there will be waiting moves, allowing each player to "pass its turn" and to allow the other player to perform many moves in a row); hence, for the game, there will be states $\overline{\varepsilon}$, where the system is in state ε and the resource manager has the lead, and states $\underline{\varepsilon}$, where the system is in state ε and the coalition of processes has the lead.

The moves of each player are as follows:

1. for the resource manager:
   - $\overline{\varepsilon} \longrightarrow \underline{\varepsilon}'$ if ε and ε' are realizable, $\varepsilon_i = (s_i^j, s_i^k)$ and $\varepsilon_i' = s_i^k$ for one or more i's, and $\varepsilon_i = \varepsilon_i'$ for the other ones, i.e., one or more requests are granted;
   - $\overline{\varepsilon} \longrightarrow \underline{\varepsilon}$ if some processes are in a working state in ε, 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: one has to grant some requests, or recognize we lost;
2. for the processes:
   - $\underline{\varepsilon} \longrightarrow \overline{\varepsilon}'$ 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 $\varepsilon_i = \varepsilon_i'$;
   - $\underline{\varepsilon} \longrightarrow \overline{\varepsilon}$ unless all the processes are finished, i.e., $\forall i : \varepsilon_i = f_i$; indeed in that case, the processes recognize 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[2] for the resource manager: $\mathcal{U} = \{\overline{\varepsilon} \mid \exists i, j, k : \varepsilon_i = (s_i^j, s_i^k)$, all the non-requesting processes are finished, and no request may be granted $\}$; this is easy to construct when we build the graph of the game;
2. **while**
   $\exists \underline{\varepsilon} \notin \mathcal{U}, \exists \overline{\varepsilon}' \in \mathcal{U} : \underline{\varepsilon} \longrightarrow \overline{\varepsilon}'$, we add $\underline{\varepsilon}$ to $\mathcal{U}$
   or
   $\exists \overline{\varepsilon} \notin \mathcal{U}, \forall \underline{\varepsilon}' : \overline{\varepsilon} \longrightarrow \underline{\varepsilon}' \implies \underline{\varepsilon}' \in \mathcal{U}$, we add $\overline{\varepsilon}$ to $\mathcal{U}$.
   endwhile

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 since 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 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 stabilized, it is not only a set of unsafe situation, but the set of all of them.*

[2] 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. Let $\mathcal{S}$ be the complement of $\mathcal{U}$, i.e., the set of all the configurations of the game that were not incorporated in $\mathcal{U}$. By construction, if $\overline{\varepsilon} \in \mathcal{S}$, there must be some $\underline{\varepsilon}' \in \mathcal{S}$ too such that $\overline{\varepsilon} \longrightarrow \underline{\varepsilon}'$, since otherwise $\overline{\varepsilon}$ could have been added to $\mathcal{U}$ and would not be in $\mathcal{S}$; and if $\underline{\varepsilon} \in \mathcal{S}$, then $\forall \underline{\varepsilon} \longrightarrow \overline{\varepsilon}'$, $\overline{\varepsilon}' \in \mathcal{S}$, for the very same reason; hence, if we are in $\mathcal{S}$, 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 $\mathcal{S}$, a winning strategy for the resource manager, i.e. $\mathcal{S}$ 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). $\square$

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

1. let $\mathcal{S}'$ be the singleton set containing the winning situation for the resource manager: $\mathcal{S}' = \{\underline{(f_1, \ldots, f_i, \ldots, f_P)}\}$
2. **while**
   $\exists \overline{\varepsilon} \notin \mathcal{S}', \exists \underline{\varepsilon}' \in \mathcal{S}' : \overline{\varepsilon} \longrightarrow \underline{\varepsilon}'$, we add $\overline{\varepsilon}$ to $\mathcal{S}'$
   or
   $\exists \underline{\varepsilon} \notin \mathcal{S}', \forall \overline{\varepsilon}' : \underline{\varepsilon} \longrightarrow \overline{\varepsilon}' \implies \overline{\varepsilon}' \in \mathcal{S}'$, we add $\underline{\varepsilon}$ to $\mathcal{S}'$.
   endwhile

Again, at each step $\mathcal{S}'$ only contains safe configuations 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 $\mathcal{S}'$ 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 stabilize $\mathcal{S}'$); hence, in case there are loops, even with a single process, the initial state will never be recognized as safe. Stated in other words, if we start from outside the stabilized $\mathcal{S}'$, there is a strategy for the processes allowing them to stay outside $\mathcal{S}'$, 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, Hebelkar's algorithm determines if a specific configuration is safe or not, while here we characterize 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 them a prori store then 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 $\forall \varepsilon : \overline{\varepsilon}$ is (un)safe $\iff \underline{\varepsilon}$ 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 ε 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.

References

1. P.A. Abdulla et al. *On-the-fly analysis of systems with unbounded lossy fifo channels.* CAV'98, LNCS 1427, pp 305–318, 1998.
2. P.A. Abdulla et al. *Symbolic verification of lossy channel systems: Application to the bounded retransmission protocol.* TACAS'99, LNCS 1579, pp 208–222, 1999.
3. R. Devillers. *The* Prime Game *for a few representations.* Proceedings AutoMathA'09 (Liège, Belgium), 2009.
4. A. Finkel et al. *On the omega-language expressive power of extended Petri nets*, TCS, vol 356 (3), pp 374–386, 2006.
5. E. G. Coffman, M. Elphick and A. Shoshani. *System Deadlocks*, ACM Computing Surveys, vol 3 (2), pp 67–78, 1971.
6. L. de Alfaro, M. Faella, R. Majumdar and V. Raman. *Code Aware Resource Management.* EMSOFT: Fifth ACM International Conference on Embedded Software, ACM Press, pp 191–202, 2005.
7. R. Devillers, *Game Interpretation of the Deadlock Avoidance Problem.* CACM , vol 20 (10), pp 741–745, 1977.
8. E.W. Dijkstra. *Cooperating Sequential Processes*, Programming Languages: NATO Advanced Study Institutes, pp 43-112, ed. Genuys, Academic Press, 1968.
9. A.N. Habermann. *On the Harmonious Cooperation of abstract Machines.* PhD Thesis T.H. Eindhoven, 1967.
10. A.N. Habermann. *Prevention of system deadlocks.* CACM, vol 12 (7), pp 373–377, 1969.

11. J.W. Havender. *Avoiding Deadlocks in Multitasking Systems*, IBM System Journal, vol 2, pp 74–84, 1968.
12. P.G Hebalkar. *Deadlock-free Resource Sharing in Asynchronous Systems*, PhD thesis, MIT, 1970.
13. P.G Hebalkar. *A Graph Model for Analysis of Deadlock Prevention in Systems with Parallel Computations*, IFIP Congress, vol TA3, pp 168–172, 1971.
14. R.C. Holt. *On Deadlock in Computer Systems*, PhD thesis Cornell, 1971.
15. R.C. Holt. *Some Deadlock Properties of Computer Systems*, Computing Surveys, vol 4 (3), pp 179–196, 1972.
16. A. Shoshani and E. G. Coffman. *Detection, Prevention and Recovery from System Deadlocks*, Proceedings of the 4th Annual Princeton Conference on Information Science and Systems, pp 355–360, 1970.