Deadlock models and general algorithm for distributed deadlock detection

Jerzy Brzezinski, Jean-Michel Hélary, Michel Raynal

INRIA

UNITE DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

1 9 9 2

25ème
anniversaire

N° 1776

Programme 1
Architectures parallèles, Bases de données,
Réseaux et Systèmes distribués

DEADLOCK MODELS AND GENERAL ALGORITHM FOR DISTRIBUTED DEADLOCK DETECTION

Jerzy BRZEZINSKI
Jean-Michel HÉLARY
Michel RAYNAL

Octobre 1992

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTEMES ALEATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CEDEX FRANCE
Tél. : 99 84 71 00 - Télex : UNIRISA 950 473 F

Programme 1, projet ADP (Algorithmes Distribués et aPplications)

Octobre 1992

Publication Interne n°681 - 26 pages

Deadlock Models and General Algorithm for Distributed Deadlock Detection

Jerzy Brzezinski*, **Jean-Michel Hélary, Michel Raynal**

Institute of Computing Science
Technical University of Poznan
60-965 Poznan, POLAND
e-mail:brzezins@plpotu51.bitnet

IRISA
Campus de Beaulieu
35042 Rennes Cedex, FRANCE
helary, raynal@irisa.fr

Abstract: This paper deals with the problem of deadlock detection in asynchronous message communication systems. The considered system model covers unspecified receptions, not *FIFO* channels, and general resource (message) requests including, among others, *AND*, *OR*, *AND-OR*, *k-out-of-n* requests. In this context hierarchies of deadlock models and deadlock detection problems are introduced and discussed. Then, a token-based algorithm for detection of deadlocked sets is proposed, formally described, evaluated and proven correct. Moreover, some possible extensions and refinements of the basic solution concerning individual and global terminations, improvements of token routing, and parallel detection execution, are outlined and analyzed. When compared to deadlock detection algorithms described so far the solution proposed here behaves favorably with respect to generality, flexibility and communication complexity.

Index terms: Communication deadlock, deadlock models, distributed algorithms, distributed deadlock detection, message communication systems, resource deadlock.

Modèles d'interblocage et algorithme général de détection répartie d'interblocage

Résumé: *On examine le problème de la détection d'interblocages dans un système asynchrone basé sur la communication par messages. Le modèle considéré prend en compte le cas des réceptions non spécifiées, les canaux non-FIFO, et une forme générale de requêtes comprenant, entre autres, les modèles ET, OU, ET-OU, k-parmi-n. Dans ce contexte une hiérarchie de modèles d'interblocage et de problèmes de détection associés est présentée. Puis, un algorithme à jeton permettant la détection d'ensembles de processus interbloqués est proposé, évalué et prouvé. De possibles extensions ou améliorations, relatives à la terminaison individuele ou globale, au routage du jeton, et à l'exécution parallèle de la détection, sont proposées et analysées. Par rapport aux algorithmes connus de détection d'interblocage, la solution proposée ici est avantageuse tant sur le plan de la généralité que sur ceux de la souplesse de mise en œuvre et de la complexité.*

*. The work of this author was supported in part by INRIA grant (when he was visiting IRISA), and by KBN Grant No. 335209102 (KBN 2).

Deadlock Models and General Algorithm for Distributed Deadlock Detection

Jerzy Brzezinski*, **Jean-Michel Hélary, Michel Raynal**

Institute of Computing Science
Technical University of Poznan
60-965 Poznan, POLAND
e-mail:brzezins@plpotu51.bitnet

IRISA
Campus de Beaulieu
35042 Rennes Cedex, FRANCE
helary, raynal@irisa.fr

Abstract: This paper deals with the problem of deadlock detection in asynchronous message communication systems. The considered system model covers unspecified receptions, not *FIFO* channels, and general resource (message) requests including, among others, *AND*, *OR*, *AND-OR*, *k-out-of-n* requests. In this context hierarchies of deadlock models and deadlock detection problems are introduced and discussed. Then, a token-based algorithm for detection of deadlocked sets is proposed, formally described, evaluated and proven correct. Moreover, some possible extensions and refinements of the basic solution concerning individual and global terminations, improvements of token routing, and parallel detection execution, are outlined and analyzed. When compared to deadlock detection algorithms described so far the solution proposed here behaves favorably with respect to generality, flexibility and communication complexity.

Categories and Subject Descriptor: C.2.4 [**Computer-Communication Networks**]:- Distributed Systems-*distributed applications; distributed databases; network operating systems;* D.4.1 [**Operating Systems**]: Process Management-*concurrency; deadlocks; synchronization;* D.4.4 [**Operating Systems**]: Communication Management-*network communication.*

General Terms: Algorithms

Additional Key Words and Phrases: Communication deadlock, deadlock models, distributed algorithms, distributed deadlock detection, message communication systems, resource deadlock.

*. The work of this author was supported in part by INRIA grant (when he was visiting IRISA), and by KBN Grant No. 335209102 (KBN 2).

1 Introduction

Development of distributed systems, such as computer networks and distributed memory parallel computers, seems to be one of the most important and promising way to meet the ever increasing demands of computer applications. This is because just distributed systems offer, at least potentially, dynamic sharing of resources (e.g., servers of one kind or another, peripheral devices or controllers, specific processing units, program files or data items), communication between many sites running simultaneously but cooperating with each other in the realization of a common goal (distributed transactions, parallel algorithms, etc), higher reliability resulting from redundancy of processing units, as well as modularity and expendability, among other things. In general these systems may be thought of as a set of nodes interconnected by transmission channels. Each node is equipped with a processor and a local memory. Communication between any pair of processors is realized only by message passing as no common memory is available.

Distributed systems can be classified as synchronous and asynchronous. In this paper we are only concerned with asynchronous systems, i.e., systems with no known bound on relative processor speeds or message transmission time. Asynchrony, characteristic for most real systems, makes coordination between distributed processors difficult. It implies, moreover, that even classical problems -like: mutual exclusion, termination detection, deadlock handling, determination of global state, election, consensus gaining, transaction control, query optimization, etc - have to be addressed again to develop new control mechanisms as only distributed algorithms are acceptable in this context. These algorithms are composed of processes which are executed at system nodes and exchange information with each other by message passing. Efficient distributed control algorithms are necessary to achieve potential profits of distributed systems.

In this paper we focus our attention on the deadlock problem. Informally, deadlock refers to the situation in which some group of processes exists, such that no process in this group can send message (release resource) until it receives the required message (resource) from other process of the group. When this occurs, the deadlocked processes wait permanently and the progress of their execution cannot be achieved. In this case, the execution of processes can turn out completely useless unless proper and careful control is exercised. To handle deadlock one can try to adapt approaches known from centralized systems, i.e., prevention, avoidance, and detection with recovery ([8], [16]). Let us recall, that deadlock prevention is based on denying one of necessary conditions for deadlock occurrence (e.g., circular wait, no preemption, or hold and wait condition). In deadlock avoidance the message is sent (resource is granted) when in spite of this event there is still at least one execution sequence that allows completion of all processes. In deadlock detection messages are sent (resources are granted) without any constraints due to possibility of deadlock. However, the state of the system is checked periodically, or when a deadlock is suspected, to determine if a set of processes is deadlocked. This checking is performed by a deadlock detection algorithm. If a deadlock is discovered, recovery from it by aborting one or more deadlocked processes is necessary. The suitability of a deadlock handling approach greatly depends on the application and environment. In distributed environment deadlock handling is peculiarly complex as distributed algorithms are required and no node has accurate knowledge of the system state.

This paper addresses deadlock detection in distributed systems. As was mentioned, it is a very important problem in various distributed applications including: communication networks, distributed database systems, massively parallel systems, etc. Depending on the application processes can make requests according to different request models (see e.g., [1], [11], [18]). The simplest possible request model is one in which a process can require at most one message (resource) at a time. For example, it refers to buffer requests in packet-switched store-and-forward networks ([2], [13], [17])). In the *AND* model (known also as *resource model*) processes are permitted to request simultaneously a set of messages. A process cannot execute until it acquires *all* messages (resources) which it is waiting for ([5], [7], [12]). This model represents, for instance, possible requests of transactions to lock several data items ([1]). Another model of requests is the *OR* model, called also *communication model*, where a process can request messages from a set of processes; the process can proceed only if it receives a message from *any* one of the processes it is waiting for. This model refers directly to alternative control structures in programming languages like *CSP* and *ADA* as well as to replicated database systems, where read request for a replicated data item can be satisfied by reading any copy of it ([1], [7]). The *OR-AND* model is a generalization of the two previous ones. It can specify any combination of message (resource) requests expressed by logical *and* and *or* operators. For example, a process may require messages from *i and* from *j or k*. If the process receives a message from *i* it continues waiting for messages from *j or k*. On the other hand, if the process receives first message from *j* it need not wait for *k*. Thus, the request for *k* can be canceled. This model is well suited to many distributed systems (e.g., distributed operating systems, replicated database systems) where several sets of equivalent resources are concerned ([1]). In the *k out of n* model processes are permitted to request any *k* messages from a set of *n* processes ([3]). This model is also a generalization of *OR* and *AND* models since an *OR* request corresponds to *1 out of n*, and an *AND* request corresponds to *n out of n* one. The *k out of n* model refers especially to replicated database systems, where a quorum-based replica control algorithm is used to preserve database consistency ([1], [11]). In such an algorithm a transaction that wants to read a replicated data item must read r copies out of n, and in order to write a transaction must write w copies out of n, where $r+w > n$ and $2w > n$. More general models have not been really considered in the deadlock detection context as they lead to general problems of global state determination ([4], [18]).

The above hierarchy of request models is useful to classify distributed deadlock detection algorithms according to the complexity of the resource requests they permit. Several algorithms have been proposed for *one resource* and *AND* models ([5], [7], [12], [20]) as well as for *OR* model ([7], [21]). Distributed deadlock detection algorithm for *OR-AND* model has been developed by Hermann and Chandy (see e.g., [18]), and for *k out of n* model by Bracha and Toueg ([3]). Many of the above mentioned solutions have been presented, carefully analyzed and compared in two excellent surveys by Knapp ([18]), and Singhal ([22]).

Although, in general, the literature concerning deadlock detection in distributed systems is comprehensive, this area still offers a great opportunity for improvement as there is lack of simple algorithm for more general system models permitting, for instance, requests expressed by any logical combination of *k out of n* requests, any (not necessarily *FIFO*) communication channels, etc. This paper presents a distributed deadlock detection algorithm allowing such generalizations in the context of reliable distributed systems. In its simplest form the proposed algorithm applies token passing mechanism as a means for global information exchange. The application of token makes the algorithm easy to understand and implement, however, more parallel mechanism (e.g., tree diffusion) can be also used by the algorithm to capture global information. When compared to deadlock detection algorithms described so far, the algorithm proposed here behaves favorably with respect to generality, simplicity, and communication complexity.

The paper consists of five sections. Section 2 introduces fundamental definitions and models for deadlock in message communication systems. In Section 3 the basic formulation of the proposed algorithm is described and proven correct Section 4 displays possible refinements and modifications of the algorithm, and its adaptation to solve the distributed termination problem. Finally, Section 5 concludes the paper.

2 Basic definitions and problem formulation

2.1 The underlying system model

The underlying communication system supporting distributed applications is made of a set of nodes sharing no common memory and communicating only by message exchange through communication channels. These channels are assumed to be asynchronous (transfer delays are unpredictable) and reliable (no message is lost, corrupted or duplicated); they can be *FIFO* or not and they have infinite buffer capacity. Moreover there is no global physical clock accessible to the nodes.

The message sent by a process running on a node is confided to the underlying system. This system carries the message till the destination node and puts it in a local buffer (the message has then *arrived)*. This message can then be extracted from the buffer when the application process requires it (and the message is then *consumed*).

2.2 The application program

The application program is composed of a set **P** of processes P_i, $1 \le i \le n$, communicating by asynchronous message passing. We suppose that there is one-to-one correspondence between nodes and processes, and that the assignment is static.

As communication is asynchronous, the process sending a message is never blocked. At any time a process is either *active* or *passive*. Only active processes can send and consume messages. Moreover, an active process can become spontaneously passive requiring some messages (resources) in order to continue its execution. The requirement is expressed by an activation condition (detailed in Section 2.3) defined over the set $\mathbf{DS}_i$ of processes from which passive process P_i is expecting messages. The set $\mathbf{DS}_i$ is called *dependent set* of P_i. A passive process can only become active when its activation condition is fulfilled. (Moreover when a process is activated messages whose arrivals fulfilled the associated activation condition are extracted from input buffers and consumed.) The formulation of the activation condition depends on the request model considered. The next Section presents such models in increasing complexity order.

Let us remark, that this model allows *unspecified reception*: this occurs when a message sent by a process has arrived but it will never be consumed by the destination process.

A passive process that has terminated its computation executing, for example, an *end* or *stop* statement is said to be *individually terminated*: its dependent set is empty and therefore it can never be activated. We assume for a while that processes are never individually terminated; this assumption is only for a sake of presentation simplicity and it will be relaxed in Section 4.1.

2.3 Request models

2.3.1 *AND* model

In this model a passive process P_i can be activated (and so the activation condition is fulfilled) only after a message from each process P_j belonging to $\mathbf{DS}_i$ has arrived. (This models receive statements that are atomically on several messages.)

2.3.2 *OR* model

In *OR* model, a passive process P_i can be activated when a message from any process P_j belonging to $\mathbf{DS}_i$ has arrived. (This models classical non-deterministic choices of receive statements.)

2.3.3 *OR-AND* model

For *OR-AND* model, the requirement of a passive process P_i is defined by a set $\mathbf{R}_i$ of set $\mathbf{DS}_i^{1}$, $\mathbf{DS}_i^{2}$,..., $\mathbf{DS}_i^{q}i$, such that for all r, $1 \leq r \leq q_i$, $\mathbf{DS}_i^{r} \subseteq \mathbf{P}$. The dependent set of P_i is: $\mathbf{DS}_i = \mathbf{DS}_i^{1} \cup \mathbf{DS}_i^{2} \cup ... \cup \mathbf{DS}_i{}^{q}i$. We mean here that process P_i waits for messages from *all* processes belonging to $\mathbf{DS}_i^{1}$, *or* for messages from *all* processes belonging to $\mathbf{DS}_i^{2}$, *or...*, *or* for messages from *all* processes belonging to $\mathbf{DS}_i^{q}i$. As an example, suppose P_i waits for messages from: P_a *or* $(P_b$ *and* $(P_c$ *or* $(P_d$ *and* $P_e)))$. In disjunctive form this gives: P_i waits for P_a *or* $(P_b$ *and* $P_c)$ *or* $(P_b$ *and* P_d *and* $P_c)$. In this case $\mathbf{DS}_i^{1}=\{P_a\}$, $\mathbf{DS}_i^{2}=\{P_b, P_c\}$ and $\mathbf{DS}_i^{3}=\{P_b, P_d, P_e\}$. Let us note, that if messages from P_c and P_d have arrived, and then message from P_b arrives, the activation condition of P_i is fulfilled (and this activation will provoke consumption of the messages from P_b and P_c).

2.3.4 Basic *k out of n* model

In this model the requirement of a passive process P_i is defined by the set $\mathbf{DS}_i$ and an integer k_i, $1 \le k_i \le |\mathbf{DS}_i| = n_i$. Process P_i can be activated when messages from k_i distinct processes belonging to $\mathbf{DS}_i$ have arrived.

2.3.5 Disjunctive *k out of n* model

A more general request model can be introduced including additionally *k out of n* requests. The requirement of a passive process P_i is defined by a set $\mathbf{R}_i$, as previously (Section 2.3.3), and by a set of integers $\mathbf{K}_i = \{k_i^1, k_i^2, ..., k_i^{q_i}\}$ with $1 \le k_i^r \le |\mathbf{DS}_i^r| = n_i^r$ for all r, $1 \le r \le q_i$. The dependent set of P_i is: $\mathbf{DS}_i = \mathbf{DS}_i^1 \cup \mathbf{DS}_i^2 \cup ... \cup \mathbf{DS}_i^{q_i}$. A process P_i is activated when:

messages from k_i^1 processes composing $\mathbf{DS}_i^1$ have arrived *or*
messages from k_i^2 processes composing $\mathbf{DS}_i^2$ have arrived *or*

.

.

.

messages from $k_i^{q_i}$ processes composing $\mathbf{DS}_i^{q_i}$ have arrived .

Let us note, that if for all r, $1 \le r \le q_i$, $k_i^r = n_i^r = |\mathbf{DS}_i^r|$ this model reduces to the *OR-AND* model. On the other hand, when $q_i = 1$ and $k_i^1 \le |\mathbf{DS}_i^1|$, then we have the basic *k out of n* model, with $k = k_i^1$ and $n = n_i^1$. The *AND* model is deduced when $q_i = 1$, $\mathbf{DS}_i^1 = \mathbf{DS}_i$ and $k_i^1 = n_i^1 = |\mathbf{DS}_i|$. The simple *OR* model is obtained when $q_i \ge 1$ and for all r, $1 \le r \le q_i$, $|\mathbf{DS}_i^r| = 1$.

2.3.6 Predicate *fulfilled*

Finally, in order to abstract the activation condition of a passive process P_i the following predicate *fulfilled*$_i$ (**A**) is introduced, where **A** is a subset of **P**. Predicate *fulfilled*$_i$ (**A**) is *true* if and only if messages arrived (and not yet consumed) from all processes belonging to set **A** are sufficient to activate process P_i. Of course, the following monotonicity property is verified: if $\mathbf{X} \subseteq \mathbf{Y}$ and *fulfilled*$_i$ (**X**) is *true*, then *fulfilled*$_i$ (**Y**) is also *true*; moreover *fulfilled*$_i(\emptyset)$ is

false.

If we consider the previous disjunctive *k out of n* model the predicate is expressed as follows. Let P_i be a passive process whose requirements are defined by the sets $\mathbf{R}_i$ and $\mathbf{K}_i$. Then we get the following definition:

$fulfilled_i(\mathbf{A}) \equiv \exists r\colon 1 \le r \le q_i\colon |\mathbf{DS}_i^r \cap \mathbf{A}| \ge k_i^r$

Similar definition can be obtained for the other models.

2.4 Deadlock definition

Let *deadlock* (**B**) be a predicate meaning, at time moment *t*, that the nonempty set **B** of processes is deadlocked at this time. If *deadlock* (**B**) is *true* at some time moment, it remains *true* as deadlock occurrence is persistent: *deadlock* (**B**) is *stable property* ([4], [6], [14]). According to the request model the formal definition of this predicate can take several forms. In order to state these definitions, the following variables are introduced:

- $passive_i$: *true* iff P_i is passive.
- $arr_i\,(j)$: *true* iff a message from P_j has arrived and has not yet been consumed by P_i.
- *empty* (j, i): *true* iff all messages sent by P_j to P_i have arrived.

All these elements allow to give precise definitions of *deadlock* (**B**) within the different models.

AND model

$$deadlock\,(\mathbf{B}) \equiv (\mathbf{B} \neq \varnothing) \wedge (\forall P_i\!: P_i \in \mathbf{B}\!: (passive_i \wedge (\exists P_j\!: P_j \in \mathbf{DS}_i \cap \mathbf{B}\!: (empty\,(j, i) \wedge \neg\, arr_i\,(j)))))$$

OR model

$$deadlock\,(\mathbf{B}) \equiv (\mathbf{B} \neq \varnothing) \wedge (\forall P_i\!: P_i \in \mathbf{B}\!: (passive_i \wedge (\mathbf{DS}_i \subseteq \mathbf{B}) \wedge (\forall P_j\!: P_j \in \mathbf{DS}_i\!: (empty\,(j, i) \wedge \neg\, arr_i\,(j)))))$$

EXAMPLE: Let us consider a sample underlying system consisting of nodes N_a, N_b, N_c, N_d and N_e, whose topology is presented in Fig. 1a. Processes P_a, P_b, P_c, P_d and P_e compose the application program. We analyze system state at time *t*. At this moment processes P_a, P_b, P_d and P_e are passive, but process P_c is active. Dependent sets are as follows (see Fig.1b.): $\mathbf{DS}_a = \{P_c, P_d\}$, $\mathbf{DS}_b = \{P_d\}$, $\mathbf{DS}_c = \varnothing$, $\mathbf{DS}_d = \{P_b, P_e\}$ and $\mathbf{DS}_e = \{P_b\}$. Local buffers of nodes N_a, N_c, N_d and N_e are empty. In local buffer of node N_b is a message *m'* arrived from P_e (but P_b is not expecting messages from P_e at this moment). Moreover, a message *m* sent by P_a to P_b has not yet arrived and thus *empty (a, b)* is *false*. However, $P_a \notin \mathbf{DS}_b$ and therefore when message *m* will arrive at N_b it will not be able to activate P_b (unless P_b will become active and then passive again with different dependent set). In this state only process P_a can be activated by active P_c. Consequently, as far as *OR* request model is concerned, *deadlock* (**B**) is *true* at time *t*, for $\mathbf{B} = \{P_b,$

$P_d, P_e\}$. On the other hand, if P_a needs for its activation messages from P_c *and* P_d (*AND* request model), then at this time the set of deadlocked processes is equal to $\{P_a, P_b, P_d, P_e\}$.

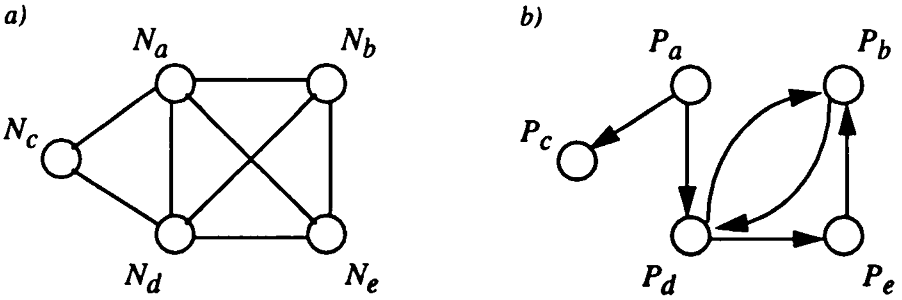

Fig. 1. An illustration of deadlock in the *OR* model.

a) The topology of underlying system
b) The graph representing the requests of processes

OR-AND model

$$\begin{aligned} &\textit{deadlock}\ (\mathbf{B}) \equiv (\mathbf{B} \neq \emptyset) \wedge (\forall P_i\text{: } P_i \in \mathbf{B}\text{:}\\ &\qquad (\textit{passive}_i \wedge (\forall r\text{: } 1 \leq r \leq q_i\text{:}\\ &\qquad\qquad (\exists P_j\text{: } P_j \in \mathbf{DS}_i^r \cap \mathbf{B}\text{:}\\ &\qquad\qquad\qquad (\textit{empty}(j, i) \wedge \neg \textit{arr}_i\ (j)))))) \end{aligned}$$

k out of n model

We have $n_i = |\,\mathbf{DS}_i\,|$, $1 \leq k_i \leq n_i$

$$\begin{aligned} &\textit{deadlock}\ (\mathbf{B}) \equiv (\mathbf{B} \neq \emptyset) \wedge (\forall P_i\text{: } P_i \in \mathbf{B}\text{:}\\ &\qquad (\textit{passive}_i \wedge\\ &\qquad (\exists \mathbf{D}_i\text{: } \mathbf{D}_i \subseteq \mathbf{DS}_i \cap \mathbf{B}\text{:}\\ &\qquad\quad ((|\,\mathbf{DS}_i \setminus \mathbf{D}_i\,| < k_i) \wedge (\forall P_j\text{: } P_j \in \mathbf{D}_i\text{:}\\ &\qquad\qquad\qquad (\textit{empty}\ (j, i) \wedge \neg\ \textit{arr}_i\ (j))))))) \end{aligned}$$

This means that with each process P_i belonging to **B** can be associated a set $\mathbf{D}_i$ such that arrivals of new messages from processes belonging $\mathbf{D}_i$ are not possible as $\mathbf{D}_i \subseteq \mathbf{B}$ and $\forall P_j$: $P_j \in \mathbf{D}_i$: *empty(j, i)*. Thus, one can await at most $|\,\mathbf{DS}_i \setminus \mathbf{D}_i\,|$ expected messages, but it is not sufficient to activate P_i when $|\,\mathbf{DS}_i \setminus \mathbf{D}_i\,| < k_i$ and $\forall P_j$: $P_j \in \mathbf{D}_i$: $\neg\ \textit{arr}_i\ (j)$.

Disjunctive *k out of n* model

$$
\begin{aligned}
\mathit{deadlock}\,(\mathbf{B}) \equiv (\mathbf{B} \neq \varnothing)\ \wedge\ (\forall P_i\!: P_i \in \mathbf{B}:\ & \\
& (\mathit{passive}_i \wedge \\
& (\forall r\!:1 \leq r \leq q_i\!: \\
& \quad (\exists \mathbf{D}_i^r\!: \mathbf{D}_i^r \subseteq \mathbf{DS}_i^r \cap \mathbf{B}: \\
& \qquad ((|\,\mathbf{DS}_i^r \setminus \mathbf{D}_i^r\,| < k_i^r) \wedge (\forall P_j\!:P_j \in \mathbf{D}_i^r: \\
& \qquad\qquad (\mathit{empty}\,(j, i) \wedge \neg\, \mathit{arr}_i\,(j))))))))
\end{aligned}
$$

General model

Let $\mathbf{ARR}_i$ denote the set of all processes P_j such that $arr_i\,(j) = true$. Moreover, let $\mathbf{NE}_i$ denote the set of all processes P_j such that $\neg$ *empty* (j, i).

$$
\begin{aligned}
\mathit{deadlock}\,(\mathbf{B}) \equiv (\mathbf{B} \neq \varnothing)\ \wedge (\forall P_i\!: P_i \in \mathbf{B}:\ & \\
& (\mathit{passive}_i \wedge \neg \mathit{fulfilled}_i\,(\mathbf{ARR}_i\ \cup \mathbf{NE}_i \cup (\mathbf{P} \setminus \mathbf{B}))))
\end{aligned}
$$

This predicate means that any $P_i \in \mathbf{B}$ cannot be activated even if messages from all processes belonging to $\mathbf{ARR}_i \cup \mathbf{NE}_i$ and from all processes not deadlocked, will arrive.

2.5 Deadlock detection problems

With the previous precise deadlock definitions various formulations for deadlock detection problem can be considered.

2.5.1 Detection of deadlock occurrence

In this formulation the question is: does there exist set **B**, $\mathbf{B} \subseteq \mathbf{P}$, such that *deadlock* (**B**) is *true*. The answer *yes* or *no* does not depend on process obtaining it.

2.5.2 Detection of a deadlocked process

In this formulation the question is: given a process P_α, is this process deadlocked or not, i.e., does there exist a set **B** such that *deadlock* (**B**) = *true* and $P_\alpha \in \mathbf{B}$. The answer *yes* or *no* depends on the process P_α.

2.5.3 Detection of a deadlocked set

In this formulation the problem is: find **B**, $\mathbf{B} \subseteq \mathbf{P}$, such that *deadlock* (**B**) = *true*. Of course, when as a result of searching for **B** we conclude **B** = ∅, then there is no deadlock.

2.5.4 Detection of the maximum deadlocked set

In this formulation the problem is: find **B**, such that:
$(\mathbf{B} \subseteq \mathbf{P}) \wedge (\mathit{deadlock}\ (\mathbf{B})) \wedge (\forall \mathbf{Q}: \mathbf{Q} \supset \mathbf{B}: \neg\ \mathit{deadlock}\ (\mathbf{Q}))$

It is easy to note that deadlock detection problems have been presented in increasing complexity order. Simultaneously, this order is consistent with increasing amount of information available at the time the detection terminates. This additional information is important as it is useful for deadlock recovery. Solution of any of the above problems is difficult in distributed message communication systems because no process has accurate knowledge of the global system state, i.e., the global state is not visible to any real observer. Thus, in practice, only states related to earlier observations can be obtained. However, during the collection of local states, these states are dynamically changing. Therefore, any deadlock detection algorithm in a distributed system can ensure only that deadlock which has occurred before the initiation of the algorithm will be detected, and that a detected set **B** of deadlocked processes is really deadlocked at the moment when the detection algorithm terminates.

3 A token-based algorithm for detection of a deadlocked set

3.1 Required properties of the algorithm

As usual, the specification of the detection algorithm can be stated using classical decomposition into *safety* and *liveness* properties. A boolean flag *dd* and a set **PD** constitute the results of a detection and express deadlock occurrence and a set of deadlocked processes, respectively.

- **Liveness** (progress): Each execution of the algorithm terminates in finite time.
- **Safety** (consistency): Let t_b and t_e be time moments of detection initiation and termination, respectively
  - 1. If the algorithm terminates with *dd* = *false*, then for any set **B** such that $\mathbf{B} \subseteq \mathbf{P}$, the predicate *deadlock*(**B**) was *false* at the time t_b.
  - 2. If the algorithm terminates with *dd* = *true*, then the predicate *deadlock*(**PD**) is *true* at the time t_e; moreover, for any set **B** such that $\mathbf{B} \subseteq \mathbf{P}$ and *deadlock*(**B**) at t_b, we have $\mathbf{B} \subseteq \mathbf{PD}$.

3.2 Informal description

A *control* process C_i, called *controller*, is associated with each application process P_i. Its role is, on the one hand, to observe the behavior of P_i and, on the other hand, to cooperate with other controllers C_j in order to consistently detect, if any, deadlock occurrence. (In general, controllers need not be separated as special processes since their tasks can be incorporated into application processes using the superposition rules described e.g. in [6]. Thus, the separation of controllers is merely a matter of interpretation).

State variables

Each process P_i is endowed with a state variable $state_i$ whose value, readable by C_i, is *active* or *passive*; $passive_i$ will be used as a shortcut for the boolean value $state_i = passive$. Moreover, as announced in Section 2.1, the node associated with P_i is endowed with input buffer where messages arrived and not yet consumed are stored. Let us recall that $\mathbf{ARR}_i$ is the set comprising the senders of all messages arrived but not yet consumed by P_i. Controler C_i can atomically read $\mathbf{ARR}_i$.

Token and virtual ring

In order to detect deadlock a control process, say C_α, initiates at a time moment t_b a detection by sending a valued token to its next on a virtual ring covering all the controllers C_i. The token carries a set **PD** of processes which, according to present knowledge, are potentially deadlocked since the first visit of the token relatively to this detection; initially **PD=P.** Controller C_i can only remove P_i from **PD** when it has the token. So the token moves around the virtual ring and when C_α notices that the value of the set **PD** has not changed since the previous visit, it concludes that set **PD** of processes is deadlocked (*dd* is set to *true* by C_α); if it notices $\mathbf{PD} = \emptyset$ it concludes there was not deadlock at the moment t_b of detection launching (*dd* is set to *false* by C_α).

The basic test

The core of the algorithm is the rule C_i has to follow to execute $\mathbf{PD} := \mathbf{PD} \setminus \{P_i\}$. Generally, controller C_i removes P_i from **PD** when, according to its present knowledge, P_i was certainly not deadlocked at time t_b, i.e., when at any time $t \geq t_b$ the process was active or *potentially active* (could be activated by known not deadlocked processes).

First, the activation of P_i, even temporary, can be locally detected by controller C_i and recorded by triggering boolean variable cp_i (continuously passive) to *false*. More precisely, at any time after the first token visit, relative to the current detection, the control variable cp_i is *true* if and only if C_i has observed that P_i was remaining continuously passive since the previous visit of the token. Second, let us note, that passive process P_i is potentially active during token visit, when $fulfilled_i(\mathbf{ARR}_i \cup (\mathbf{P} \setminus \mathbf{PD}))$ is *true*, as the set $\mathbf{ARR}_i \cup (\mathbf{P} \setminus \mathbf{PD})$ represents the set of the processes that sent or could potentially send messages to P_i.

Channel states

In order to be consistent the detection must not forget in-transit messages, i.e., messages sent and not yet arrived, as arrival of such messages could possibly give the value *true* to the predicate *fulfilled*$_i$. To solve this problem one can utilize an acknowledgment mechanism realized usually by underlying communication system, and observe whether all sent messages have been acknowledged. If this mechanism is not available, then each controller has to acknowledge every message as soon as a message has arrived and to count the number of messages sent by P_i and not yet acknowledged; a variable *notack*$_i$ initialized to *0* is used for this purpose. Now, if P_i is potentially deadlocked when C_i receives the token (condition $P_i \in \mathbf{PD}$ and cp_i), C_i keeps the token until either new arrival of messages will give the value ***true*** to the predicate *fulfilled*$_i$ (and cp_i will become *false)* or *notack*$_i$ will be reduced to zero (channels will be empty). This ensures that messages sent by P_i have arrived and so they are ready to be consumed by their destination processes.

First token turn

Finally, the token carries a second field *fv*, a boolean value, indicating whether it is, or not, the first turn of the token relatively to this detection. The first turn is used to properly initialize the boolean variable cp_i indicating that P_i has been continuously passive since the previous visit of the token, and to guarantee that outgoing channels of passive processes are empty before the next turn of the token.

3.3 Formal description

The local variables used by each C_i have been introduced in the previous sections. The behavior of C_i is described by the following statements S1 to S5. All the statements are executed atomically except the one including ***keep the token*** which can be interrupted whilst its condition is *false*.

```
S1:   when $P_i$ sends message to any $P_j$
          $notack_i := notack_i + 1$

S2:   when message arrives from any $P_j$
          send $ack$ to $C_j$

S3:   when $C_i$ receives $ack$ message from any $C_j$
          $notack_i := notack_i - 1$

S4:   when $P_i$ becomes active
          $cp_i := false$

S5:   when $C_i$ receives $token(\mathbf{PD}, fv)$
          if $P_i \in \mathbf{PD}$ then
              if $fv$ then $cp_i := (state_i = passive)$ fi;
              keep the token until $(\neg cp_i) \vee fulfilled_i(\mathbf{ARR}_i \cup (\mathbf{P} \setminus \mathbf{PD})) \vee (notack_i = 0)$;
              if $(\neg cp_i) \vee fulfilled_i(\mathbf{ARR}_i \cup (\mathbf{P} \setminus \mathbf{PD}))$ then $\mathbf{PD} := \mathbf{PD} \setminus \{P_i\}$ fi;
```

```
        $cp_i := (state_i = passive)$
    fi;
    send token (PD, fv) to next
```

The controller C_α initiating deadlock detection executes the previous S1, S2, S3 and S4 statements with $i = \alpha$. Moreover it executes the following statements S6 and S7. The result of detection is recorded by the boolean variable *dd* (deadlock flag).

```
S6:   when controller $C_\alpha$ decides to check deadlock occurrence
          PD:= P;
          $fv := true$;
          $previous := |PD|$;
          send token (PD, fv) to next

S7:   when initiator $C_\alpha$ receives token(PD, fv)
            if $P_\alpha \in$ PD then the same behavior as the corresponding part in S5 fi;
            if $(fv \vee previous \neq |PD|) \wedge (|PD| \neq 0)$
               then
                  $fv := false$;
                  $previous := |PD|$;
                  send token (PD, fv) to next
               else
                  $dd := (PD \neq \emptyset)$
            fi;
```

3.4 Performance analysis

It is not hard to note that the number of token transmissions is equal to $n(n\text{-}1)$ in the worst case, where $n=|\mathbf{P}|$. The maximum deadlock detection delay is the same. Each controller requires only an acknowledgment counter and a boolean flag cp_i. The size of token is fixed and equal to $(n+1)$ bits, as one bit per process is enough.

3.5 Proof of the algorithm

In order to prove correctness of the algorithm we have to show its liveness and safety properties as stated in Section 3.1.

3.5.1 Notation

Considering one execution of the algorithm, let us denote by t_i^k the time moment at the end of the k-th visit of the token at the controller C_i (i.e., just before the token leaves C_i). Consequently, t_α^k denotes the end of the k-th token's visit at the initiator C_α, i.e., the time when k-

th token turn terminates. We have $t_i^k < t_i^{k+1}$. Also, for any entity X (variable, predicate, etc) and any time t, $X[t]$ will denote the value of X at time t. In particular, $\mathbf{PD}[t_i^k]$ denotes the value of set **PD** contained in the token at the end of its k-th visit at controller C_i. Moreover, we will use the following definition:

$cont\_pas_i(t, t') \equiv (t \leq t') \wedge (\forall \sigma : t \leq \sigma \leq t' : passive_i[\sigma])$

3.5.2 Liveness

Theorem 1:

If the detection algorithm is initiated at time t_b, then it will stop a finite time after t_b.

Proof. When the token visits controller C_i, either it is sent immediately to the next (if $P_i \notin \mathbf{PD}$) or it is kept by C_i until one of the three conditions holds:

- $\neg cp_i$
- $fulfilled_i(\mathbf{ARR}_i \cup (\mathbf{P} \backslash \mathbf{PD}))$
- $notack_i=0$

But ($cp_i \wedge notack_i \neq 0$) cannot hold indefinitely, since a passive process cannot send messages, and all messages sent are acknowledged in finite time. Thus, the condition under which controller C_i releases the token will hold eventually.

Consequently, each complete turn of the token is accomplished in finite time. The next turn k+2 is launched provided that $|\mathbf{PD}\ [t_\alpha^k]| > |\mathbf{PD}\ [t_\alpha^{k+1}]|$. Thus, the non-negative function $k \rightarrow |\mathbf{PD}\ [t_\alpha^k]|$ is monotonically decreasing, with initial value $|\mathbf{P}|$, whence the number of turns is finite.

Q.E.D.

3.5.3 Safety

Theorem 2:

Let t_b and t_e be the time moments of the detection initiation and termination, respectively.

i. If the algorithm terminates with *dd=false*, then for any set **B** such that $\mathbf{B} \subseteq \mathbf{P}$ the predicate *deadlock*(**B**) was *false* at time t_b.

ii. If the algorithm terminates with *dd=true*, then the predicate *deadlock*(**PD**) is true at time t_e; moreover, for any set **B** such that $\mathbf{B} \subseteq \mathbf{P}$ and *deadlock*(**B**) was *true* at time t_b, we have: $\mathbf{B} \subseteq \mathbf{PD}$.

Proof of point *i*. We will prove the equivalent implication:

(1) $(\exists \mathbf{B}:\mathbf{B} \subseteq \mathbf{P}: deadlock(\mathbf{B})\ [t_b]) \Rightarrow dd[t_e]$

According to the assumption, at time t_b we have: $\mathbf{B} \neq \emptyset$ and *deadlock*(**B**). By construction initially **PD** is equal to **P** and thus at t_b $\mathbf{B} \subseteq \mathbf{PD}$. As *deadlock*(**B**) is a stable property, we have:
$\forall \tau: \tau \geq t_b: deadlock(\mathbf{B})\ [\tau]$ whence, according to the definition of predicate *deadlock*:
$\forall \tau: \tau \geq t_b: (\forall P_i: P_i \in \mathbf{B}: passive_i[\tau] \wedge \neg fulfilled_i(\mathbf{NE}_i[\tau] \cup \mathbf{ARR}_i[\tau] \cup \mathbf{P \backslash B}))$

($\mathbf{NE}_i$ has been defined in Section 2.4; it denotes the set of all P_j such that $\neg$*empty(j, i)*)

All processes P_i belonging to **B** are continuously passive since time t_b and, hence, since the first visit of the token. We are going to show, by contradiction, that no process belonging to **B** can be removed from **PD**. Suppose, thus, this is not the case; let P_i be the first process of **B** which can be removed from **PD**. Thus, at time $t > t_b$ just before this possible removal, the relation $\mathbf{B} \subseteq \mathbf{PD}$ holds. The continuously passive process P_i can be removed from **PD** provided that the following condition is satisfied at t:
$fulfilled_i(\mathbf{ARR}_i[t] \cup \mathbf{P \backslash PD}\ [t]) = true$. But it is easy to note that:
$\mathbf{ARR}_i[t] = \mathbf{ARR}_i[t_b] \cup \{$processes whose messages arrived at P_i between t_b and $t\}$

Let us consider a message arrived at P_i between t_b and t. Its sender either belongs to **P\B** or it is a process belonging to **B** whose outgoing channel towards P_i was not empty at time t_b. Thus,
$\mathbf{ARR}_i[t] \subseteq \mathbf{ARR}_i[t_b] \cup \mathbf{NE}_i[t_b] \cup \mathbf{P \backslash B}$. Hence, as by hypothesis
$fulfilled_i(\mathbf{ARR}_i[t] \cup \mathbf{P \backslash PD}\ [t]) = true$, we have:
$fulfilled_i(\mathbf{ARR}_i[t_b] \cup \mathbf{NE}_i[t_b] \cup \mathbf{P \backslash B} \cup \mathbf{P \backslash PD}\ [t]) = true$
As $\mathbf{B} \subseteq \mathbf{PD}\ [t]$, this reduces to: $fulfilled_i(\mathbf{ARR}_i[t_b] \cup \mathbf{NE}_i[t_b] \cup \mathbf{P \backslash B}) = true$.
The last statement contradicts the assumption that *deadlock*(**B**) is *true* at time t_b. Thus, no process P_i belonging to **B** can be removed from **PD** and this proves point *i*. Moreover, when the algorithm terminates with *dd* = *true*, we have:

(2) $\mathbf{B} \neq \emptyset$ and $\mathbf{B} \subseteq \mathbf{PD}$

Proof of point *ii*. According to the construction, the algorithm terminates at $t_e = t_\alpha^{k+1}$ with result *dd=true* if and only if $|\mathbf{PD}\ [t_\alpha^{k+1}]| = |\mathbf{PD}\ [t_\alpha^k]|$ and $\mathbf{PD}\ [t_\alpha^{k+1}] \neq \emptyset$. Let σ^k be a time moment such that $\max\ \{t_i^k\} \leq \sigma^k \leq \min\ \{t_i^{k+1}\}$. A time moment σ^k exists as it can be equal to t_α^k, for instance. We have, in particular,
$\forall P_i: P_i \in \mathbf{P}: t_i^k \leq t_\alpha^k \leq \sigma^k \leq t_i^{k+1} \leq t_\alpha^{k+1}$
We note that, since the set **PD** can only decrease, then
$\forall P_i: P_i \in \mathbf{P}: \mathbf{PD}[t_i^k] \supseteq \mathbf{PD}[t_\alpha^k] \supseteq \mathbf{PD}[\sigma^k] \supseteq \mathbf{PD}[t_i^{k+1}] \supseteq \mathbf{PD}\ [t_\alpha^{k+1}]$
Thus, if algorithm terminates at $t = t_\alpha^{k+1}$ with *dd* = *true*, then
$\forall P_i: P_i \in \mathbf{P}: \mathbf{PD}[t_\alpha^k] = \mathbf{PD}[\sigma^k] = \mathbf{PD}[t_i^{k+1}] = \mathbf{PD}[t_\alpha^{k+1}]$
Let **PD** denote this set. We will prove that *deadlock*(**PD**) holds at time $t = \sigma^k$, i.e. the following conditions hold at time σ^k:

(C1) $\mathbf{PD} \neq \emptyset$ *and*

(C2) $\forall P_i{:}P_i \in \mathbf{PD}{:}\ passive_i[\sigma^k]$ *and*

(C3) $\forall P_i{:}P_i \in \mathbf{PD}{:}\ \neg fulfilled_i(\mathbf{NE}_i\,[\sigma^k] \cup \mathbf{ARR}_i\,[\sigma^k] \cup \mathbf{P} \backslash \mathbf{PD})$

We show step by step that these three conditions are directly implied by detection termination with *dd* = *true.*

- By construction detection terminates with *dd* = *true* only if $\mathbf{PD} \neq \emptyset$. The condition (C1) for deadlock occurrence at time σ^k is thus verified.

- Let $P_i \in \mathbf{PD}$. By construction, only processes continuously passive since the first visit of the token at t_i^1 can belong to **PD**. Hence, we have $cont\_pas_i\,(t_i^1, t_i^{k+1})$ and $passive_i\,[\sigma^k]$. The second condition (C2) for deadlock occurrence at time σ^k is thus verified.

- Since $P_i \in \mathbf{PD}$, we have by construction $\neg fulfilled_i\,(\mathbf{ARR}_i\,[t_i^{k+1}] \cup \mathbf{P} \backslash \mathbf{PD})$; otherwise P_i would be removed from **PD**. But as $\mathbf{ARR}_i$ can only increase and $\mathbf{P} \backslash \mathbf{PD}$ is constant in time interval $(t_\alpha{}^k, t_i{}^{k+1})$, we have:

  (3) $\mathbf{ARR}_i[\sigma^k] \cup \mathbf{P} \backslash \mathbf{PD} \subseteq \mathbf{ARR}_i[t_i{}^{k+1}] \cup \mathbf{P} \backslash \mathbf{PD}$

  By construction all output channels of processes belonging to **PD** are empty at time t_α^k as the token is kept by each controller associated with passive process belonging to **PD,** until all sent messages are acknowledged ($notack_i$ = 0). Because these processes are continuously passive since the first token visit, their output channels are empty also at time σ^k. Hence:
  $\mathbf{NE}_i\,[\sigma^k] \cap \mathbf{PD} = \emptyset$ and therefore $\mathbf{NE}_i\,[\sigma^k] \subseteq \mathbf{P} \backslash \mathbf{PD}$. Consequently:

  (4) $\mathbf{NE}_i\,[\sigma^k] \cup \mathbf{ARR}_i[\sigma^k] \cup \mathbf{P} \backslash \mathbf{PD} = \mathbf{ARR}_i[\sigma^k] \cup \mathbf{P} \backslash \mathbf{PD}$

  From (3) and (4), we obtain:

  (5) $\mathbf{NE}_i\,[\sigma^k] \cup \mathbf{ARR}_i[\sigma^k] \cup \mathbf{P} \backslash \mathbf{PD} \subseteq \mathbf{ARR}_i[t_i{}^{k+1}] \cup \mathbf{P} \backslash \mathbf{PD}$

  But from definition of predicate $fulfilled_i$ we have:
  if $\mathbf{X} \subseteq \mathbf{Y}$ and $\neg fulfilled_i\,(\mathbf{Y})$, then $\neg fulfilled_i\,(\mathbf{X})$. Thus, from this definition and from (5) we can conclude that if

  (6) $\neg fulfilled_i\,(\mathbf{ARR}_i[t_i{}^{k+1}] \cup \mathbf{P} \backslash \mathbf{PD})$,

  then $\neg fulfilled_i(\,\mathbf{NE}_i\,[\sigma^k] \cup \mathbf{ARR}_i[\sigma^k] \cup \mathbf{P} \backslash \mathbf{PD})$.
  By construction (6) is satisfied for each $P_i \in \mathbf{PD}$. Thus, the third condition (C3) for deadlock occurrence at time σ^k is verified.

The above points proved that *deadlock*(**PD**) is *true* at σ^k. As deadlock is stable property and $\sigma^k \leq t_e$, then predicate *deadlock*(**PD**) is also *true* at time t_e; that proves the first assertion of point *ii.*

Finally, if there is a set **B** such that *deadlock*(**B**)[t_b], then from properties (1) and (2) (see point *i*) we conclude that **B** $\subseteq$ **PD**. That proves the second assertion of point *ii* of the safety property.

Q.E.D.

3.6 Initiation of deadlock detection

For completeness of the proposed deadlock detection approach it is still necessary to precisely define when controllers should initiate deadlock detection (i.e., make up decision to check deadlock occurrence), in order to guarantee that every set of deadlocked processes will be detected in finite time. We will mention two possible solutions to this problem.

The first solution is a natural extension of the basic token-based algorithm. It assumes that token travelling along virtual ring is either *busy* or *free*. Token is *busy* when it is just involved (used) in deadlock detection which is controlled by the algorithm directly corresponding to the basic one. When the detection terminates, the initiator switches token to *free* and sends it to the next controller in the ring. The free token received by a controller C_i associated with active process P_i is sent further immediately; however, if P_i is passive, then C_i is obliged to switch token to *busy* and initiate in finite time subsequent deadlock detection.

The second solution of detection initiation problem is based on the rule requiring that controller initiates deadlock detection every time its application process becomes passive (immediately or after a finite delay). As a consequence several tokens can travel simultaneously in the system. To distinguish these tokens they can be endowed with the unique stamp *(j, s)* composed of the initiator identity *j* and *sequence number s* of token initiations, as in [7] for instance. In this context, the basic algorithm requires, of course, some extensions, to guarantee detection consistency. Thus, controller C_i is endowed with variables $latest_i(j)$ and $cp_i(j)$ which are associated with each possible initiator C_j. The variable $latest_i(j)$ is equal to the largest sequence number *s* of tokens initiated by C_j. The boolean variable $cp_i(j)$ denotes that P_i has been continuously passive since the last visit of the token initiated by C_j. The variable $cp_i(j)$ can be updated (set to *true)* only by the token with stamp *(j, s)*, such that $s \geq latest_i(j)$.

Embedding consistently one of the above mentioned ideas to the basic algorithm, we obtain algorithm call *extended* one.

Theorem 3:

The extended algorithm detects every set of deadlocked processes in finite time.

Proof. According to the proposed constructions, each deadlocked process entails an initiation of deadlock detection after it became passive the last time. For the second extension it is trivial as every passive process initiate detection whenever it becomes passive. For the first one it is also true because every detection is terminated in finite time as shows Theorem1, and then the free token is sent immediately to the next controller in the ring. Thus, the controller of a continuously passive (deadlocked) process will receive in finite time the *free* token, and then it will initiate detection.

Consequently, in both cases, the controller associated with a deadlocked process which becomes passive as the last one, will initiate detection in finite time. According to Theorem 1 and Theorem 2, this detection computation will detect all deadlocked processes infinite time.

Q.E.D.

It should be stressed that, according to classification of deadlock detection problems given in Section 2.5, the extended algorithm solves the problem of the maximum deadlocked set detection.

4 Refinements

4.1 Individual termination

As was introduced in Section 2.2, individually terminated process P_i is characterized by the following predicate:

$$state_i = passive \text{ } and \text{ } \mathbf{DS}_i = \varnothing$$

To take into account such individual terminations the definition of predicate *deadlock*(**B**) introduced in Section 2.4, has to be slightly modified. Let **T** be the set of individually terminated processes. As a process belonging to **T** cannot send messages which a passive process P_i is waiting for, we consider that P_i as deadlocked. Thus, for the *AND* request model the predicate *deadlock* (**B**) becomes (for other models a similar modification has to be done):

$$\begin{aligned}deadlock\,(\mathbf{B}) \equiv (\mathbf{B} \neq \varnothing) \wedge (\forall P_i\!: P_i \in \mathbf{B}\!:\\ (passive_i \wedge (\exists P_j\!: P_j \in \mathbf{DS}_i \cap (\mathbf{B} \cup \mathbf{T})\!:\\ (empty\,(j,\, i) \wedge \neg\, arr_i\,(j)))))\end{aligned}$$

Let us note that with this definition a circuit in the classical *wait-for-graph* representation (which can be associated with *AND* request model) is no more a necessary condition for deadlock occurrence.

In the proposed extension of the basic algorithm, token carries an additional field **IT** representing the set of individually terminated processes. When a controller C_i receives the token it simply adds P_i to **IT** if P_i is individually terminated, and proceeds as previously (of course the predicate *fulfilled*$_i$ evaluates to *false* for every individually terminated process P_i). So **PD** includes the set of potentially deadlocked or individually terminated processes from the token viewpoint. At the end of the detection **PD** \ **IT** defines the set of deadlocked processes.

4.2 Termination detection

The termination detection problem (global termination) consists in detecting a state from which there is no more activity in the program execution ([9],[10]). A lot of particular algorithms have been proposed to solve it (see e.g., [19]). This problem is in fact equivalent to one presented in Section 2.5.3, namely: compute a set **B** of deadlocked processes in which the set **B** is predefined and equal to the set **P** of all processes. Thus, the set **B** is now fixed and known *a priori*, and the question is: are all processes globally terminated? Consequently the answer is *yes* or *no.*

To solve the termination detection problem one can use the basic algorithm proposed here with the following slight modification: the last statement of S7 (i.e., $dd := (\mathbf{PD} \neq \emptyset))$ has to be augmented by $td := (\mathbf{PD} = \mathbf{P})$, where *td* is a boolean variable indicating whether the global termination occurred. If individual termination of processes is also permitted, the additional instruction should be the same ($td := (\mathbf{PD} = \mathbf{P})$) as $\mathbf{IT} \subseteq \mathbf{PD}$. Moreover, as soon as a controller C_i removes P_i from **PD** the answer will be $td = false$, hence the algorithm can be easily further modified to take this into account and answer *no* quicker.

Finally, let us emphasize that the obtained termination detection distributed algorithm allows non FIFO channels and unspecified receptions. From practical point of view, that constitutes advantageous characteristics when this solution is compared with other termination detection algorithms.

4.3 Token routing

Analyzing the basic algorithm described in Section 3, one can easily note that token management leaves opportunity for improvement. First of all, number of token transmissions can be significantly reduced, directing token only to the controllers associated with processes composing the set **PD**, or to the initiator controller C_α. It can be realized introducing notion *next(***PD***)* meaning *next* along the ring from controllers C_i, such that $P_i \in \mathbf{PD} \cup \{P_\alpha\}$, and substituting *next* by *next*(**PD**). Then, at most $(n+2)(n-1)/2$ transmissions of token are needed, in the worst case.

If we are interested only in detection of a deadlocked process (see Section 2.5.2), then we can apply the following modification to the basic algorithm, resulting in the algorithm which is more efficient with respect to detection delay. The general idea is to change the first turn of the token, taking into account that activation of any process P_i depends directly only on processes belonging to its dependent set $\mathbf{DS}_i$. This observation has been utilized successfully in many distributed algorithms detecting a deadlocked process (see e.g., [3], [7]), Thus, the proposed modification is as follows. During the modified first turn token carries, apart from **PD** and *fv*, a set **V** of processes already visited in this turn. The controller C_α, that wants to know if P_α is deadlocked, behaves as initiator and before the first token launching, sets $\mathbf{PD} := \mathbf{DS}_\alpha \cup \{P_\alpha\}$, $\mathbf{V} := \emptyset$, and then sends the token to *next*(**PD**). Controllers C_i associated with processes active during token visit set $\mathbf{PD} := \mathbf{PD} \setminus \{P_i\}$, $\mathbf{V} := \mathbf{V} \cup \{P_i\}$, and send the token to *next*($\mathbf{PD} \setminus \mathbf{V}$). On the other hand, each controller associated with passive process P_i, sets $\mathbf{PD} := \mathbf{PD} \cup (\mathbf{DS}_i \setminus \mathbf{V})$ and $\mathbf{V} := \mathbf{V} \cup \{P_i\}$ (we subtract **V** from $\mathbf{DS}_i$ because processes already visited in this turn and removed from

PD as active need not be visited again). Then, C_i sends the token to *next*(**PD** \ **V**). When the modified first turn terminates, one can continue detection, launching the token with *fv* = *true* and **PD** initiated to value obtained as a result of the modified turn, and control further detection according to the basic algorithm. The above outlined approach seems to be very efficient in all cases where only a few processes are involved in a deadlock. In this context it is worth mentioning theoretical and empirical studies showing that for most distributed database applications over 90% of deadlocks involve only two processes (see e.g. [1], [18]).

4.4 Parallel detection execution

All proposition presented here till now apply one token in every detection computation. Hence, detection process is in fact sequential, and therefore detection delay is O(n^2) in the worst case. However, this disadvantage can be overcome using the following parallel scheme.

Let us assume that the set **P** of processes is fully connected. Thus, virtual *star* structure with the initiator C_α as the center node is available. The initiator begins detection broadcasting *query* message to all controllers (including itself). The *query* message is endowed with set **PD** and flag *fv*, like the token in the basic algorithm. Receiving a query, controller C_i executes operation according to the basic algorithm, but it sends *reply* message with updated **PD** = $\mathbf{PD}_i$ directly to the initiator. Controller C_α collects all *reply* messages and constructs globally updated set **PD** with the following assignment:

$\mathbf{PD} := \mathbf{PD}_1 \cap \mathbf{PD}_2 \cap ... \cap \mathbf{PD}_n$

Then, depending on termination conditions equivalent to those of the basic algorithm, the next *query- reply* exchange is eventually initiated.

The correctness of the above parallel algorithm results directly from the observation that a sequential order in which controllers are visited by the token is not material provided that every full turn is completed before initiating the next one. This essential property, together with conditions under which the visit is allowed to progress (expressed in the ring structure by construction like *keep the token until*) has been abstracted as the "guarded wave sequence" concept [15]; this concept is a tool for methodological design of algorithms detecting stable properties (in particular, deadlock detection). Thus, any process structure allowing to implement the concept of wave (star, more general tree, ring, etc) can be also easily applied.

Let us note, that application of the parallel algorithm to the star structure reduces deadlock detection delay to O(n) in the worst case.

5 Conclusion

In this paper deadlock detection in asynchronous message communication systems has been considered. It is a very important problem for various applications including computer networks, database systems, massively parallel systems, etc. In distributed asynchronous environment deadlock detection is peculiarly subtle and complex, as distributed algorithms are required and no node has accurate knowledge of the whole system state. These difficulties brought about a large number of errors in published deadlock detection algorithms. To reduce as much as possible this danger, the paper has first introduced formal deadlock models and hierarchy of deadlock detection problems. Then, a general algorithm detecting sets of deadlocked processes has been proposed, described formally and proven correct. This algorithm has used token passing mechanism, however, it has been shown, that other more parallel mechanisms can also be applied to improve efficiency of the detection computation. Moreover, some extensions of the algorithm have been outlined to cover individual and global termination detection problem, and to improve token management. The very important and advantageous characteristic of the proposed algorithm is possibility of its straightforward application in distributed systems even with unspecified receptions, not *FIFO* channels, and general request models permitting, among others, *AND*, *OR*, *AND-OR*, *k-out-of-n* requests. Thus, the solution presented here has a superiority over the deadlock detection algorithms considered up to now.

References

[1] Bernstein, P.A., Hadzilacos V., Goodman N., *Concurrency Control and Recovery in Database Systems,* Addison Wesley, Reading, Mass, 1987, 370 pages.

[2] Blazewicz J., Brzezinski J., Gambosi G., Time-stamp approach to store-and-forward deadlock prevention, *IEEE Trans. on Comm.,* vol. COM-35,5, 1987, pp. 490-495.

[3] Bracha G., Toueg S., Distributed deadlock detection, *Distributed Computing,* vol. 2,3, 1987, pp. 127-138

[4] Chandy K.M., Lamport L., Distributed snapshots: determining global state of distributed systems, *ACM Trans. on Comp. Systems,* vol. 3,1, 1985, pp. 63-75

[5] Chandy K.M., Misra J., A distributed algorithm for detecting resource deadlock in distributed systems, *Proc. of ACM Symposium on Principles of Distributed Computing,* ACM, New York, 1982, pp. 157-164.

[6] Chandy K.M., Misra J., *Parallel Program Design: a Foundation,* Addison Wesley, 1988, 516 pages.

[7] Chandy K.M., Misra J., Hass L.M., Distributed deadlock detection,*ACM Trans. on Comp. Systems.,*vol. 1,2, 1983, pp. 144-156.

[8] Coffman E,G.Jr, Elphick M.J., Shoshani A., System deadlock, *ACM Computing Surveys,* vol. 3,2, 1971, pp. 66-78

[9] Dijkstra E.W.D., Scholten C.S., Termination detection for diffusing computation, *Inf. Proc. Letters,* vol. 13,1, 1980, pp. 1-4.

[10] Francez N., Distributed termination, *ACM Trans. on Progr. Lang. and Systems.,* vol. 2,1, 1980, pp. 42-55.

[11] Gifford D.G., Weighted voting for replicated data, *Proc. of the 7th ACM symposium on Operating Systems Principles,* ACM, New York, 1979, pp. 150-163.

[12] Gligor V., Shattuck S., On deadlock detection in distributed databases, *IEEE Trans. Soft. Eng.,* vol. SE-6,5, 1980, pp. 435-440.

[13] Gunther K.D., Prevention of deadlock in packet-switched data transport system, *IEEE Trans. on Comm.*, vol. COM-29,4, 1981, pp. 512-524.

[14] Helary J.M., Jard Cl., Plouzeau N., Raynal M., Detection of stable properties in distributed systems, *Proc. 6th ACM Symposium on Principles of Distributed Computing*, ACM, New York, 1987, pp. 289-285.

[15] Helary J.M., Raynal M., Distributed evaluation: a tool for constructing distributed detection programs, *Proc. Symp. on Theory of Computing and Systems, LNCS 601*, Springer-Verlag, 1992, pp. 184-194.

[16] Holt R.C., Some deadlock properties of computer systems, *ACM Computing Surveys*, vol. 4,3, 1972, 179-196.

[17] Jaffe J.M., Sidi M., Distributed deadlock resolution in store-and-forward networks, *Algorithmica*, vol. 4,3, 1989, pp. 417-436.

[18] Knapp E., Deadlock detection in distributed databases, *ACM Computing Surveys*, vol. 19,4, 1987, pp. 303-328.

[19] Mattern F., Algorithms for distributed termination detection, *Distributed Computing*, vol. 2,3, 1987, pp. 161-175.

[20] Mitchell D., Merritt M.J., A distributed algorithm for deadlock detection and resolution, In *Proc. of the 3rd ACM Symposium on Principles of Distributed Computing*, ACM, New York, 1984, pp. 282-284.

[21] Natarajan N., A distributed scheme for detecting communication deadlocks, *IEEE Trans. Soft. Eng.*, vol. SE-12,4, 1986, pp. 531-537.

[22] Singhal M., Deadlock detection in distributed systems, *IEEE Computer*, vol. 22,11. 1989, pp. 37-48.

LISTE DES DERNIERES PUBLICATIONS INTERNES PARUES A L'IRISA

PI 672 ORDRES REPRESENTABLES PAR DES TRANSLATIONS DE SEGMENTS DANS LE PLAN
Vincent BOUCHITTE, Roland JEGOU, JeanXavier RAMPON
Juillet 1992, 8 pages.

PI 673 AN EXCEPTION HANDLING MECHANISM FOR PARALLEL OBJECT-ORIENTED PROGRAMMING
Valérie ISSARNY
Août 1992, 36 pages.

PI 674 A CALCULUS OF GAMMA PROGRAMS
Chris HANKIN, Daniel LE METAYER, David SANDS
Juillet 1992, 32 pages.

PI 675 EVALUATION DES PERFORMANCES D'UN NOYAU DE SIMULATION REPARTIE
Philippe INGELS, Carlos MAZIERO
Septembre 1992, 36 pages.

PI 676 FONT METRICS
Jacques ANDRE
Septembre 1992, 20 pages.

PI 677 GRIF ET LES INDEX ELECTRONIQUES
Hélène RICHY
Septembre 1992, 40 pages.

PI 678 ETUDE DE QUELQUES ORGANISATIONS D'ANTEMEMOIRES
Nathalie DRACH, André SEZNEC
Octobre 1992, 44 pages.

PI 679 AN ADAPTIVE SPARSE UNSYMMETRIC LINEAR SYSTEM SOLVER
Miloud SADKANE, Roger B. SIDJE
Octobre 1992, 28 pages.

PI 680 BRANCHING BISIMULATION FOR CONTEXT-FREE PROCESSES
Didier CAUCAL, Dung HUYNH, Lu TIAN
Octobre 1992, 36 pages.

PI 681 DEADLOCK MODELS AND GENERAL ALGORITHM FOR DISTRIBUTED DEADLOCK DETECTION
Jerzy BRZEZINSKI, Jean-Michel HELARY, Michel RAYNAL
Octobre 1992, 26 pages.

ISSN 0249-6399