Banker's Algorithm Implementation in CPN Tools

Michal Žarnay[1]

University of Žilina, Univerzitná 8215/1, SK-01026 Žilina, Slovak Republic.
michal.zarnay@fri.uniza.sk

Abstract. When constructing discrete simulation models of complex transportation systems, their designers face problems of deadlock states occurring in the course of simulation. When analyzing it, the issue was transformed to a problem of solving deadlock states in resource allocation systems (RAS) with non-sequential processes with flexible routing and use of resources of multiple types at once. As a suitable deadlock-avoidance policy, the banker's algorithm (BA) has been chosen. The task was to modify the basic version of the BA and to test the developed algorithm on a sample transportation system with the outlined properties. As a suitable environment for this, the CPN Tools were chosen, what led to an implementation of the modified version of the BA in the CPN ML, language used by the CPN Tools. The paper explains modifications of the algorithm, describes an implementation of it in the CPN ML and shows its use on a coloured Petri net model of a small example from the outlined category of the RAS.

1 Introduction

Motivation for this work came from the field of detailed computer simulation of complex transportation systems, such as railway marshalling yard processing a few thousands of wagons per hour in trains of various technological processing descriptions with help of over one hundred resources (individual tracks, locomotives, members of personnel). From experience with real projects, main technological processes in complex systems are usually clearly defined, however, there are often little details complicating the models and causing that designers of models face problems of deadlock states occurring in the course of simulation.

Deadlock state is a state of a system, where two or more system processes are blocked in their execution because they wait for two or more resources, and the awaited resources are at the same time occupied by the processes included in the waiting list. The waiting processes thus block and are blocked. Unblocking this state is possible only by an exceptional operation.

When analyzing the issue, we learnt that it is similar to solving of deadlock states in other fields like flexible manufacturing systems, and that it has been tackled in literature for many years. However, none of the proposed solutions seemed to be adequate for this problem. Further analysis in [1] transformed the issue to a problem of solving deadlock states in resource allocation systems

(RAS) with non-sequential processes with flexible routing and use of resources of multiple types at once.

As a suitable deadlock-avoidance policy for such a system, the banker's algorithm (BA) has been chosen [1]. The task was to modify the basic version of the BA and to test the developed algorithm on a sample transportation system with the outlined properties. From our literature review, we are not aware of any use of the BA in a RAS combining flexible routing with concurrent processing. As a suitable environment for this, the CPN Tools were chosen: to construct a CPN model for the sample system and to implement an adjusted version of the BA. Reasons of this decision lie in the abilities of Petri nets and CPN Tools to construct a model of a RAS quicker compared to other means, e.g. high-level programming languages, and to facilitate a qualitative analysis of the model with and without the BA for testing its effectiveness in deadlock avoidance. In the end, the algorithm itself has been implemented in the CPN ML, language used by the CPN Tools.

From the complex work, this paper focuses on explaining basic modifications of the BA, describing an implementation of it in the CPN ML and showing its successful use on a coloured Petri net model. For the illustrative model, however, only a theoretical example from the outlined category of the RAS has been included, since the model of a sample transportation system developed in [1] and briefly described in [2] is too complex to serve as a basis for showing the algorithm.

The paper is organised as follows. In the section 2, we introduce background for the paper about resource allocation systems and their modelling in Petri nets, and the banker's algorithm and our modifications for the studied system. The section 3 describes the implementation of the modified algorithm in the CPN ML, its integration with the CPN model and its analysis results. That is followed by discussing main contributions and issues of the work in the conclusion.

2 Starting Point

2.1 Resource Allocation System in Petri Net

Resource allocation system (RAS) is a system consisting of concurrently running processes that in certain stages, in order to get successfully completed, require an exclusive use of certain number of system resources [3]. Resources are limited and re-usable as their allocation and de-allocation changes neither their character nor quantity. Based on its character, a process in the RAS is *sequential* or *non-sequential*, i.e. some stages of the process run concurrently. The resulting system is then either sequential (if all involved processes are sequential) or non-sequential (if at least one process in the system is non-sequential). Furthermore a process may contain *flexible routing*, which means that in a certain moment, a process execution continues in one of available options and if correctly defined, taking any of them brings the process to the same final state. Finally, the number and the type of resources allocated at the same time distinguish between

a *single-unit* RAS (every process is allowed to have only one resource unit allocated at a time, i.e. before allocation of the next resource, the previous must be returned), a *single-type* RAS (at least one process has two or more units of the same resource type at a time) or a *multiple-type* RAS (at least one process has two or more units from two or more resource types at a time). The outlined attributes create categories of the RAS with varied complexity.

The system used in this paper is a non-sequential multiple-type RAS with flexible routing. It means that at least one process in the system combines flexible routing with concurrent execution. To distinguish modelling elements of both properties, we use the following naming convention in the paper:

- Variant – one of possible routes of a process execution (sequential or non-sequential) that is chosen by flexible routing
- Branch – a part of a process that is executed concurrently with another part (branch) of the same process

For the modelling of a RAS, the coloured Petri net (CPN) is used. In the net, subnets of two types are found: a *process subnet* and a *resource subnet*. A *process subnet* consists of places, transitions and arcs in a structure starting by an initial transition and ending with a final transition and describing causal relations between stages of a process. A stage (a task) in the process corresponds to a place and events of beginning and ending of tasks correspond to transitions. Together with a place for idle processes (let's denote it $P0$), which connects the final transition with the initial transition of the process description, it makes a strongly connected component. The variants of the flexible routing in the process description are created, when at least one place has at least two output transitions (conflict in Petri net, like in state machines) and another place has at least two input transitions, while all possible routes in the process subnet contain the place $P0$. The branches of the concurrent processing are created, when a transition has at least two output places and another transition has at least two input places (synchronization in Petri net, like in marked graphs), while the place $P0$ is in a part of the subnet with no more than one branch. A *resource subnet* consists of one place and adjacent arcs. Content of the place represents actually free resources and arcs express their allocation and release to and from stages of processes. Typically, there are several process subnets, one for each modelled process, and one resource subnet in a RAS model.

For a description of the system's dynamic behaviour, we'll distinguish between process types and process instances. The *process type* is an abstract description of a process. The *process instance* is a concrete occurrence of a process according to a process type. In the CPN, the process type is modelled by the process subnet and the process instance by one or more tokens of one colour. A position of a token in a place of the process subnet represents a *stage* of the process and a combination of places occupied by tokens of the same value constitutes the actual *state* of the process instance. Similarly, there are resource types and resource instances. The former modelled by colours of a colour set for all resources (one resource type = one colour) and the latter by individual coloured

tokens (number of tokens of a colour = number of resources of the respective resource type).

The figure 1 depicts a CPN model of an illustrative RAS that we use in this paper. The system contains one non-sequential process type modelled by the process subnet. The places $P1, \ldots, P9$ represent individual activities (tasks) in the process. The place $P0$ represents the outside environment, where the finished process instances (tokens coming to the place $P0$) get replaced by new ones before their processing (tokens leaving the place $P0$). Transitions $T1, \ldots, T9$ interlink the activities to give them a logical structure.

An execution of the process type is divided into two branches from the initial transition $T1$, which continue until the final transition $T9$. The left-hand branch is further divided into two variants between the places $P1$ and $P8$. The left-hand variant of these has further 2 branches between transitions $T2$ and $T8$. The place $P0$ contains two tokens of different colours from the integer colour set $cProcessID$ modelling two process instances currently outside of the system. When an instance is in the system, it is represented by two or three tokens of the same colour, depending on the number of branches visited in the actual state. Movements of the coloured tokens are facilitated by the variable *proc* present on all the arcs in the process subnet.

The place *Resources* with adjacent arcs represents the resource subnet. The arcs define relations of allocation/release of resources to/from process activities. Nine tokens in the place *Resources* of 3 colours $R1, R2$ and $R3$ from the colour set $cResources$ represent nine available resource instances, three of each resource type.

2.2 Banker's Algorithm

The *banker's algorithm*, first introduced in [4], uses information about a current system state to decide, whether an allocation request of a process instance can be fulfilled. It is called every time, when an allocation request is made. It tries to find, whether the allocation leads to a *safe state*, i.e. a state, from which all running processes can finish their execution. If yes, the request can be fulfilled, otherwise, the requesting process instance must wait until another process instance returns resources. In order to decide, whether the state is safe, the BA tries to order all running process instances in such a sequence, so that each of them can be finished with resources that it has currently allocated or that are currently available in the system or that are returned from finished process instances already in the sequence prior to the tested process. If it succeeds in finding such a sequence, we say that the state is *ordered*, and since every ordered state is safe [5], the state is also safe. If it fails to find the sequence, we say that the state is *unordered*, which does not mean that the state is unsafe. However, the allocation request cannot be fulfilled. This is due to the suboptimality of the BA, while finding an optimal algorithm for solving the question about state safeness is a NP-hard problem.

The basic principle of the BA is described in a pseudo-code based on the Algorithm 1 from [5]. It uses the following data structures:

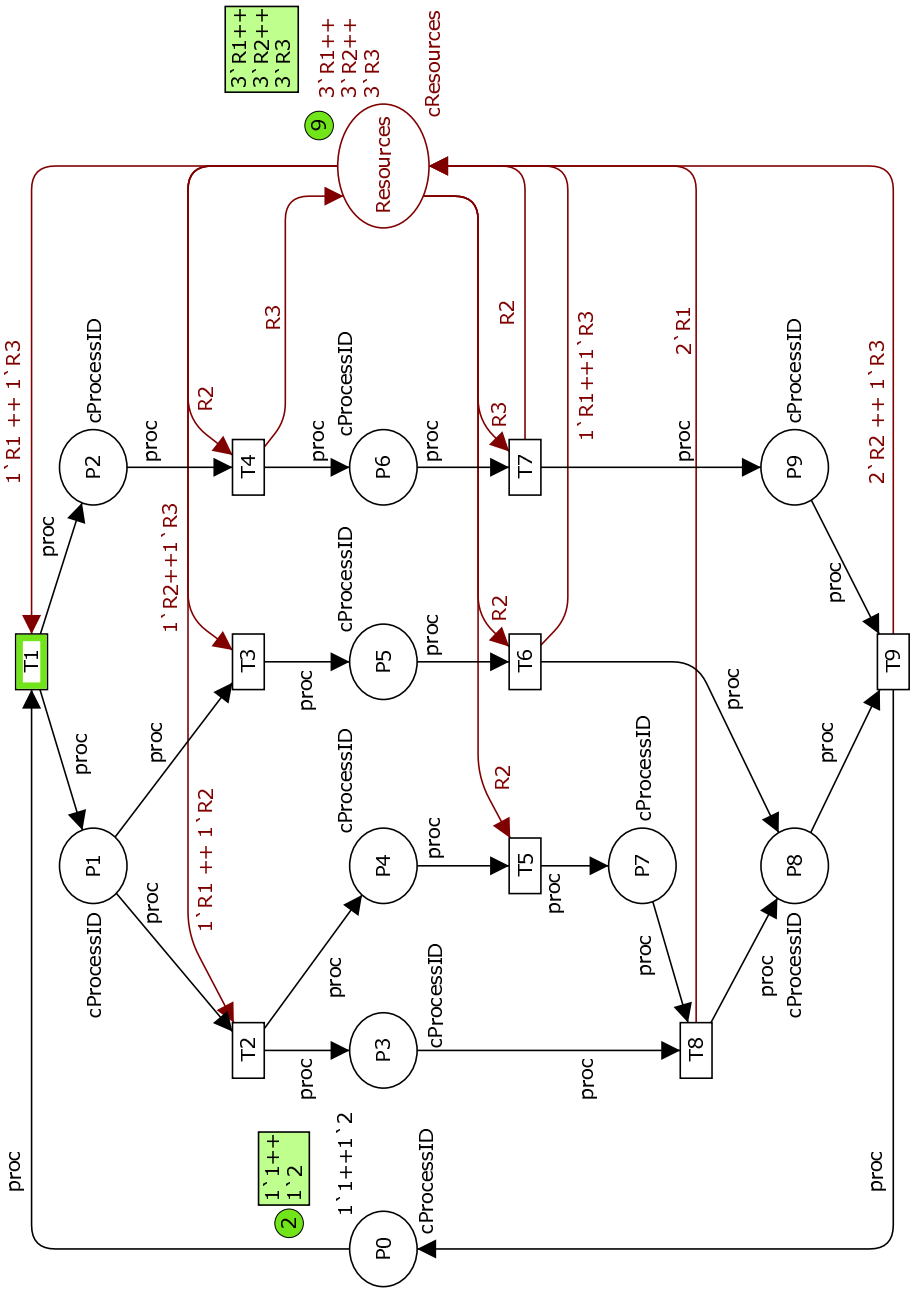

Fig. 1. CPN model of illustration RAS.

- *Allocated* – a matrix of allocated resource units to process instances
- $Allocated[i][j]$ – a number of resources of the kind j allocated to the process instance i
- $Needed$ – a matrix of needed resource units to finish execution of process instances
- $Needed[i][j]$ – a number of resources of the kind j needed by the process instance i
- *Available* – a vector of available resource units in the system
- $Available[j]$ – a number of available resources of the kind j
- S – a set of all process states currently being executed by process instances
- Π – a set of identification numbers of process stages in S
- R – a set of all resource types
- $p(i)$ – the i-th item in the vector of all running process instances p

When called, the algorithm uses data about a current state of the system in the data structures *Allocated*, $Needed$ and *Available* as input and provides an answer to the question *Is the given state ordered?* as output values *Admit* or *Reject*:

```
begin
            // Set of currently running process instances. //
   Π = {1, 2, ..., | S |};
   loop
            // If all instances have been ordered, admit the state. //
      if Π = ∅ then return Admit
            // Otherwise find a p(i) that can be added to the order. //
      else find i ∈ Π such that
         Needed[i][j] ≤ Allocated[i][j] + Available[j];
      if no such i exists, return Reject;
            // Otherwise, add p(i) to the order. //
      for j = 1 to | R | do begin
         Available[j] = Available[j] + Allocated[i][j];
         Allocated[i][j] = 0;
         Needed[i][j] = 0;
      end;
      Π = Π \ i
   end loop
end
```

The outlined algorithm has been tested for a sequential single-unit resource allocation system (RAS) with flexible routing [5].

The algorithm assumes that as each process instance enters the system, it declares the maximum number of units of each resource type needed for its execution. In a more elaborated version, supportive routines update the number of remaining needed resources of each type of the system processes based on descriptions of their execution and actual positions of process instances [5] [6] and [7].

Applications of the banker's algorithm described in the mentioned literature sources as well as in [8], [9] and [10] have been connected with the sequential RAS-s only. On the other side, authors in [11] consider a banker's-like deadlock avoidance policy for a RAS with non-sequential processes without flexible routing. Our literature review did not bring to our attention any application of the banker's algorithm or banker's-like deadlock avoidance policies that would treat a RAS combining both concurrent processing as well as flexible routing in a process.

2.3 Modifications of Banker's Algorithm

Data structures (the matrices *Allocated*, $Needed$ and the vector *Available*) used by the original version of the BA stay the same in our implementation except renaming the $Needed$ matrix to $RemainingNeeded$ to express more precisely that a content of the data structure is modified during a process instance execution. Resources that will not be requested any more, are subtracted from the vector after their allocation. Only those resources will stay recorded that will be requested in the remaining part of execution. This is based on the assumption that we know process descriptions and we know states, when values of the $RemainingNeeded$ vector are changed.

In [7], it is proposed to assign values of the $RemainingNeeded$ vector to individual process stages of a process on its route to the end of execution. This proposal is suitable for flexible routing, but not for concurrent processing of non-sequential processes. In such processes, the current process instance state may be represented by tokens in more than one place, where the places are in different branches. Since tokens in branches move concurrently, the order of their movements is non-deterministic and it is not possible to know the number of currently allocated and the number of remaining needed resources for each place (for the time, when a place is occupied by a token) generally for all possible executions. That's why we introduce vectors of *relative* changes (called $Change$) which modify the $RemainingNeeded$ vector for the current process instance according to the change in the allocation of resources in the related stage. If resources are being allocated, the $Change$ vector will contain a positive number of the allocated units, if released, the number will be negative. The $Change$ vector must be defined for each process stage with any allocation or release of resources. In addition, it is necessary to take into account, whether resource units will be allocated to the same process instance repeatedly, i.e. allocations and releases of a unit of the same resource type occur in disjunctive time periods at least two times till the end of the process instance execution. This is recorded in the $RemainingNeeded$ vector through values of the $Change$ vector distinguishing two groups of bits in the integer used by every *release of resource units*. An item of the $Change$ vector expresses a number of units of the relevant resource type that are to be allocated/released. When a value of the $Change$ vector contains a non-zero number encoded in the lower half of the bits (the bits 0-3 by 8-bit numbers), the corresponding units will be used again. When a non-zero number

is encoded in the upper half (the bits 4-7), the units won't be used again. For instance, if there are 2 units released without a planned re-allocation, the number will be -32. If they were both to be re-allocated later, the number will be -2, if only one should be re-allocated, the correct number will be -17. The outlined mechanism is relevant only for releases of resources. By allocations of resource units, only the lower half of bits is considered.

As for the division of bits, it is not necessary that the available bits are divided into two halves. It is important that the smallest used part of bits is enough to record the highest number of resource units allocated or released at once in the modelled system. The chosen bits are manipulated with help of operations divide ($/$) and modulo ($\backslash$) and of a relevant constant called $ByteDiv$ (for the discussed division to upper and lower bits of an 8-bit value, $ByteDiv = 16$).

The routine updating the three main data structures then looks as follows:

for $j = 1$ to $\mid R \mid$ do
  if $Change[j] \neq 0$ then do begin
    if $Change[j] < 0$
    then $ChangeValue = -((-Change[j])/ByteDiv$
      $+ (-Change[j]) \setminus ByteDiv)$
    else $ChangeValue = Change[j]/ByteDiv + Change[j] \setminus ByteDiv$;
    $Available[j] = Available[j] - ChangeValue$;
    $Allocated[i][j] = Allocated[i][j] + ChangeValue$;
    if $Change[j] \leq 0$
    then $ChangeValue = -(-Change[j]) \setminus ByteDiv$
    else $ChangeValue = Change[j] \setminus ByteDiv$;
    $RemainingNeeded[i][j] = RemainingNeeded[i][j] - ChangeValue$
  end;

The vectors of relative changes are used not only for (de)allocation of resources, but also for flexible routing. The $RemainingNeeded$ vectors may be different for individual variants of a process description, while every process instance must have one of them in its initial state. That is why we propose two steps to carry out:

1. To order all available variants of the process description and to set the first of them as *primary* – its $RemainingNeeded$ vector will be initial for every process.
2. To define a differential vector between an old and a new variant for every point in the net, where a switch of the two variants is realized.

In the illustrative example, the switch from the primary to the secondary variant is necessary on realizing the transition $T3$ providing the variant with the transitions $T2$, $T5$ and $T8$ is primary and variant with $T3$ and $T6$ is secondary.

In summary, our modifications to the banker's algorithm consist of renaming the $Needed$ data structure to $RemainingNeeded$ (otherwise the original algorithm in the section 2.2 is the same), specifying details of the routine updating the BA data structures and defining related change vectors for every stage of

processes with a (de)allocation in the underlying system. The changes were motivated by the fact that the original algorithm has been constructed for a simpler class of RAS than our application RAS.

3 Banker's Algorithm in CPN

3.1 Construction in CPN ML

Apart from the BA main logic (finding, if the new state is ordered), the implementation needs data structures and routines for recording and managing information about a current state of running processes in the system. Both parts are implemented in user-defined functions of the language CPN ML. In the following sections, we describe colour sets, variables, constant values and functions used. They are fully cited in the appendix.

Data Structures The principal data structure is defined by the colour set $cBankerAlgData$, which is a product of three components corresponding with the above mentioned matrices $Allocated$, $RemainingNeeded$ and the vector $Available$. The first two components are of the colour set $cAllProcessesWRes$, which is a list of items as products $cProcessID * cResNumbersList$. Each product represents a process ($cProcessID$) and a list of numbers of resources ($cResNumbersList$), where each list item corresponds to one resource type. The third component of the $cBankerAlgData$ product is of the latter colour set. An example of use of the $cBankerAlgData$ structure is seen in the constant value $vInitBAData$.

Content of the value $vInitBAData$ corresponds to the above discussed illustrative RAS in its initial state. The model has two process instances, which in the beginning contain no resources, hence the $Allocated$ part of the $vInitBAData$ contains the following list of values: $[(1, [0, 0, 0]), (2, [0, 0, 0])]$. Needs of the process instances (initial value of the $RemainingNeeded$ matrix) are in the second row of the $vInitBAData$ value definition and express that both process instances have the same needs expressed by another constant value:
$[(1, vInitMaxNeedPrimary), (2, vInitMaxNeedPrimary)]$. The value $vInitMaxNeedPrimary$ corresponds to maximal needed numbers of resources for one execution of the primary description of the process type, and that is 2 units of the resource type R1, 3 units of R2 and 1 unit of R3 (it can be verified in the model at fig. 1).

$vMaxNeedPrimarySecondaryDiff$ is the difference vector between the primary and the secondary variants of the example's process type description. This means, that the secondary variant has the maximal needs $[1, 3, 2]$. $vByteDiv$ is the factor for the division of bits in a byte to two halves.

The final group of constant values defines the change vectors (implemented as CPN ML lists) that modify the BA data structure by firing of individual transitions $T1$ to $T9$. Values in vectors correspond to the explanation in the section 2.3 and are connected to the original model on the fig. 1. For instance,

the list [∼16,1,∼16] of the $vChangeT6$ constant value corresponds to the change on the $T6$ transition in the model: an allocation of one unit of the resource $R2$ and a release of the resources $R1$ and $R3$ (one unit of each), while both units will not be requested again in the process type description.

Functions Hierarchy Relations among functions in the CPN implementation are depicted at the figure 2. Arcs represent relations of calling – from a superior function to a subordinate function in the direction of their arc. The functions are divided to three groups. The *Main Algorithm* group corresponds to the slightly modified pseudo-code from the section 2.2. The *Data Structure Manipulation* functions implement the manipulation with data structures introduced in the section 2.3. Finally, there are *General* functions that are not directly attributed to the banker's algorithm. They work with lists of integers – they use a recursion to look through the lists and produce their results.

General Functions The $ModifyList$ function modifies the list pA with the list pB returning a new list in which for every item: $pA{+}pOper{*}pB$ (relevant items in the given lists). It is assumed that both initial lists contain integer values and the parameter $pOper$ determines, whether the operation is an addition ($pOper = 1$) or a subtraction ($pOper = \sim 1$). The $IsIn$ function checks, if all items of the list pA are less than or equal to equivalent items of the list pB (i.e. pA "is in" pB). The function is widely used to compare vectors of resources required and available, and to check, if the request can be covered.

The $ULBits$ and $LowerBits$ functions look at given numbers in the $pList$ list parameter as two-part numbers: upper and lower bits, where the edge is defined by the $vByteDiv$ value (bits manipulation is discussed in the section 2.3). The former function sums up numbers encoded in the upper and the lower bits of values in the given $pList$, e.g. the given list [∼32,18,∼7] is changed to [∼2,3,∼7]. The latter function retrieves only numbers encoded in lower bits of values in the given $pList$, in the example it returns [0,2,∼7].

Data Structure Manipulation Functions Most of the CPN ML functions for the banker's algorithm work with lists, thus use a recursion to traverse them.

A set of functions uses an abbreviation PRL that stands for a list of process instances with a list of resources adjacent to it, shorter *process-resources-list*. The functions are used to manipulate with data in the *Allocated* and the $RemainingNeeded$ lists. The $LocateListInPRL$ function locates the PRL of the process of $pKey$ in the $pPRL$ list and returns its list of resource numbers. The $ModifyPRL_List$ function modifies the given PRL-type list: it selects the item with the $pKey$ and modifies its resource number values according to the $pOper$ operation (addition or subtraction) and returns the updated PRL list. The $RemoveItemFromPRL_List$ function removes the item with $pKey$ from $pPRL_List$ and returns the updated PRL list.

The simple function $ChangeMaxNeed$ only updates BA data according to the $pChange$ item. It is used for switching from an old to a new variant of

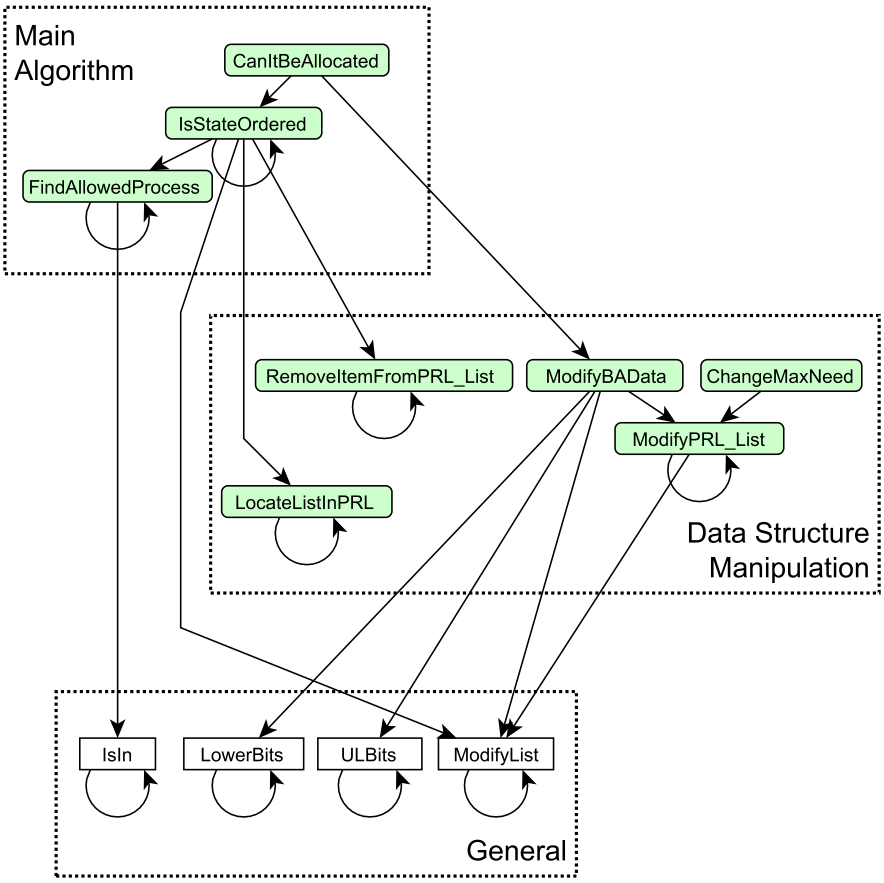

Fig. 2. Diagram of functions in the CPN ML implementation.

process description by flexible routing. The *pChange* argument contains the ID of the process instance and the difference vector between the switched variants. It affects only the middle part of the BA data structure related to the *RemainingNeeded* matrix and only its item related to the given process instance ID.

The $ModifyBAData$ function modifies data for the banker's algorithm given in the *pBAData* parameter by data from the *pChange* parameter, which identifies the respective process instance and contains the change vector with information encoded in its lower and upper bits as explained above. All resources stated in the change vector (i.e. upper and lower bits) are added to the allocated resources of the given process and subtracted from the list of all available resources in the system. However, only the resources encoded in lower bits affect the remaining needed resources of the process (see the 2^{nd} statement in the function).

Main Algorithm Functions The $FindAllowedProcess$ function looks for an item in the list of running process instances (*pRemainNeed*), remaining needed resources of which can be covered by the list of available resources (*Avail*), i.e. a process that can be finished with current available processes. If no process is found, it returns ∼1, otherwise the ID of the process found.

The $IsStateOrdered$ function is the principal function of the banker's Algorithm. It tries to find an ordered sequence of process instances that can be finished in the given conditions. If it is successful, it returns the ordered process sequence. If not, it returns a list with 1 item: [∼1]. [∼2] serves only for recognizing the bottom of the recursion – when all processes were chosen to the order.

Finally, the top function $CanItBeAllocated$ in the hierarchy answers the question, if the process instance with its resource request can be allocated the requested resources. It checks, if the state after the allocation will be ordered – then it returns true, otherwise false.

3.2 Adding Banker's Algorithm to CPN Model

In this phase, there are two tasks to fulfill:

- To construct the required data structure in the CPN and to connect it to the underlying CPN model of RAS,
- To interlink all points of allocation in the model with calls of the BA.

The BA data structure is represented by one token of the *cBankerAlgData* colour set in a dedicated place called $Banker'sAlgData$ (see fig. 3). Its initial value is equal to the constant value $vInitBAData$ (see section 3.1). The connection of the new place to the underlying model is made via pairs of arcs with all transitions, at which contents of the BA data structure are to be changed, i.e. all transitions, where at least one resource allocation or release is modelled. One of the arcs in a pair leads from the place $Banker'sAlgData$ to the transition and

brings the BA data structure token through the variable $BAData$ in the arc inscription to make it available for an execution of the banker's algorithm and for an update of data in case the transition fires. The other arc leads in the opposite direction and contains a call to the function $ModifyBAData$ in order to update the data structure after the transition has been fired. As arguments, the function needs the process ID, the respective change vector in resource allocation and the BA data structure itself, for example:

```
ModifyBAData ((proc, vChangeT5), BAData)
```

where $vChangeT5$ is a constant value containing the change vector for firing the transition $T5$ ($proc$ and $BAData$ are variables bringing the other needed arguments in).

The BA is called in guards of all transitions where a resource allocation request is made. In our example, it is at all transitions except $T7$ and $T8$. The top BA function $CanItBeAllocated$ is called with the same arguments as the $ModifyBAData$ function, for example:

```
[CanItBeAllocated((proc, vChangeT5), BAData)]
```

Being in the transition guard, the algorithm has direct effect on whether the transition is enabled or not. It serves as the last condition for enabling the firing after all basic conditions secured by the structure of RAS (a process instance is in the appropriate state, resources to be allocated are available) are fulfilled. If the state to come after firing is safe according to the algorithm, the $CanItBeAllocated$ function will allow the transition firing. In case that not, the transition will not be enabled. It can be, however, enabled in the future – when the state of the RAS model changes and the transitions in the CPN affected by the change will be re-calculated, the result of the function (while not changing the basic conditions for the transition) may become positive and enable the transition.

When a process instance chooses another variant of execution by flexible routing, the guard of the relevant transition contains a modification of the maximal needed resources for the given process instance via the $ChangeMaxNeed$ function. In the illustrative model, it happens on the transition T3 and its guard is the following:

```
[BADataAmended =
ChangeMaxNeed ((proc, vMaxNeedPrimarySecondaryDiff), BAData),
CanItBeAllocated ((proc, vChangeT3), BADataAmended)]
```

The amended data structure in $BADataAmended$ is then used in the inscription of the arc from the transition to the $Banker'sAlgData$ place instead of $BAData$. For the transition T3 in the example, it is:

```
ModifyBAData ((proc, vChangeT3), BADataAmended)
```

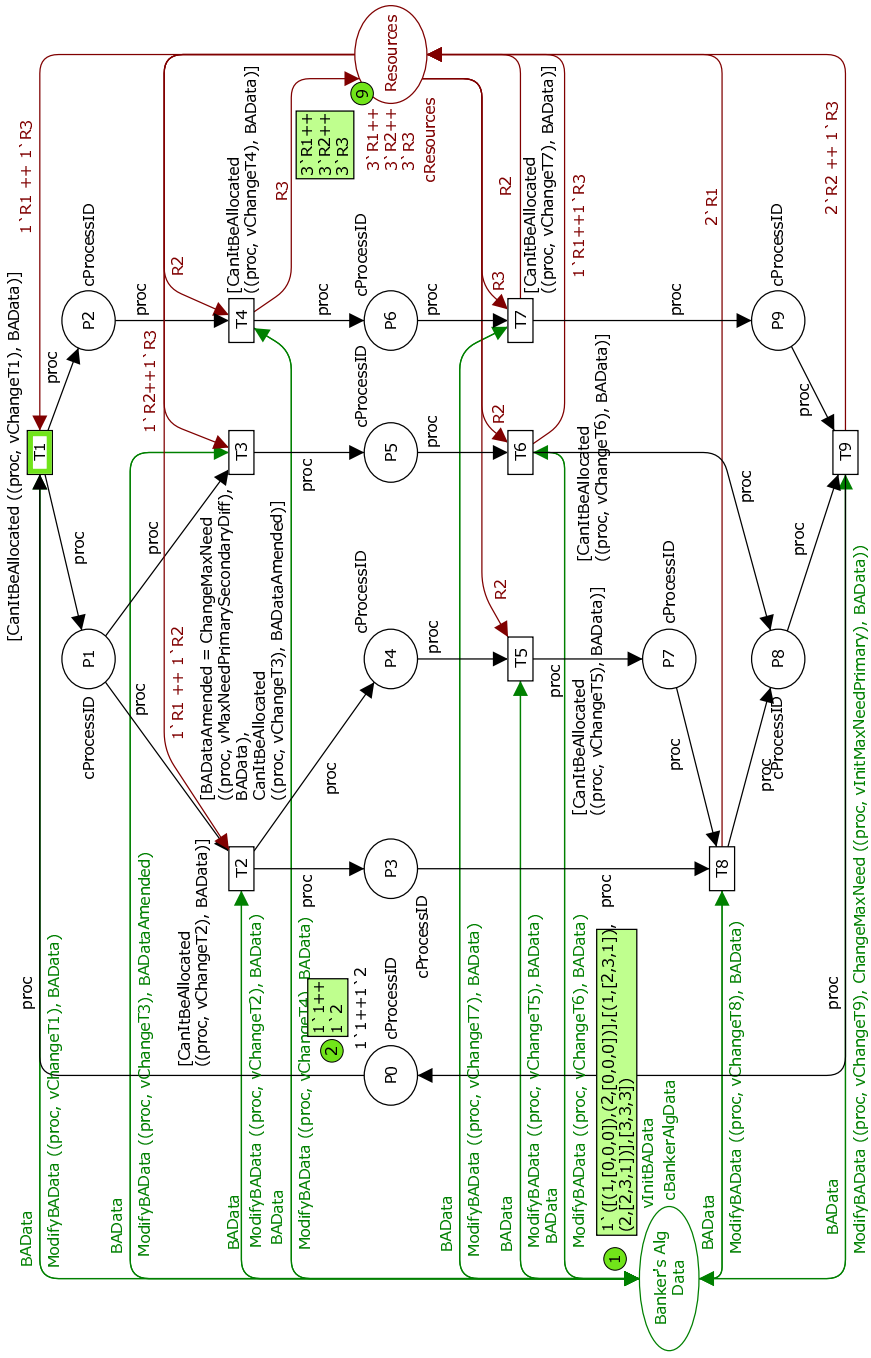

Fig. 3. CPN model of illustration RAS with banker's algorithm.

After its execution, a finished process instance is replaced by a new process instance in the place $P0$. In the BA control subsystem, it is reflected by a use of the $ChangeMaxNeed$ function in the inscription of the arc from the last transition of the process type description to the $Banker'sAlgData$ place – to update maximal needed resources for the new process instance in the BA data structure to the initial $vInitMaxNeedPrimary$ value. Inscription of the arc (in the illustration model from transition T9) looks like this:

```
ModifyBAData ((proc, vChangeT9),
  ChangeMaxNeed ((proc, vInitMaxNeedPrimary), BAData))
```

3.3 Analysis Results

The effect of the banker's algorithm to avoid deadlock states in a modelled system is measured by a number of deadlock states in the occurrence graphs of two CPN models. One is the RAS CPN model without the BA implementation and the other one with it. It is expected that in the first case, the number of deadlock states is not zero, i.e. there are deadlock states in the original RAS to be eliminated. Then, if the deadlock avoidance of the BA is effective, the number of deadlock states in the second model goes down to zero.

In the validation and verification phases, we also further used liveness and home properties of the available analysis tool to check, if the existence of the BA control does not restrict the behaviour of the model in an unappropriate manner, i.e. if all the transitions in the model can occur and whether all reachable markings form a home space. The state space report proved that it was correct.

Furthermore, it is interesting to see, how the algorithm restricts the state space of the original CPN model, since it avoids not only the deadlock states, but also unsafe states leading to deadlock states and also some safe states that are however not accepted by the algorithm due to its suboptimal calculation (see section 2.2).

Tests of the outlined algorithm with two RAS CPN models always showed that it avoids all deadlock states as expected. In the illustrative example, the CPN model of the RAS without the BA contains 12 deadlock states, while the CPN model with the BA contains no deadlock states. As for restriction of their state space, the occurrence graph of the original model has 152 nodes and 378 arcs, while the BA-controlled model has only 113 nodes and 260 arcs.

4 Conclusion

In this paper, we focused on an implementation of the banker's algorithm (BA) for deadlock avoidance in a resource allocation system (RAS) with non-sequential processes with flexible routing and a use of resources of multiple types at once, all modelled as a coloured Petri net in the environment of the CPN Tools. The original version of the BA has been slightly modified in the direction of its application for the outlined system, requiring the introduction of vectors of relative

changes necessary for processes combining all three outlined properties: concurrent processing, flexible routing and use of multiple resources of multiple types at once.

The algorithm has been verified to be effective on such a system. This result we consider as the major contribution of this work, since the application of the banker-like algorithms to such a class of RAS has not been apparently discussed in the literature so far.

Selection of the CPN Tools environment was mainly motivated by the abilities of the fast construction of the underlying RAS model in the CPN and of the available analysis of occurrence graphs on the presence of deadlock states. Both proved to be beneficial. Especially the analysis ability is a very strong tool – it enables to further study behaviour of the BA and its versions on a toy example via detailed analysis of its state space.

However, the implementation of the basic version of the BA in 12 CPN ML functions looks rather complicated compared to sequential programming. Also maintaining information about the global state of a RAS modelled in CPN is rather complicated – it is concentrated to one place, which is connected to many transitions, where allocation requests occur. This makes the CPN more difficult to read, especially for large RAS models. Solution to this is using hierarchy for the CPN: dividing process subnet(s) in RAS to CPN subpages, each of them containing only a few transitions. In the illustrative example in the paper, the process subnet could be split e.g. to three subpages. However, in order to show the whole approach, the hierarchy was not used for the model in this paper.

Further issues are connected to the use of the outlined results in the field, where the motivation comes from – for deadlock avoidance in computer simulation of complex transportation systems.

First, the complex models produce very large state spaces that cannot be fully verified in a reasonable time like the presented simple example. We believe that once the BA has been verified on a small scale example preserving the outlined properties of the complex transportation system, it will be effective also on complex models with tens of process types and process instances and hundreds of resource types as well as resource units. The research will be rather focused on making the run of the BA more effective for the large system.

Secondly, the BA needs to be adjusted to a more complicated way of allocation and release of resources that is currently present in the original simulation models. The first version of the BA with this property has been already made [1], however, it requires further research and testing.

Thirdly, apart from the BA's basic version, we have implemented two more versions of the BA according to [5] – for partially-ordered and V1-ordered states in the CPN Tools [1]. Their explanation requires however more space and is thus outside of scope of this paper. The open question here is, if implementation of other versions, for Vn-ordered states (according to the mentioned paper) is effective and beneficial in our application field.

All the briefly outlined issues frame our future work in this context.

Acknowledgements. This paper has been supported by the grant of the Scientific Grant Agency VEGA 1/4057/07 in the Slovak Republic and the research project MŠM 0021627505 – Theory of transport systems in the Czech Republic.

References

1. Žarnay, M.: Systém na podporu rozhodovania pre riadenie dopravných procesov [Decision-support system for transportation systems control]. PhD thesis, Faculty of Management Science and Informatics, University of Žilina (January 2007) in Slovak.
2. Žarnay, M.: Solving deadlock states in model of railway station operation using coloured petri nets. In Tarnai, G., Schnieder, E., eds.: Formal Methods for Automation and Safety in Railway and Automotive Systems. (2008) to appear
3. Peterson, J.L.: Operating System Concepts. Addison-Wesley (1981)
4. Dijkstra, E.W.: Co-operating sequential processes. In Genuys, F., ed.: Programming Languages, New York, Academic Press (1968) 43112 Reprinted from: Technical Report EWD-123, Technological University, Eindhoven, the Netherlands, 1965.
5. Lawley, M.A., Reveliotis, S.A., Ferreira, P.M.: The application and evaluation of banker's algorithm for deadlock-free buffer space allocation in flexible manufacturing systems. International Journal of Flexible Manufacturing Systems **10** (1998) 73–100
6. Tricas, F., Colom, J.M., Ezpeleta, J.: Some improvements to the banker's algorithm based on the process structure. Proceedings of IEEE International Conference on Robotics and Automation **3** (2000) 2853–2858 San Francisco, CA, USA.
7. Tricas, F.: Deadlock Analysis, Prevention and Avoidance in Sequential Resource Allocation Systems. PhD thesis, Departamento de Informática e Ingeniería de Sistemas, Universidad de Zaragoza (May 2003)
8. Ezpeleta, J., Tricas, F., García-Vallés, F., Colom, J.M.: A banker's solution for deadlock avoidance in fms with flexible routing and multiresource states. IEEE Transactions on Robotics and Automation **18**(4) (August 2002) 621–625
9. Lang, S.D.: An extended banker's algorithm for deadlock avoidance. IEEE Transactions on software engineering **25**(3) (1999) 428–432
10. Reveliotis, S.A.: Conflict resolution in agv systems. IIE Transactions **32**(7) (2000) 647–659
11. Ezpeleta, J., Valk, R.: A polynomial deadlock avoidance method for a class of nonsequential resource allocation systems. IEEE Transactions on Systems, Man and Cybernetics, Part A **36**(6) (2006) 1234–1243

Appendix

A complete listing of colour sets, variables, constant values and functions used for the implementation of the BA in the CPN ML is available here. It complements the description of the BA implementation in the CPN ML in the section 3.1 Concrete values are related to the presented illustrative CPN model.

Data Structures

```
colset cProcessID = INT;
colset cResources = with R1 | R2 | R3;
colset cResNumbersList = list INT;
colset cProcessList = list cProcessID;
colset cResources4Process = product cProcessID * cResNumbersList;
colset cAllProcessesWRes = list cResources4Process;
colset cBankerAlgData = product cAllProcessesWRes *
  cAllProcessesWRes * cResNumbersList;

var proc: cProcessID;
var BAData, BADataAmended: cBankerAlgData;

val vInitMaxNeedPrimary = [2,3,1]
val vMaxNeedPrimarySecondaryDiff = [~1,0,1];
val vByteDiv = 16;
val vInitBAData =
([(1,[0,0,0]), (2,[0,0,0])],
[(1,vInitMaxNeedPrimary), (2,vInitMaxNeedPrimary)],
[3,3,3]);

val vChangeT1 = [1,0,1];
val vChangeT2 = [1,1,0];
val vChangeT3 = [0,1,1];
val vChangeT4 = [0,1,~1];
val vChangeT5 = [0,1,0];
val vChangeT6 = [~16,1,~16];
val vChangeT7 = [0,~16,1];
val vChangeT8 = [~32,0,0];
val vChangeT9 = [0,~32,~16];
```

General Functions

```
fun ModifyList (pA, _, []) = pA
| ModifyList ([], _, pB) = pB
| ModifyList (pA, pOper, pB) =
  (hd pA + pOper * hd pB) :: ModifyList (tl pA, pOper, tl pB);

fun IsIn ([],[]) = true
| IsIn ([], _) = false
| IsIn (_, []) = false
| IsIn (pA, pB) =
  if hd pA <= hd pB andalso IsIn (tl pA, tl pB)
  then true else false;
```

```
fun ULBits ([]) = []
| ULBits (pList) =
  if hd pList < 0 then
      ~(~(hd pList) div vByteDiv + ~(hd pList) mod vByteDiv) ::
      ULBits (tl pList)
    else ((hd pList) div vByteDiv + (hd pList) mod vByteDiv) ::
      ULBits (tl pList);

fun LowerBits ([]) = []
| LowerBits (pList) =
  if hd pList < 0 then
      ~(~(hd pList) mod vByteDiv) :: LowerBits (tl pList)
    else ((hd pList) mod vByteDiv) :: LowerBits (tl pList);
```

Data Structure Manipulation Functions

```
fun LocateListInPRL ([], _) = []
| LocateListInPRL (pPRL: cAllProcessesWRes, pKey) =
  if #1 (hd pPRL) = pKey then #2 (hd pPRL)
  else LocateListInPRL (tl pPRL, pKey);

fun ModifyPRL_List ([], _, _) = []
| ModifyPRL_List (pPRLList: cAllProcessesWRes,
pOper, pPRLItem: cResources4Process) =
  if #1 (hd pPRLList) <> #1 (pPRLItem)
  then (hd pPRLList) ::
        ModifyPRL_List (tl pPRLList, pOper, pPRLItem)
  else (#1 (hd pPRLList), ModifyList (#2 (hd pPRLList),
    pOper, #2 pPRLItem)) :: (tl pPRLList);

fun RemoveItemFromPRL_List ([], _) = []
| RemoveItemFromPRL_List (pPRL_List: cAllProcessesWRes, pKey) =
  if #1 (hd pPRL_List) = pKey then tl pPRL_List
  else hd pPRL_List ::
    RemoveItemFromPRL_List (tl pPRL_List, pKey);

fun ChangeMaxNeed (pChange, pBAData: cBankerAlgData) =
(#1 pBAData,
 ModifyPRL_List (#2 pBAData, 1, pChange),
 #3 pBAData);

fun ModifyBAData (pChange: cResources4Process,
pBAData: cBankerAlgData): cBankerAlgData =
(
  ModifyPRL_List (#1 pBAData, 1,
    (#1 pChange, ULBits(#2 pChange))),
```

```
  ModifyPRL_List (#2 pBAData, ~1,
    (#1 pChange, LowerBits(#2 pChange))),
  ModifyList (#3 pBAData, ~1, ULBits(#2 pChange))
);
```

Main Algorithm Functions

```
fun FindAllowedProcess ([], _): cProcessID = ~1
| FindAllowedProcess (pRemainNeed: cAllProcessesWRes,
pAvail: cResNumbersList): cProcessID =
  if IsIn (#2 (hd pRemainNeed), pAvail) then #1 (hd pRemainNeed)
  else FindAllowedProcess (tl pRemainNeed, pAvail);

fun IsStateOrdered (_, [], _) = [~2] (* recursion at the bottom *)
| IsStateOrdered (Alloc, RemainNeed: cAllProcessesWRes,
  Avail: cResNumbersList): cProcessList =
  let
    (* looking for a process that can be chosen to the order *)
    val proc = FindAllowedProcess (RemainNeed, Avail)
  in
    (* if unsuccessful, state is not ordered and return [~1] *)
    if proc = ~1 then [~1]
    else
      (* if process found, continue to the next round *)
      let val result = IsStateOrdered
          (RemoveItemFromPRL_List (Alloc, proc),
          RemoveItemFromPRL_List (RemainNeed, proc),
          ModifyList (LocateListInPRL (Alloc, proc), 1, Avail))
      in
          case result of (* of previous round of recursion *)
            [~1] => [~1] (* was unsuccessful, pass it further *)
          | [~2] => [proc] (* returned from end of recursion,
               start to build up the ordered process sequence *)
          | _ => proc :: result  (* was successful:
               building up the process sequence *)
        end
  end;

fun CanItBeAllocated (pRequest: cResources4Process,
pBAData: cBankerAlgData): BOOL =
  if IsStateOrdered (ModifyBAData (pRequest, pBAData)) = [~1]
  then false
  else true;
```