The Resource Allocation Problem in Software Applications: A Petri Net Perspective*

Juan-Pablo López-Grao[1] and José-Manuel Colom[2]

[1] Dpt. of Computer Science and Systems Engineering (DIIS)
[2] Aragonese Engineering Research Institute (I3A)
University of Zaragoza, Spain
Email: {jpablo,jm}@unizar.es

Abstract. Resource Allocation Systems (RAS) have been intensively studied in the last years in the domain of Flexible Manufacturing Systems (FMS). The success of this research line has been based on the identification of particular subclasses of Petri Nets that correspond to a RAS abstraction of this kind of systems. In this paper we take a parallel road to that travelled through for FMS, but for the case of software applications. The considered applications present concurrency and deadlocks can happen due to the allocation of shared resources. We will evince that the existing subclasses of Petri Nets used to study this kind of deadlock problems are insufficient, even for very simple software systems. From this starting point we propose a new subclass of Petri Nets that generalizes the previously known RAS subclasses and we present a taxonomy of anomalies that can be found in the context of software systems.

1 Introduction

Among the most recurrent patterns in a wide disparity of engineering disciplines, the competition for shared resources between concurrent processes takes a prominent position. The reader might think of examples in the context of distributed systems, operations research, manufacturing plants, etc. The perspective of discrete event systems theory proves appropriate and powerful as a framework in which provide solutions to the so-called resource allocation problem [1]. Systems of this kind are often called Resource Allocation Systems (RAS) [2, 3].

RAS are usually conceptualized around two distinct entities, processes and resources, thanks to a prior abstraction process which is inherent in the discipline. The resource allocation problem refers to satisfying successfully the requests for resources made by the processes, ensuring that no process ever falls in a deadlock. A set of processes is deadlocked when they indefinitely wait for resources that are already held by other processes of the same set [4].

RAS can be categorized both on the type of processes (sequential, non-sequential) and resources (serially reusable, consumable) [5]. Hereafter, we will

* This work has been partially supported by the European Community's Seventh Framework Programme under Project DISC (Grant Agreement n. INFSO-ICT-224498) and the project CICYT-FEDER DPI2006-15390.

Recent Advances in Petri Nets and Concurrency, S. Donatelli, J. Kleijn, R.J. Machado, J.M. Fernandes (eds.), CEUR Workshop Proceedings, ISSN 1613-0073, Jan/2012, pp. 219–233.

focus on Sequential RAS with serially reusable resources. This means that a process can increase or decrease the quantity of free resources during its execution. However, the process will contervail that operation before terminating, i.e. resources are used in a conservative way.

Although other models of concurrency have also been considered [6], Petri nets [7] have arguably taken a leading role among the family of formal models used for dealing with the resource allocation problem [8, 9]. One of the strengths of this approach is the smooth mapping between the main entities of RAS and the basic elements of Petri net models. A resource type can be modelled using a place: the number of instances of it being modelled with tokens. Meanwhile, sequential processes are modelled with tokens progressing through state machines. Arcs from resource places to transitions (from transitions to resource places) represent the acquisition (return) of some resources by a process. Petri nets thus provide a natural formal framework for the analysis of RAS, besides benefiting from the goods of compositionality.

This fact is well notorious in the domain of Flexible Manufacturing Systems (FMS), where Petri net models for RAS have widely succeeded since the seminal work of Ezpeleta et al. was introduced [8]. This is mostly due to a careful selection of the subclass of Petri nets used to model these FMS, based upon two solid pillars. First, the definition of a rich syntax from a physical point of view, which enables the natural expression of a wide disparity of plant configurations. And second, the contribution of sound scientific results which let us characterize deadlocks from the model structure, as well as provide a well-defined methodology to automatically correct them in the real system.

Nowadays, there exists a plethora of Petri net models for modelling RAS in the context of FMS, which often overcome some of the syntactical limitations of the S^3PR class [8]. S^4PR net models [10, 11] generalize the earlier, while allowing multiple simultaneous allocations of resources per process. S^*PR nets [12] extend the expressive power of the processes to that of state machines: hence internal cycles in their control flow is allowed. However, deadlocks in S^*PR net models are not fully comprehended from a structural perspective. Other classes such as NS-RAP [9], ERCN-merged nets [13] or PNR nets [14] extend the capabilities of S^3PR/S^4PR models beyond Sequential RAS by way of lot splitting or merging operations.

Most analysis and control techniques in the literature are based on the computation of a structural element which univocally characterizes deadlocks in many RAS models: the so-called *bad siphon*. A bad siphon is a siphon which is not the support of a p-semiflow. If bad siphons become (sufficiently) emptied, their output transitions die since the resource places of the siphon cannot regain tokens anymore, thus revealing the *deadly embrace*. Control techniques thus rely on the insertion of monitor places [15], i.e. controllers in the real system, which limit the leakage of tokens from the bad siphons.

Although there exist obvious resemblances between the resource allocation problem in FMS and that of parallel or concurrent software, previous attempts to bring these well-known RAS techniques into the field of software engineering

have been, to the best of our knowledge, either too limiting or unsuccessful. Gadara nets [16] constitute the most recent attempt, yet they fall in the over-restrictive side in the way the resources can be used, as a result of inheriting the design philosophy applied for FMS. In this work, we will analyze why the net classes and results introduced in the context of FMS fail when brought to the field of concurrent programming.

Section 2 presents a motivating example and discusses the elements that a RAS net model should desirably feature in order to successfully explore the resource allocation problem within the software enginering discipline. Taking into account those considerations, section 3 introduces a new Petri net class, called PC^2R. Section 4 relates the new class to those defined in previous works and establishes useful net transformations which forewarn us about new behavioural phenomena. Section 5 introduces some of these anomalies which highlight the fact that previous theoretical results in the context of FMS are insufficient in the new framework. Finally, section 6 summarizes the results of the paper.

2 The RAS view of a software application

Example 1 presents a humorous variation of Dijkstra's classic problem of the dining philosophers. We will adopt and adapt the beautiful writing by Hoare at [17] for its enunciation.

Example 1. The pragmatic dining philosophers. "Five philosophers spend their lives thinking and eating. The philosophers share a common dining room where there is a circular table surrounded by five chairs, each belonging to one philosopher. A microwave oven is also available. In the center of the table there is a large bowl of spaghetti which is frequently refilled (so it cannot be emptied), and the table is laid with five forks. On feeling hungry, a philosopher enters the dining room, sits in his own chair, and picks up the fork on the left of his place. Then he touches the bowl to feel its temperature. If he feels the spaghetti got too cold, he will leave his fork and take the bowl to the microwave. Once it is warm enough, he will come back to the table, sit on his chair and leave the bowl on the table after recovering his left fork (please bear in mind that the philosopher is *really* hungry by now). Unfortunately, the spaghetti is so tangled that he needs to pick up and use the fork on his right as well. If he can do it before the bowl gets cold again, he will serve himself and start eating. When he has finished, he puts down both forks and leaves the room."

According to the classic RAS nomenclature, each philosopher is a sequential process, and the five forks plus the bowl are serially reusable resources which are shared among the five processes. From a software perspective, each philosopher can be a process or a thread which will be executed concurrently.

Algorithm 1 introduces the code for each philosopher. Notationally, we modelled the acquisition / release of resources by way of the `wait()` / `signal()` operations, respectively. Both of them have been generalized for the acquisition of multiple resources (separated by commas when invoking the function). Finally,

the `trywait()` operation is a non-blocking wait operation. If every resource is available at the time `trywait()` is invoked, then it will acquire them and return `TRUE`. Otherwise, `trywait()` will return `FALSE` without acquiring any resource. For the sake of simplicity, it is assumed that the conditions with two or more literals are evaluated atomically.

Algorithm 1 - Code for Philosopher i (where $i \in \{1, 2, 3, 4, 5\}$)

```
var
     fork: array [1..5] of semaphores; // shared resources
     bowl: semaphore;                  // shared resource
begin
   do while (1)
      THINK;
      Enter the room;
(T1)  wait(fork[i]);
      do while (not(trywait(bowl, fork[i+1 mod 5]))
                or the spaghetti is cold)
(T2)     if (trywait(bowl)
                and the spaghetti is cold) then
(T3)        signal(fork[i]);
            Go to the microwave;
            Heat up spaghetti;
            Go back to table;
(T4)        wait(fork[i]);
(T5)        signal(bowl);
         end if;
(T6)  loop;
      Serve spaghetti;
(T7)  signal(bowl);
      EAT;
(T8)  signal(fork[i], fork[i+1 mod 5]);
      Leave the room;
   loop;
```

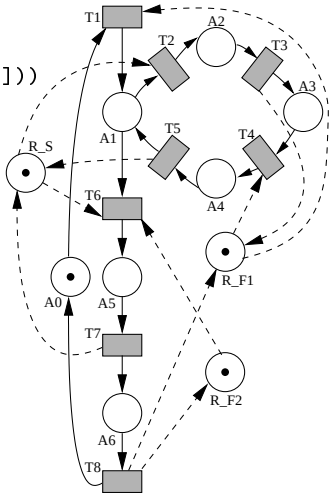

Fig. 1. Philosopher 1.

Figure 1 depicts the net for algorithm 1, with $i = 1$, after abstracting the relevant information from a RAS perspective. Figure 2 renders the composition of the five philosopher nets via fusion of the common shared resources. Note that if we remove the dashed arcs from figure 2, then we can see five disjoint strongly connected state machines plus six isolated places.

The five state machines represent the control flow for each philosopher. Every state machine is composed of seven states (each state being represented by a place). Tokens in a state machine represent concurrent processes/threads which share the same control flow. In this case, the unique token in each machine is located at the so-called *idle place*. This means that, at the initial state, every philosopher is thinking (outside the room). In general, the idle place can be seen

as a mechanism which enforces a structural bound: the number of concurrent *active threads* (i.e. non-idle) is limited. Here, at most one philosopher of type i can be inside the room, for each $i \in \{1, 2, 3, 4, 5\}$.

The six isolated places are called *resource places*. A resource place represents a certain resource type, and the number of tokens in it represents the quantity of free instances of that resource type. In this case, every resource place is monomarked. Thus, at the initial state there is one fork of type i, for every $i \in \{1, 2, 3, 4, 5\}$, plus one bowl of spaghetti (modelled by way of the resource place at the centre of the figure).

Finally, the dashed arcs represent the acquisition or release of resources by the active threads when they change their execution state. Every time a transition is fired, the total amount of resources available is altered. Please note, however, that moving one isolated token of a state machine (by firing its transitions) until the token reaches back the idle state, leaves the resource places marking unaltered. Thus, the resource usage is conservative.

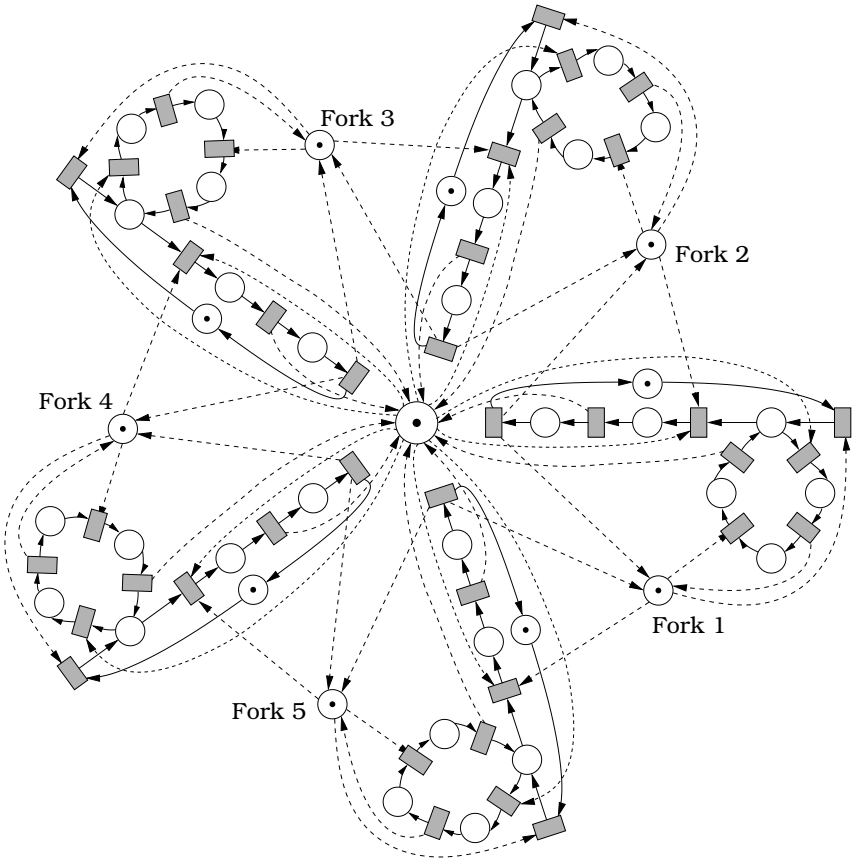

Fig. 2. The dining philosophers are thinking. Arcs from/to P_R are dashed for clarity.

At this point, we will discuss some capabilities that (in our humble opinion) a RAS model should have so as to support the modelling of concurrent programs.

Although acyclic sequential state machines are rather versatile as models for sequential processes in the context of FMS (as the success of the S^3PR and S^4PR classes prove), this is clearly too constraining even for very simple software systems. Considering Böhm and Jacopini's theorem [18], however, we can assume that every non-structured sequential program can be refactored into a structured one using `while-do` loops. Meanwhile, calls to procedures and functions can be substituted by way of inlining techniques. Let us also remind that `fork`/`join` operations can also be unfolded into isolated concurrent sequential processes, as evidenced in [9]. As a result, we can restrict process models to state machines in which decisions and iterations (in the form of `while-do` loops) are supported, but not necessarily every kind of internal cycle.

Another significant difference between FMS and software systems from a RAS perspective is that resources in the latter are not necessarily physical (e.g., a file) but can also be logical (e.g., a semaphore). This has strong implications in the degree of freedom allowed for allocating those resources: we will return to this issue a little later.

In this domain, a resource is an object that is shared among concurrent processes/threads and must be used in mutual exclusion. Since the number of resources is limited, the processes will compete for the resource and will use it in a non-preemptive way. This particular allocation scheme can be imposed by the resources' own access primitives, which may be blocking. Otherwise, the resource can be protected by a binary semaphore/mutex/lock (if there is only one instance of that resource type) or by a counting semaphore (multiple instances). Note that this kind of resources can be of assorted nature (e.g., shared memory locations, storage space, database table rows) but the required synchronization scheme is inherently similar.

On the other side, it is well-known that semaphores used in that aim can be also seen as non-preemptive resources which are used in a conservative way. For instance, a counting semaphore that limits the number of connections to a database can be interpreted in that way from a RAS point of view. Here processes will wait for the semaphore when attempting to establish a database connection, and will release it when they decide to close the aforementioned connection.

However, semaphores also perform a relevant role as an interprocess signaling facility, which can also be a source of deadlocks. In this work, our goal is the study of the resource allocation problem, so this functionality is out of scope. We propose fixing deadlock problems due to resource allocation issues firstly, and later apply other techniques for amending those due to message passing.

Due to their versatility, semaphore primitives are interesting for studying how resources can be allocated by a process/thread. For instance, XSI semaphores (also known as System V semaphores) have a multiple wait primitive (`semop` with `sem_op<0`). An example of multiple resource allocation appears in algorithm 1. Besides, an XSI semaphore can be decremented atomically in more than one. Both POSIX semaphores (through `sem_trywait`) and XSI semaphores (through

`semop` with `sem_op<0` and `sem_flag=IPC_NOWAIT`) have a non-blocking wait primitive. Again, algorithm 1 could serve as an example. Finally, XSI semaphores also feature inhibition mechanisms (through `semop` with `sem_op=0`), i.e. processes can wait for a zero value of the semaphore.

As we suggested earlier, the fact that resources in software engineering do not always have a physical counterpart is a very peculiar characteristic with consequences. In this context, processes do not only consume resources but also can *create* them. A process will destroy the newly created resources before its termination. For instance, a process can create a shared memory variable (or a service!) which can be allocated to other processes/threads. Hence the resource allocation scheme is no longer *first-acquire-later-release*, but it can be the other way round too. Nevertheless, all the resources will be used in a conservative way by the processes (either by a create-destroy sequence or by a wait-release sequence). As a side effect, and perhaps counterintuitively, there may not be free resources during the system startup (as they still must be created), yet being the system live.

Summing up, for successfully modelling RAS in the context of software engineering, a Petri net model should have at least the following abstract properties:

1. The control flow of the processes should be represented by state machines with support for decisions (`if-then-else` blocks) and nested internal cycles (`while-do` blocks).
2. There can be several resource types and multiple instances of each one.
3. State machines can have multiple tokens (representing concurrent threads).
4. Processes/threads use resources in a conservative way
5. Acquisition/release arcs can have non-ordinary weights (e.g., a semaphore value can be atomically incremented/decremented in more than one unit)
6. Atomic multiple acquisition/release operations must be allowed
7. Processes can have decisions dependent of the allocation state of resources (due to the non-blocking wait primitives, as in figure 2)
8. Processes can lend resources. As a side effect, there could exist processes that depend on resources which must be created/lent by other processes (hence they cannot finish if executed in isolation)

3 PC^{2}R nets

In this section, we will present a new Petri net class, which fulfills the requirements advanced in section 2: the class of Processes Competing for Conservative Resources (PC^{2}R). This class generalizes other subclasses of the S^{n}PR family while respecting the design philosophy on these. Hence, previous results are still valid in the new framework. However, PC^{2}R nets can deal with more complex scenarios which were not yet addressed from the domain of S^{n}PR nets.

Definition 1 presents a subclass of state machines which is used for modelling the control flow of the processes in isolation. Iterations are allowed, as well as decisions within internal cycles, in such a way that the control flow of structured

programs can be fully supported. Non-structured processes can still be refactored into them as discussed in Section 2.

Definition 1. *An* iterative state machine $\mathcal{N} = \langle \{p_0\} \cup P, T, C \rangle$ *is a strongly connected state machine such that either every cycle contains* p_0 *or* P *can be partitioned into two subsets* P_1, P_2*, with a place* $p \in P_2$ *such that:*

1. *The subnet generated by* $\langle \{p\} \cup P_1, {}^\bullet P_1 \cup P_1{}^\bullet \rangle$ *is a strongly connected state machine in which every cycle contains* p*, and*
2. *The subnet generated by* $\langle \{p_0\} \cup P_2, {}^\bullet P_2 \cup P_2{}^\bullet \rangle$ *is an iterative state machine.*

In figure 1, if we remove the resource places R_F1, R_F2 and R_S then we obtain an iterative state machine, with $P_1 = \{A2, A3, A4\}$, $P_2 = \{A1, A5, A6\}$, $p_0 = A0$ and $p = A1$. The definition of iterative state machines is instrumental for introducing the class of PC^2R nets.

PC^2R nets are modular models. Two PC^2R nets can be composed into a new PC^2R model via fusion of the common shared resources. Please note that a PC^2R net can simply be one process modelled by an iterative state machine along with the set of resources it uses. Hence the whole net model can be seen as a composition of the modules for each process. We will formally define the class in the following:

Definition 2. *Let* $I_\mathcal{N}$ *be a finite set of indices. A* PC^2R *is a connected generalized pure P/T net* $\mathcal{N} = \langle P, T, C \rangle$ *where:*

1. $P = P_0 \cup P_S \cup P_R$ *is a partition such that: (a)* [idle places] $P_0 = \{p_{0_1}, ..., p_{0_{|I_\mathcal{N}|}}\}$*; (b)* [process places] $P_S = P_1 \cup ... \cup P_{|I_\mathcal{N}|}$*, where* $\forall i \in I_\mathcal{N}$*:* $P_i \neq \emptyset$ *and* $\forall i, j \in I_\mathcal{N}$*:* $i \neq j$*,* $P_i \cap P_j = \emptyset$*; (c)* [resource places] $P_R = \{r_1, ..., r_n\}, n > 0$.
2. $T = T_1 \cup ... \cup T_{|I_\mathcal{N}|}$*, where* $\forall i \in I_\mathcal{N}, T_i \neq \emptyset$*, and* $\forall i, j \in I_\mathcal{N}, i \neq j, T_i \cap T_j = \emptyset$.
3. *For all* $i \in I_\mathcal{N}$ *the subnet generated by restricting* $\mathcal{N}$ *to* $\langle \{p_{0_i}\} \cup P_i, T_i \rangle$ *is an iterative state machine.*
4. *For each* $r \in P_R$*, there exists a unique minimal p-semiflow associated to* r*,* $Y_r \in \mathbb{N}^{|P|}$*, fulfilling:* $\{r\} = \|Y_r\| \cap P_R, (P_0 \cup P_S) \cap \|Y_r\| \neq \emptyset$*, and* $Y_r[r] = 1$.
5. $P_S = \bigcup_{r \in P_R} (\|Y_r\| \setminus \{r\})$.

Please note that the support of the Y_r p-semiflows (point 4 of definition 2) may include P_0: this is new with respect to S^4PR nets. Such a resource place r is called a *lender* resource place. If r is a lender, then there exists a process which creates (*lends*) instances of r. In our model, processes can start their execution creating resource instances, but *before* acquiring any other resource. Otherwise, it could happen that the support of a minimal p-semiflow would contain more than one resource place (thus infriging condition 4 of definition 2).

The class supports iterative processes, multiple resource acquisitions, non-blocking wait operations and resource lending. Inhibition mechanisms are not natively supported (although some cases can still be modelled with PC^2R nets).

The next definition generalizes the notion of acceptable initial marking introduced for the S^4PR class. In software systems all processes/threads are initially

inactive and start from the same point (the `begin` operation). Hence, all of the corresponding tokens are in the idle place in the initial marking (the process places being therefore empty). Note that lender resource places may be empty for an acceptable initial marking. Figure 2 shows a P^2CR net with an acceptable initial marking which does not belong to the S^4PR class.

Definition 3. *Let* $\mathcal{N} = \langle P_0 \cup P_S \cup P_R, T, C \rangle$ *be a* PC^2R. *An initial marking* m_0 *is acceptable for* $\mathcal{N}$ *iff* $||m_0|| = P_0 \cup P_R$ *and* $\forall p \in P_S, r \in P_R \ : \ Y_r^T \cdot m_0 \geq Y_r[p]$.

4 Some transformations and related classes

In [19], we introduced a new class of Petri net models for RAS, called SPQR (Systems of Processes Quarreling over Resources). SPQR nets feature an appealing syntactical simplicity and expressive power though they are very challenging from an analytical point of view. They can be roughly described as RAS nets in which the process subnets are acyclic and the processes can lend resources in any possible (conservative) manner. Every PC^2R can be transformed into a Structurally Bounded SPQR net (SB SPQR net).

The transformation rule is based on the idea of converting every while-do block in an acyclic process which is activated by a lender resource place. This lender place gets marked once the thread reaches the while-do block. The token is removed at the exit of the iteration. This transformation must be applied starting by the most intern loops, proceeding in decreasing nesting order. Figure 3 depicts the transformation rule. The rule preserves the language accepted by the net (and thus liveness) since it basically consists in the addition of a implicit place (place $P1$ in the right hand net of figure 3, since R_P1 can be seen as a renaming of $P1$ in the left hand net).

Figure 4 illustrates the transformation of the net of example 1 but restricted to two philosophers into the corresponding SB SPQR.

Thanks to such transformations, the SB SPQR class can express the widest range of systems in the Sequential RAS Petri net family. Figure 5 introduces the inclusion relations between a variety of Petri net classes for Sequential RAS.

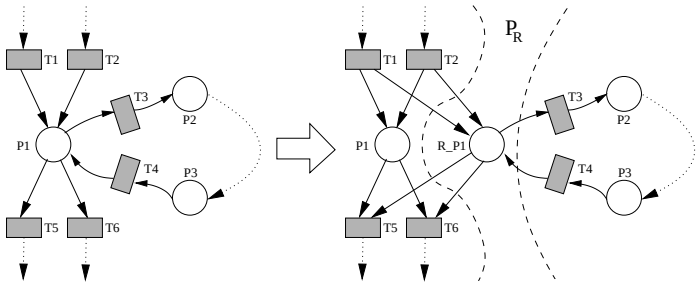

Fig. 3. Transforming PC^2Rs into SB SPQRs: From iterative to acyclic processes

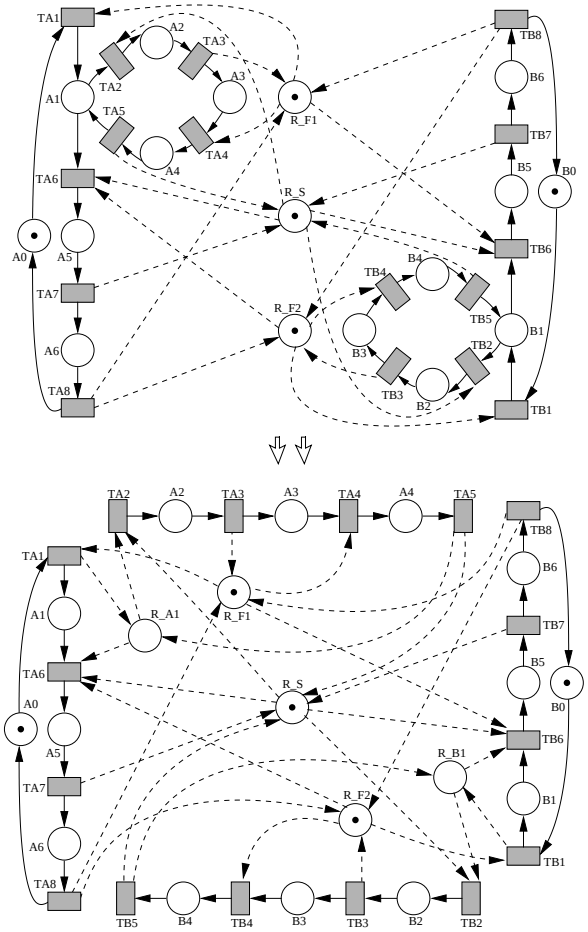

Fig. 4. From PC^{2}R to SB SPQR: Two pragmatic dining philosophers

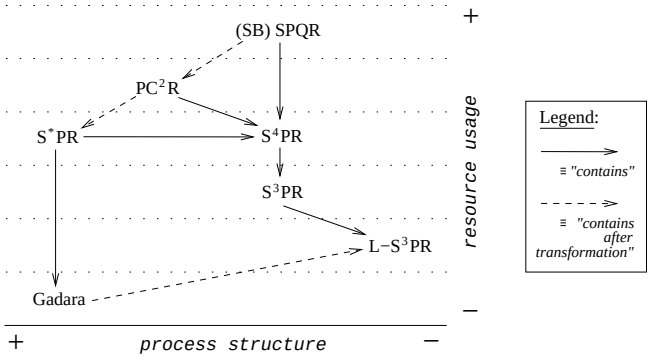

Fig. 5. Inclusion relations between Petri net classes for RAS

5 Some bad properties through examples

The bad news about the discussion in sections 2 and 3 is that siphon-based control techniques for RAS do not work in general for concurrent software, even ignoring (i.e., not using) the *resource lending* feature introduced by PC^{2}R nets.

Let us have a look back at example 1 and its related algorithm 1. It is not difficult to see that, if every philosopher enters the room, sits down and picks up the fork on the left of himself, the philosophers will be trapped in a livelock. Every philosopher can eventually take the bowl of spaghetti and heat it up in the microwave. This pattern can be repeated infinitely, but it is completely useless, since no philosopher will ever be able to have dinner.

This behaviour is obviously reflected in the corresponding net representation at figure 2. Let us construct a firing sequence σ containing only the first transition of each state machine (i.e., the output transition of its idle place). The firing order of these transitions is irrelevant. Now let us fire such a sequence, and the net falls in a livelock. The internal cycles are still firable in isolation, but no idle place can ever be marked again. Unfortunately, the net has several bad siphons, but none of them is empty or insufficiently marked in the livelock. In other words, for every reachable marking in the livelock, there exist output transitions of the siphons which are firable. As a result, the siphon-based non-liveness characterization for earlier net classes (such as S^{4}PR [10]) is not sufficient in the new framework.

A similar pattern can be observed in the upper net of figure 4. There exist three bad siphons, which are $D_1 = \{A2, A3, A4, A5, A6, B2, B4, B5, B6, R\_F2, R\_S\}$, $D_2 = \{A2, A4, A5, A6, B2, B3, B4, B5, B6, R\_F1, R\_S\}$ and $D_3 = \{A2, A4, A5, A6, B2, B4, B5, B6, R\_F1, R\_F2, R\_S\}$. Besides, every transition in the set $\Omega = \{TA2, TA3, TA4, TA5, TB2, TB3, TB4, TB5\}$ is an output transition of D_1, D_2 and D_3. After firing $TA1$ and $TB1$ from the initial marking, the state $A1 + B1 + R\_S$ is reached. This marking belongs to a livelock with other six markings. The reader can check that, unfortunately, there exists a firable transition in Ω for every marking in the livelock. A similar phenomenon can be observed for the SB SPQR net at the bottom of figure 4.

In general, livelocks are not a new phenomenon in the context of Petri net models for RAS. Even for $L - S^3PR$ nets, which are the simplest models in the family, deadlock freeness does not imply liveness [20]. However, deadlocks and livelocks always could be related to the existence of a siphon which was 'dry'. Unfortunately, this no longer holds. Another well-known result for simpler subclasses was that liveness equalled reversibility for nets with acceptable initial markings. For PC^{2}R, this is also also untrue, as figure 6 proves.

We believe that the transformation of PC^{2}R nets into SB SPQR can be useful to understand the phenomena from a structural point of view. Intuitively speaking, the concept of *lender resource* seems a simple yet powerful instrument which still remains to be fully explored. Still, SB SPQRs can present very complex behaviour. For instance, acceptably marked SB SPQR nets do not even hold the directness property [21] (which e.g. was true for S^{4}PR nets). Figure 7 shows a marked net which has no home states in spite of being live. This and

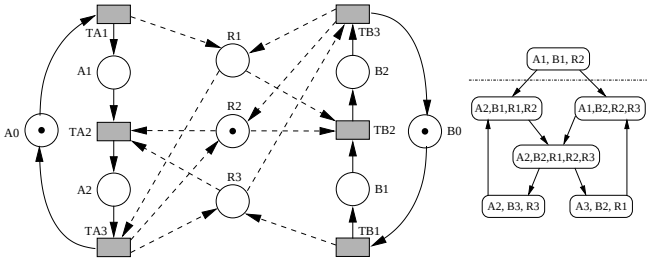

Fig. 6. An acceptably marked PC^{2}R which is live but not reversible

other properties are profoundly discussed (along with their implications) in a previous work [19].

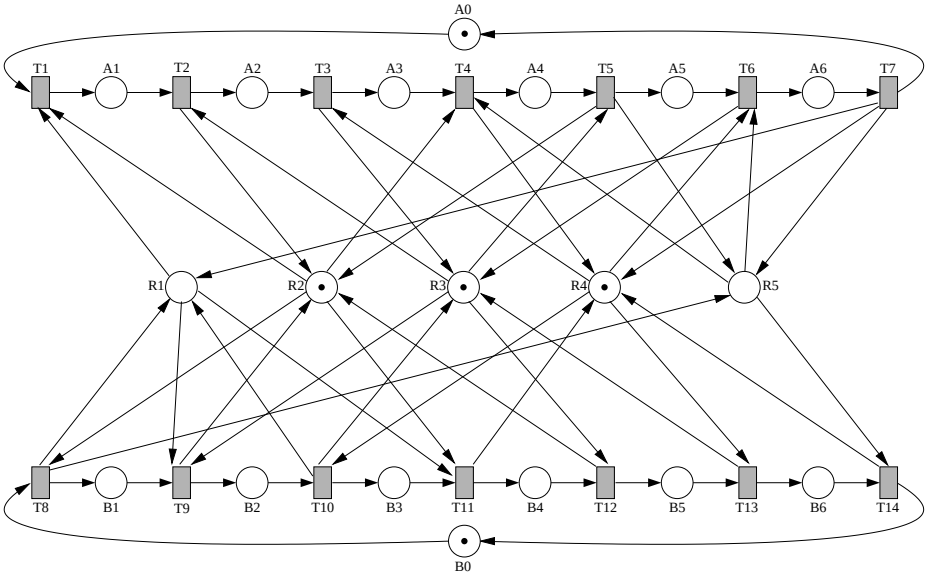

Fig. 7. A marked SB SPQR which is live but has no home states

6 Conclusion and future work

Although there exist a variety of Petri net classes for RAS, many of these definition efforts have been directed to obtain powerful theoretical results for the analysis and synthesis of this kind of systems. Nevertheless, we believe that the process of abstraction is a central issue in order to have useful models from a real-world point of view, and therefore requires careful attention. In this work,

we have followed that path and constructed a requirements list for obtaining an interesting Petri net subclass of RAS models applied to the software engineering domain. Considering that list, we defined the class of PC^2R nets, which fulfills those requirements while respecting the design philosophy on the RAS view of systems. We also introduced some useful transformation and class relations so as to locate the new class among the myriad of previous models. Finally we observed that the problem of liveness in the new context is non-trivial and presented some cases of bad behaviour which will be subject of subsequent work.

A Petri Nets: Basic definitions

A *place/transition net* (P/T net) is a 3-tuple $\mathcal{N} = \langle P, T, W \rangle$, where W is a total function $W : (P \times T) \cup (T \times P) \to \mathbb{N}$, being P, T non empty, finite and disjoint sets. Elements belonging to the sets P and T are called respectively *places* and *transitions*, or generally nodes. P/T nets can be represented as a directed bipartite graph, where places (transitions) are graphically denoted by circles (rectangles): let $p \in P$, $t \in T$, $u = W(p,t)$, $v = W(t,p)$, there is a directed arc, labelled u (v), beginning in p (t) and ending in t (p) iff $u \neq 0$ ($v \neq 0$).

The *preset* (*poset*) or set of input (output) nodes of a node $x \in P \cup T$ is denoted by ${}^\bullet x$ ($x^\bullet$), where ${}^\bullet x = \{y \in P \cup T \mid W(y,x) \neq 0\}$ ($x^\bullet = \{y \in P \cup T \mid W(x,y) \neq 0\}$). The preset (poset) of a set of nodes $X \subseteq P \cup T$ is denoted by ${}^\bullet X$ ($X^\bullet$), where ${}^\bullet X = \{y \mid y \in {}^\bullet x, x \in X\}$ ($X^\bullet = \{y \mid y \in x^\bullet, x \in X\}$

An *ordinary P/T net* is a net with unitary arc weights (i.e., W can be defined as a total function $(P \times T) \cup (T \times P) \to \{0,1\}$). If the arc weights can be non-unitary, the P/T net is also called *generalized*. A *state machine* is an ordinary net such that for every transition $t \in T$, $|{}^\bullet t| = |t^\bullet| = 1$. An *acyclic state machine* is an ordinary net such that for every transition $t \in T$, $|{}^\bullet t|, |t^\bullet| \leq 1$, and there is no circuit in it.

A self-loop place $p \in P$ is a place such that $p \in p^{\bullet\bullet}$. A *pure P/T net* (also self-loop free P/T net) is a net with no self-loop places. In pure P/T nets, the net can be also defined by the 3-tuple $\mathcal{N} = \langle P, T, C \rangle$, where C is called the *incidence matrix*, $C[p,t] = W(p,t) - W(t,p)$. Nets with self-loop places can be easily transformed into pure P/T nets without altering most significant behavioural properties, such as liveness, as shown in figure 8.

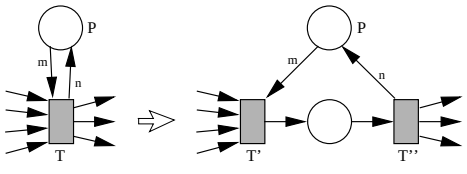

Fig. 8. Removing self-loop places

A *p-flow* is a vector $Y \in \mathbb{Z}^{|P|}$, $Y \neq \mathbf{0}$, which is a left annuler of the incidence matrix, $Y \cdot C = \mathbf{0}$. The support of a p-flow is denoted $\|Y\|$, and its places are said to be covered by Y. A *p-semiflow* is a non-negative p-flow, i.e. a p-flow such that $Y \in \mathbb{N}^{|P|}$. The P/T net $\mathcal{N}$ is *conservative* iff every place is covered by a p-semiflow. A *minimal p-semiflow* is a p-semiflow such that the g.c.d of its non-null components is one and its support $\|Y\|$ is not an strict superset of the support of another p-semiflow.

A set of places $D \subseteq P$ is a *siphon* iff every place $p \in {}^\bullet D$ holds $p \in D^\bullet$. The support of a p-semiflow is a siphon but the opposite does not hold in general.

Let $\mathcal{N} = \langle P, T, W \rangle$ be a P/T net, and let $P' \subseteq P$ and $T' \subseteq T$, where $P', T' \neq \emptyset$. The P/T net $\mathcal{N}' = \langle P', T', W' \rangle$ is the subnet generated by P', T' iff $W'(x, y) \Leftrightarrow W(x, y)$, for every pair of nodes $x, y \in P' \cup T'$.

A *marking* m of a P/T net $\mathcal{N}$ is a vector $\mathbb{N}^{|P|}$, assigning a finite number of marks $m[p]$ (called *tokens*) to every place $p \in P$. Tokens are represented by black dots within the places. The *support* of a marking, $\|m\|$, is the set of places which are marked in m, i.e. $\|m\| = \{p \in P \mid m[p] \neq 0\}$. We define a *marked P/T net* (also P/T net system) as the pair $\langle \mathcal{N}, m_0 \rangle$, where $\mathcal{N}$ is a P/T net, and m_0 is a marking for $\mathcal{N}$, also called *initial marking*. $\mathcal{N}$ is said to be the structure of the system, while m_0 represents the system state.

Let $\langle \mathcal{N}, m_0 \rangle$ be a marked P/T net. A transition $t \in T$ is *enabled* (also *firable*) iff $\forall p \in {}^\bullet t \ : \ m_0[p] \geq W(p, t)$, which is denoted by $m_0[t\rangle$. The *firing* of an enabled transition $t \in T$ changes the system state to $\langle \mathcal{N}, m_1 \rangle$, where $\forall p \in P \ : \ m_1[p] = m_0[p] + C[p, t]$, and is denoted by $m_0[t\rangle m_1$. A *firing sequence* σ from $\langle \mathcal{N}, m_0 \rangle$ is a non-empty sequence of transitions $\sigma = t_1\, t_2 \ldots t_k$ such that $m_0[t_1\rangle m_1[t_2\rangle \ldots m_{k-1}[t_k\rangle$. The firing of σ is denoted by $m_0[\sigma\rangle t_k$. A marking m is *reachable* from $\langle \mathcal{N}, m_0 \rangle$ iff there exists a firing sequence σ such that $m_0[\sigma\rangle m$. The *reachability set* $RS(\mathcal{N}, m_0)$ is the set of reachable markings, i.e. $RS(\mathcal{N}, m_0) = \{m \mid \exists\, \sigma \ : \ m_0[\sigma\rangle m\}$.

A transition $t \in T$ is *live* iff for every reachable marking $m \in RS(\mathcal{N}, m_0)$, $\exists m' \in RS(\mathcal{N}, m)$ such that $m'[t\rangle$. The system $\langle \mathcal{N}, m_0 \rangle$ is *live* iff every transition is live. Otherwise, $\langle \mathcal{N}, m_0 \rangle$ is *non-live*. A transition $t \in T$ is *dead* iff there is no reachable marking $m \in RS(\mathcal{N}, m_0)$ such that $m[t\rangle$. The system $\langle \mathcal{N}, m_0 \rangle$ is a *total deadlock* iff every transition is dead, i.e. no transition is firable. A *home state* m_k is a marking such that it is reachable from every reachable marking, i.e. $\forall m \in RS(\mathcal{N}, m_0) \ : \ m_k \in RS(\mathcal{N}, m)$. The net system $\langle \mathcal{N}, m_0 \rangle$ is *reversible* iff m_0 is a home state.

References

1. Lautenbach, K., Thiagarajan, P.S.: Analysis of a resource allocation problem using Petri nets. In Syre, J.C., ed.: Proc. of the 1st European Conf. on Parallel and Distributed Processing, Toulouse, Cepadues Editions (1979) 260–266
2. Colom, J.M.: The resource allocation problem in flexible manufacturing systems. In van der Aalst, W-M-P. and Best, E., ed.: Proc. of the 24th Int. Conf. on Applications and Theory of Petri Nets. Volume 2679 of LNCS., Eindhoven, Netherlands, Springer–Verlag (June 2003) 23–35

3. Li, Z.W., Zhou, M.C.: Deadlock Resolution in Automated Manufacturing Systems: A Novel Petri Net Approach. Springer, New York, USA (2009)
4. Coffman, E.G., Elphick, M., Shoshani, A.: System deadlocks. ACM Computing Surveys **3**(2) (1971) 67–78
5. Reveliotis, S.A., Lawley, M.A., Ferreira, P.M.: Polynomial complexity deadlock avoidance policies for sequential resource allocation systems. IEEE Transactions on Automatic Control **42**(10) (1997) 1344–1357
6. Fanti, M.P., Maione, B., Mascolo, S., Turchiano, B.: Event-based feedback control for deadlock avoidance in flexible production systems. IEEE Transactions on Robotics and Automation **13**(3) (1997) 347–363
7. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE **77**(4) (1989) 541–580
8. Ezpeleta, J., Colom, J.M., Martínez, J.: A Petri net based deadlock prevention policy for flexible manufacturing systems. IEEE Transactions on Robotics and Automation **11**(2) (April 1995) 173–184
9. Ezpeleta, J., Recalde, L.: A deadlock avoidance approach for non–sequential resource allocation systems. IEEE Transactions on Systems, Man and Cybernetics. Part–A: Systems and Humans **34**(1) (January 2004)
10. Tricas, F., García-Valles, F., Colom, J.M., Ezpeleta, J.: A Petri net structure-based deadlock prevention solution for sequential resource allocation systems. In: Proc. of the 2005 Int. Conf. on Robotics and Automation (ICRA), Barcelona, Spain, IEEE (April 2005) 272–278
11. Park, J., Reveliotis, S.A.: Deadlock avoidance in sequential resource allocation systems with multiple resource acquisitions and flexible routings. IEEE Transactions on Automatic Control **46**(10) (2001) 1572–1583
12. 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
13. Xie, X., Jeng, M.D.: ERCN-merged nets and their analysis using siphons. IEEE Transactions on Robotics and Automation **29**(4) (1999) 692–703
14. Jeng, M.D., Xie, X.L., Peng, M.Y.: Process nets with resources for manufacturing modeling and their analysis. IEEE Transactions on Robotics **18**(6) (2002) 875–889
15. Hu, H.S., Zhou, M.C., Li, Z.W.: Liveness enforcing supervision of video streaming systems using non-sequential Petri nets. IEEE Transactions on Multimedia **11**(8) (December 2009) 1446–1456
16. Wang, Y., Liao, H., Reveliotis, S., Kelly, T., Mahlke, S., Lafortune, S.: Gadara nets: Modeling and analyzing lock allocation for deadlock avoidance in multithreaded software. In: Proc. of the 49th IEEE Conf. on Decision and Control, Atlanta, Georgia, USA, IEEE (December 2009) 4971–4976
17. Hoare, C.A.R.: Communicating sequential processes. Communications of the ACM **21**(8) (1978) 666–677
18. Harel, D.: On folk theorems. Communications of the ACM **23**(7) (1980) 379–389
19. López-Grao, J.P., Colom, J.M.: Lender processes competing for shared resources: Beyond the S^4PR paradigm. In: Proc. of the 2006 Int. Conf. on Systems, Man and Cybernetics, IEEE (October 2006) 3052–3059
20. García-Vallés, F.: Contributions to the structural and symbolic analysis of place/transition nets with applications to flexible manufacturing systems and asynchronous circuits. PhD thesis, University of Zaragoza, Zaragoza (April 1999)
21. Best, E., Voss, K.: Free choice systems have home states. Acta Informatica 21 (1984) 89–100