Conceptual Building Blocks for Modeling Reconfiguration of Component-Based Systems Using Petri Nets*

Y. Hafidi [a,1], E.P. de Vink [a,2]

[a] *Eindhoven University of Technology*
Department of Mathematics & Computer Science
P.O. Box 512, 5600MB Eindhoven, the Netherlands

Abstract

This paper deals with the formal modeling of dynamically reconfigurable systems using Petri nets. Dynamic reconfiguration provides to a system the ability to change the behavior of its components at run-time without a system shut-down. By transferring the concepts of the coordination modeling language Paradigm to the setting of Petri nets a framework is obtained for the modeling of component-based systems. The framework then allows for reasoning about coordination of components on one level of abstraction and for analysis of reconfiguration on another level of abstraction. This factorization will be beneficial to subsequent formal assessment. A workers and scheduler example, the well-known dining philosophers, and a Festo MPS casestudy serve as illustrations.

Keywords: Coordination, Dynamic reconfiguration, Petri nets, Formal specification, Foundations for software architecture design.

1. Introduction

Petri nets have a strong track record for modelling systems and system analysis, both in theory and practice [1, 2, 3]. Reconfiguration of systems, also referred to as dynamic adaptation or evolution on-the-fly, enjoys a rising interest both from academia and industry. However, for the modeling of reconfiguration with Petri nets no approach has been widely adopted so far. Within the formalism, reconfiguration and the subsequent behavioral changes often seem cross cutting and hard to pinpoint exactly. From a modeling language point of view, one of the issues in modeling reconfigurable systems is to identify which concepts are needed

*Contribution in honor of Luís Soares Barbosa on the occasion of his 60th birthday.

[1]This work was partially supported by the MACHINAIDE project (ITEA3, No. 18030).

[2]Corresponding author, `evink@win.tue.nl`.

to describe a reconfigurable system concisely. To the best of our knowledge, a convenient, convincing, and methodological framework for Petri nets is lacking so far.

In this paper we propose to transfer the terminology of the coordination modeling language Paradigm, that is based on labeled transition systems, to the setting of Petri nets. Our motivation for using standard Petri nets rather than reconfigurable Petri nets lies in the fact that there is a lack of tools supporting modeling, simulation, and verification of reconfigurable Petri nets as proposed e.g. in [4, 5, 6], while for standard Petri nets several long-standing toolsuits are available, e.g. [7, 8, 9]. In addition, it seems straightforward to apply the proposed concepts to other applications that employ different Petri net extensions.

Starting from [10], Groenewegen and co-workers have been studying dynamic adaptation of component-based software systems in the context of the coordination modeling language Paradigm. See, e.g., the publications [11, 12, 13, 14]. Central to the approach of Paradigm is (i) the decomposition of the behavior of a single component into overlapping sub-behavior, referred to as phases, (ii) signalling by a component that it has arrived near the end of its current phase in a so-called trap, and (iii) what are called consistency rules, multi-party interaction schemes that govern the coordination of synchronized phase transfers of components.

The mathematical underpinning of Paradigm [15] is based on labeled transition systems and therefore rests on an interleaving semantics. In our current investigation we seek to exploit Paradigm's concepts in the setting of Petri nets, where the notion of a distributed state is better reflected in the marking of a net. Often, interleaving vs. truly concurrent modeling can be interchanged, give and take. However, in specific application domains, e.g. robot swarms, it will be beneficial to have distributed control and decentralized coordination.

For our purposes here we consider two types of augmented Petri nets: component nets and coordinator nets (but their superposition is allowed as well). In a component net, modelling the behavior of a component in general, there are two kinds of decorations: transitions are decorated with modes that restrict a component to a specific sub-behavior (see characteristic (i) above), and places are decorated with signals (see characteristic (ii) above). A coordinator net in the system orchestrates the coordination; it has the task of monitoring and regulating the interaction of the components based on the signals they provide and the modes they are in. This implements the system's consistency rules (see characteristic (iii) above). Like components, the coordinator is also given as (a subnet of) a Petri net with decorations. For the coordinator transitions can be decorated by signals, indicating that the transition can only be taken when the signals are switched on by the components they belong to, and transitions can be decorated by mode transfers, a directive that the components are instructed to change their

mode, thereby obtaining permission to proceed, to leave the sub-behavior they were trapped in.

In short, the contributions of this paper are the following. Imposing the Paradigm concepts leads to a class of decorated Petri nets, with the decorations for coordination purposes. This allows for a compact representation of models and a clear separation of local component behavior, coordination among components, and migration of the system. The approach is introduced and illustrated by means of three examples. A workers and scheduler example explains how the decorations of the Petri nets are used to organize coordination among components. The next example is a variation of the well-known dining philosophers; the example is used to discuss dynamic adaptation of components. The third example reports on a case study of a mechatronic application with components that may fail and hence requiring dynamic reconfiguration. A formal definition of a component net and of a coordinater net are deferred to the appendix. However, it is noted that the enrichment of Petri nets doesn't add to the expressivity of the formalism; decorated nets can all be expressed as standard Petri nets straightforwardly by adding appropriate places and transitions for signals and modes.

Related work Connections with Petri nets have been addressed at various places in the context of coordination languages. E.g., [16] provides various Reo connectors with a semantics based on zero-safe nets. Graph rewriting can be used for Reo to handle dynamic reconfiguration [17]. For cyber-physical systems (static) reconfiguration can be based on hybrid Reo networks and analyzed using statistic model checking [18]. Fundamental to the approach is that a Reo network provides the coordinating 'glue' between the components, which are treated as black-boxes. In the present paper we take components as Petri nets, or rather subnets describing component behavior, as starting point for the modelling. The decorations that are added by our formalism provide the basis for subsequent formal analysis, which acts at a higher level of abstraction than the standard token game.

Other Petri net-based approaches to formal modeling and verification include self-modifying nets [4], reconfigurable Petri nets [5], and reconfigurable timed net condition/event systems (R-TNCES) [6]. See [19], for example, for an overview. Depending on the modeling language, reconfiguration can be described implicitly or explicitly in such models. A number of casestudies verified reconfigurable systems using formal methods. Because of the specific behavior of reconfigurable systems, distinguishing 'old' or 'as-is' behavior from 'intermediate' or 'migration' behavior from 'new' or 'to-be' behavior, their verification becomes often expensive in terms of time and memory. To handle the intrinsic complexity of the problem, some works focus on methods that reduce redundancies as much as possible during the verification process [20]. This is also our long-term aim with the casting of

Paradigm-style modeling in the setting of Petri nets, to make it amendable for model checking with the mCRL2 toolset [21]. It is conjectured that the structural decomposition as captured by a Paradigm description, hiding local component behavior behind signals and modes, provides a description of the interaction among components at a higher level of abstraction, which hopefully yields smaller state spaces.

Section 2 introduces the main concepts of the modelling approach of Paradigm as based on Petri nets. It discusses a critical section problem involving a number of workers and a scheduler. Initially, the workers are granted access to the critical section by the scheduler non-deterministically. The access is on-the-fly reconfigured to be done in a round-robin fashion. Section 3 discusses the dining philosophers problem for an instance with four philosophers. First, a system is described that has deadlock and can be reset by a controller. Next, it is explained how dynamic reconfiguration can be applied to achieve a system that is deadlock and starvation free, under fairness conditions. Section 4 provides a treatment of dynamic reconfiguration in a Festo MPS case study. A production system with two drills may switch between high and light performance dependent on user requests or possible failure of the drills. In case one of the drills breaks, the controller switches automatically from high to light performance and adapts all the components accordingly. Once a broken drill is repaired, the controller switches the system back to the performance level requested by the user. In Section 5 we show that the proposed concepts of Paradigm can be casted in terms of standard, non-decorated Petri nets, after which we wrap up with concluding remarks.

2. Workers and scheduler

An illustration in a setting with Petri nets of the basic concepts for modeling (dynamic) coordination with Paradigm is given by a critical section example involving three workers. Access to the critical section is to be granted by a scheduler on request of the worker. The scheduler withdraws the permission for access to the critical section once the worker has left it. The so-called component net of an individual worker is given in Figure 1; a so-called coordinator net of the scheduler is given in Figure 2.

We see in Figure 1 that a worker cycles through four states. In state `nCrit` the worker performs other, non-critical activity, in state `pre` the worker is prepared to enter the critical section, in state `crit` the worker is entering, residing in and leaving the critical section, in state `post` the worker has left the critical section, and then continues to state `nCrit` again. As to the decorations of transitions in the component net of a worker (labels *NoPm* and *Perm* in green in the figure), a worker can be in one of two modes, mode *NoPm* for having no permission to

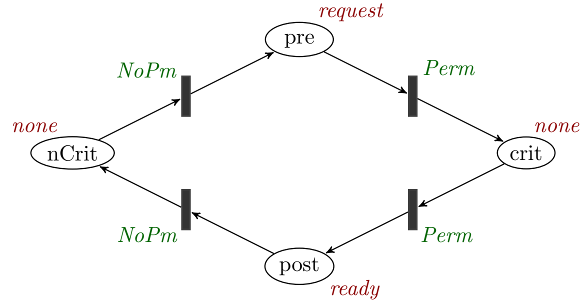


Figure 1: Component net for individual worker: places decorated with signals; transitions decorated with modes.

enter the critical section and mode *Perm* for having permission to do so. The mode *NoPm* decorates the transitions from `post` to `nCrit` and from `nCrit` to `pre`. Thus, after having reached state `pre` from state `nCrit` in mode *NoPm*, first a so-called mode transfer, viz. from mode *NoPm* to mode *Perm*, has to take place before the worker is able to fire the transition to state `nCrit`. So, progress of the worker is upheld till permission by the scheduler, which is in control of the modes, is given. A similar situation arises in state `post`. The two transitions from `pre` to `crit` and `crit` to `post` require the worker to be in mode *Perm* as indicated by the label of the two transitions. However, to proceed from state `post` the mode of the worker should first be changed from *Perm* to *NoPm*. This guarantees that the scheduler becomes aware that the worker has left the critical section. Thus, the mode transfers are controlled by the scheduler, which we discuss below. It is noted that the state `post` can left out (leading to a less symmetric example).

The states of the component net of Figure 1 are decorated with signal names (in red). Here, a worker has three signals, viz. *request*, *ready*, and *none*, the latter signifying absence of the two other signals. The signal is said to be switched on if the worker is in a state decorated with that signal; the signal is said to be switched off if the worker is in a state decorated by a signal name different from that signal. Note, in the situation that the signal *request* is switched on, the worker has no transition that is permitted in mode *NoPm* in which the worker has reached state `pre`. Similarly, upon moving to state `post`, possible in mode *Perm* only, the signal *ready* is switched on and the worker has no transition available to continue until a mode transfer to mode *NoPm* has taken place. Thus, with the signal *request* the worker requests for access to the critical section, and with the signal *ready* the worker indicates that it doesn't need to access the critical section any longer. The scheduler can act on these (mutually excluding) signals accordingly, taking the activity of other workers into account.

The behavior of the scheduler, coordinating the three workers, is described by the

coordinator net in Figure 2. (Note, the state `idle` is repeated with dotted lines at the bottom of the figure for clarity of presentation, avoiding arrows looping back from the bottom to the top of the figure. The dotted state `idle` at the bottom is to be identified with the state `idle` at the top.) In this coordinator net only transitions have decorations, which can be a signal name, e.g. *none*, or a mode transfer, e.g. $\mathit{NoPm} \to \mathit{Perm}$. A transition decorated by one or more signals can be taken if, besides the standard firing condition in a Petri net, the specific signal or signals are switched on. Thus, for example, the transition from state `check1` to state `granted1` can be taken when the scheduler is in state `check1`, i.e. in terms of Petri nets, the place holds a token, and moreover, the component representing worker 1 has its signal *request* switched on. Thus, only if worker 1 'requests' access to the critical section, the scheduler will proceed from state `check1` to state `granted1`, indicating that the scheduler has decided to grant worker 1 access.

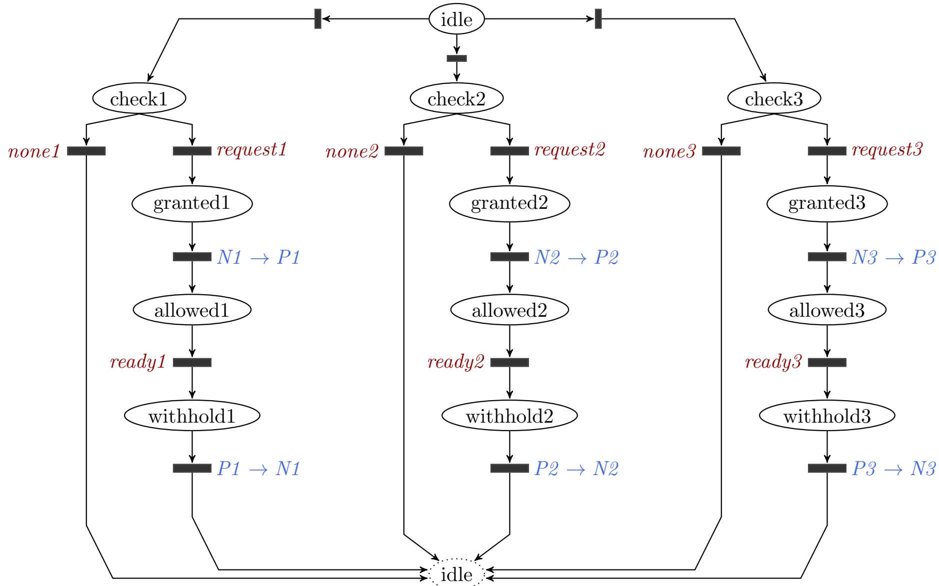


Figure 2: Coordinator net of the scheduler: the labels $N{\to}P$ and $P{\to}N$ abbreviate mode transfers $\mathit{NoPm} \to \mathit{Perm}$ and $\mathit{Perm} \to \mathit{NoPm}$, respectively.

The transition of the scheduler from its state `granted1` to state `allowed1` is an example of a transition decorated with a mode transfer. Here the transfer from mode *NoPm* to mode *Perm* for worker 1 is indicated by the label $N1{\to}P1$. The transition can be taken if the scheduler is in state `granted1`; no further condition needs to be met. By taking the transition, the scheduler goes to its state `allowed1`, while simultaneously the mode of worker 1 changes from mode *NoPm* to mode *Perm*. Thus, directives for a mode transfer from a coordinator to

a component take effect synchronously. Once worker 1 has changed to mode *Perm* it can proceed to its state `crit` and reach state `post` where it switches it signal *ready* on. But it cannot proceed from there on its own, since worker 1 cannot change its mode from mode *Perm* to mode *NoPm* by itself. The coordinator is in charge of this, not the component. Now, with the signal *ready* on, the scheduler can proceed to its state `withhold1` where it instructs indeed worker 1 to change its mode to mode *NoPm*, and worker 1 can proceed. Since it had already left the critical section, for otherwise the signal *ready* wouldn't have been switched on, the scheduler may grant access to an other worker (or to worker 1 again) after a transition from state `idle` to one of the states `check1` to `check3`.

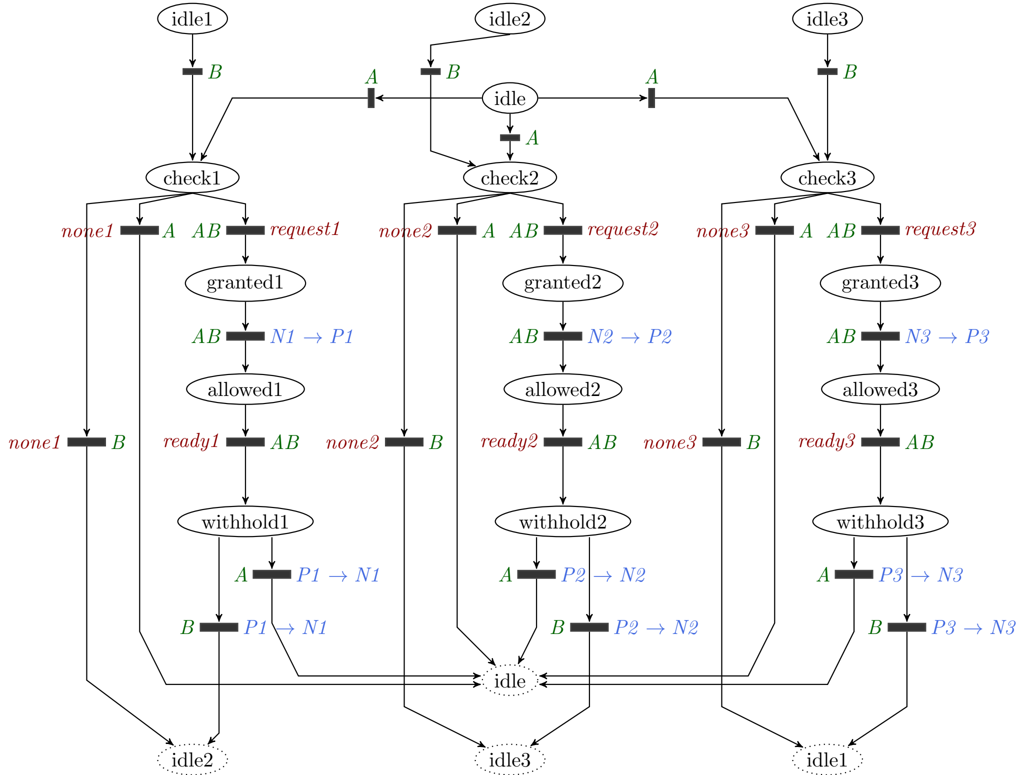


Figure 3: Reconfiguration net for the scheduler: A and B indicate the *asIs* mode and *toBe* mode of the scheduler; $N \rightarrow P$ and $P \rightarrow N$ abbreviate mode transfers $NoPm \rightarrow Perm$ and $Perm \rightarrow NoPm$, respectively.

Clearly, in the current set-up, given the non-determinism of the scheduler in its state `idle`, one or even two workers can always be excluded from access as an extreme case. In the situation as given, an acceptable fair alternative may be that the scheduler checks in a round-robin fashion, rather than non-deterministically,

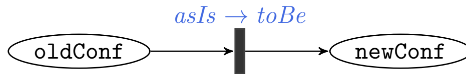


Figure 4: Coordinator net for the reconfiguration manager of the workers and scheduler example.

whether the workers are requesting access or not, to avoid local deadlock and overtaking.

To achieve this we can, like we did for the workers, endow the scheduler with modes too and put a reconfiguration manager into action to coordinate the reconfiguration of the scheduler. Thus, we have the workers whose modes are coordinated by the scheduler and the scheduler whose modes, that we introduce below, are coordinated by a new coordinator called reconfiguration manager. The scheduler is coordinator with respect to the workers and is component with respect to the reconfiguration manager. For this example it suffices to distinguish for the scheduler

(i) a non-deterministic mode, which is the 'old' or *as-is* situation to be phased out as given (indicated by A in Figure 3 abbreviating *asIs*), and

(ii) a round-robin mode, which is the 'new' or *to-be* situation to be reconfigured to (indicated by B in Figure 3 abbreviating *toBe*).

On top of this, a rather simple reconfiguration manager will guide the scheduler from its as-is 'configuration' to the to-be 'configuration'. It has two states, `oldConf` and `newConf`, and one transition from `oldConf` to `newConf` that enforces a change of modes of the scheduler, a mode transfer from the non-deterministic mode *asIs*, the as-is behavior, to the round-robin mode *toBe*, the to-be behavior. See Figure 4. Note that we do not require that the reconfiguration manager, nor the new places and transitions for the coordinator as well as the two new modes, were already present when the system was started. This behavior is added on-the-fly. We assume that the run-time can incorporate new places, transitions, and flow to provide the new behavior; places, transitions, and arcs that aren't desired anymore can be removed once having become unreachable or dependent on obsolete modes only.

Figure 3 provides the updated diagram for the scheduler, incorporating the reconfiguration as desired. Note that the diagram is a superposition of a component net and coordinator net. The scheduler is coordinating the workers and therefore synchronizes on their signals *none*, *request*, and *ready*, and prescribes transfer between their modes *NoPm* and *Perm* (in red and blue in the figure); the scheduler itself is being coordinated by the reconfiguration manager and has been imposed (upon a design decision at some point in time to change the scheduling) two modes, *asIs* and *toBe* (corresponding to the labels A and B in green in the

figure). Three states `idle1` to `idle3` are added on top of the figure (with copies for presentation purpose at the bottom) as well as transitions leaving and entering these states. Also, transitions are now labeled with A for the *asIs* mode, with B for the *toBe* mode, or with AB in case the transition is allowed in both modes. E.g., the transition from state `idle1` is only possible in the to-be mode, when reconfiguration has become in effect; the transitions leaving from state `idle` are only possible in the as-is mode, because the state `idle` will be phased out. Note, all incoming transitions for state `idle` can be fired in mode *asIs* only.

The actual reconfiguration happens either at the transition on the signal *none* from a state `check1` to `check3` or at the transition leaving any of the three states `withhold1` to `withhold3`. Note, that for these two types of situations, the transition has become dependent on the current mode imposed on the scheduler by the reconfiguration manager. For example, from `check1` on signal *none* the transitions are different in the *asIs* mode from those in the *toBe* mode. On the one hand, in the *asIs* mode, the scheduler continues as it is used to do, viz. moving to state `idle`. On the other hand, in the *toBe* mode, the scheduler moves from the checking of worker 1 via the new state `idle2` to the processing of worker 2 and moves into the round-robin scheme. (Note, e.g., that in the left part of Figure 3 has the state `idle1` for worker 1 on top, and that it has the state `idle2` for worker 2 at the bottom.) For the states `check2` and `check3` this is similar. Downward from `check1` on signal *request* the transitions in the *asIs* mode and the *toBe* mode are the same. In both modes worker 1 obtains access to the critical section. However, upon leaving the handling of worker 1 in state `withhold1`, different transitions are offered in the two modes. In the *asIs* mode control returns to state `idle`, but in the *toBe* mode control moves to the idle state `idle2` of the worker 2 according the round-robin scheme that is adopted hence forward. Thus, after the reconfiguration manager transfers the scheduler from *asIs* mode to *toBe* mode, workers are checked in round-robin fashion for a request to enter the critical section, possibly after one handling out out of the scheme.

To summarize the above, in the Paradigm approach with Petri nets the behavior of components is split up in possibly overlapping modes with signals that indicate that a transfer to another mode is needed. A coordinator regulates if and when mode transfer of a component takes places. The internal behavior of a component is hidden from the coordinator. On a higher level of abstraction, a coordinator can also be endowed with modes, in particular for reconfiguration purposes. The workers and scheduler example provides a simple case of dynamic reconfiguration. We argue that the idea of interpreting reconfiguration as coordination allows for much more variation.

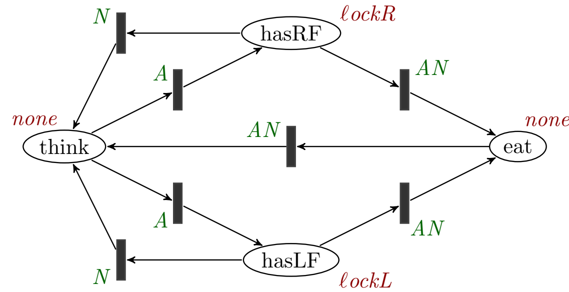


Figure 5: Component net for an individual philosopher.

3. Dining philosophers

In the previous section an example of dynamic reconfiguration is given that acts on the coordinator, viz. the scheduler, only. In this section we give an illustration of dynamic reconfiguration for a variant of the dining philosophers problem affecting all components (and their coordinator). The initial set-up has deadlock and a preliminary mechanism to overcome this. Then, while the system is running, new behavior (that was not present initially) is added to the components to prevent further deadlock. The example assumes, for convenience, four philosophers sitting around the table with one fork between each two neighboring philosophers. As usual, a philosopher can eat only when they has taken both forks at their sides. To further simplify the presentation here, the modeling of the forks has been suppressed. (A Petri net of one place to distinguish the fork lying on the table (with token) or not (without token) and involved with the philosopher's transition for picking up the fork and with the two transitions for putting down the fork for its left philosopher as well as for the relevant transitions of its right philosopher will do.) A controller is put in place to resolve deadlock in the system, would it occur.

Initially, we distinguish two modes for each philosopher component: (i) the A mode, A abbreviating *Allowed*, where a philosopher is allowed to take a fork from either of their sides in any order of choice; (ii) the N mode, N abbreviating *Not Allowed*, where a philosopher is not allowed to pick up a first fork, but if they have one fork already they either can take the other one if available, or put the fork down. In addition, if a philosopher is eating, they can finish eating and put both forks down on the table when done. So, in the A mode there are no restrictions on the think-eat cycle of the philosophers. However, when having exactly one fork, a philosopher is supposed to continue toward eating. In the N mode philosophers are urged to think if not doing so already. When having exactly one fork, a philosopher may put the fork down or pick up the other fork if available. See Figure 5 where the component net for a philosopher is given.

Clearly, the system of four philosophers and forks can get into local or global

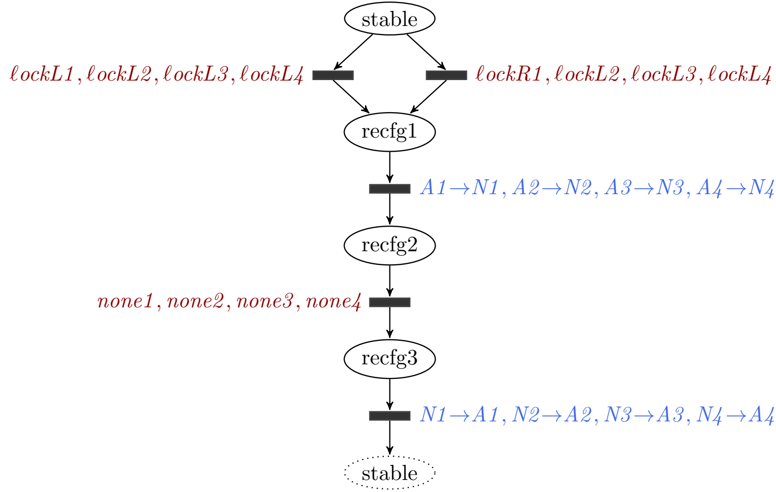


Figure 6: Coordinator net for controler four 4 philosophers implementing a reset.

deadlock since no policy for fork usage applies. In order to resolve deadlock in the system, a controller is put into place. See Figure 6. The controller monitors the signals of the philosophers to detect whether all philosophers are either having a fork in their left hand (all philosophers signalling $\ell ockL$) or all are having a fork in their right hand (all philosophers signalling $\ell ockR$). If such is the case, the transition for the controller from `stable` to `recfg1` becomes enabled. From there a reset of the philosophers takes places in three steps, thus reconfiguring the running system. First, the controller transfers each philosopher from mode A to mode N, enforcing the philosophers to move asynchronously from the place where they are holding a left or right fork, to the place where they think, without holding any fork. Thus, effectively, all philosophers are requested to put down the forks and wait while thinking for a transition to become enabled. When all philosophers have let go of their fork, before or after having had the opportunity to eat, as indicated by the signal being *none* for all four philosophers, the controller moves from `recfg2` to `recfg3`. Now, the philosophers hold no fork anymore and the deadlock has been resolved. In the subsequent transition of the controller, all four philosophers are put back to normal, into A mode where they may pick up a fork of their choice again.

The approach of resetting the philosophers in case of a deadlock doesn't resolve the problem fundamentally (nor does it address starvation). In order to do so, we will describe a reconfiguration of the system to the situation where two opposite philosophers are picking up their left fork first and the two other opposite philosophers are picking up their right fork first. The reconfiguration will be on-the-fly. The philosophers aren't forced into contemplation for the system change

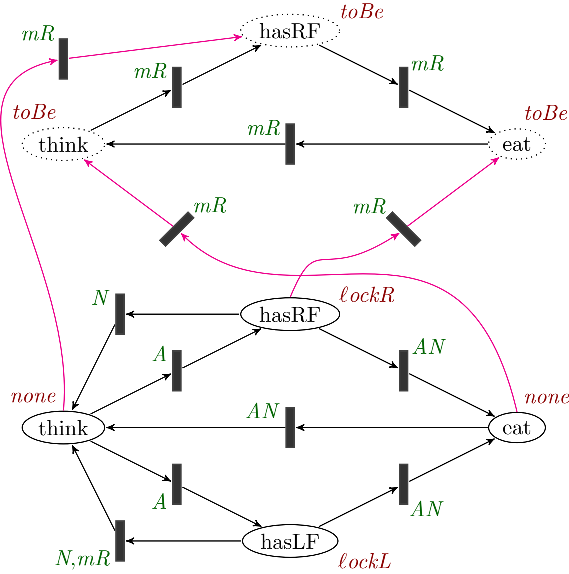


Figure 7: Component net for philosopher migrating to right-hand first behavior. Lower part original, as-is behavior, top part to-be behavior that is added.

to happen: the system will migrate to the new behavior while the think-eat cycle continues uninterruptedly.

First the behavior of the philosophers is extended as described in Figure 7 for the case of migration to a right-hand first philosopher; migration to a left-hand first philosopher is similar. The run-time system is assumed to be able to incorporate the six new transitions and related flow per philosopher. The extension isn't supposed to have been present or pre-programmed when the system was started; the new transitions are added on-the-fly. To better separate in the presentation the original, as-is behavior, from the to-be behavior, the figure has copies of the states `think`, `hasRF`, and `eat`, although this is for clarity only. The philosophers are augmented with a new mode with a left-first policy or a mode with a right-first policy, indicated by the labels mL and mR in Figures 7 and 8. The mode includes transitions to the new behavior as well as transitions of the behavior thereof. Thus, once the mR mode is enabled, the philosopher can do a transition from the as-is `think` to the to-be `hasRF`, from the as-is `hasRF` to the to-be `eat`, and from the as-is `eat` to the to-be `think`. Once arrived in the to-be behavior, the philosophers is bound to remain there, in this case preferring their right fork over their left. A subtlety is the decoration of the transition in the as-is behavior from `hasLF` to `think`, not only for the N mode but now also for the mR mode. We come back to this later.

The controller synchronizes the start and end of the migration. If the prepara-

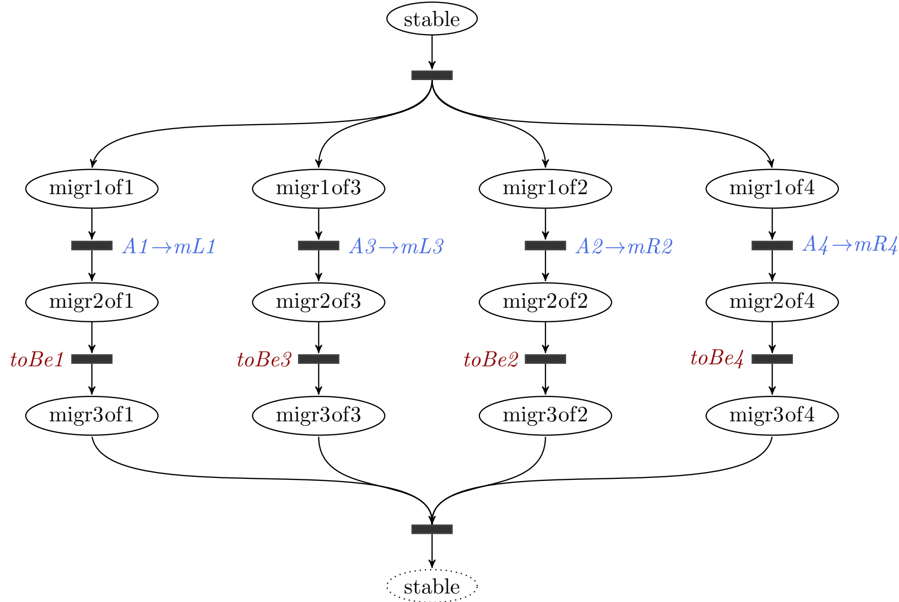


Figure 8: Coordinator net for migrating controller of four philosophers, enforcing the first and third philosopher to pick up their left fork first, and the second and fourth philosopher to pick up their right fork first.

tion for the migration, in particular the extension of the behavior of the philosophers has finished by the mechanism that is assumed to be in place for this, the controller initiates from its `stable` state four threads simultaneously, one for each philosopher separately. Along such a thread, first the philosopher is transferred from the A mode to the mL mode or the mR mode, dependent on the parity of their number. If the philosopher confirms that this has become effective via the signal *toBe*, the migration for this philosopher is completed. Migration is complete overall when all philosophers have signaled confirmation of their transfer and thus have successfully migrated. The controller returns to its stable state. Note, the above is all on-the-fly, without any specific stopping of a philosopher or synchronization between pairs of philosophers for this purpose. Also, we omitted a reconfiguration manager that nudges the controller from the behavior of Figure 6 to that of Figure 8, which is very similar to the one for reconfiguration in the workers and scheduler example.

After migration is done, philosopher 1 and philosopher 3 first take their left fork, while philosopher 2 and philosopher 4 will first take their right fork. So, in the to-be situation, if philosopher 1 has taken up its left fork, philosopher 2 is thinking. Therefore, philosopher 3 can either think, eat or pick up their left fork, if available. In the latter case, philosopher 4 will be thinking and hence philosopher 1 can proceed picking up the right fork to eat. In the former case,

with philosopher 3 thinking, philosopher 4 can finishing picking up their fork, eating and return their forks, in particular their left fork, which is the right fork for philosopher 1, back to the table. Thus, under a fairness assumption, philosopher 1 can proceed. Similar and partly symmetric reasoning applies to the other philosophers having picked up their first fork.

Although the to-be situation is just fine, a point to consider is whether the to-be situation will be reached at all, in particular that no deadlock can occur when migration has started but hasn't finished. Such may arise in principle when some philosophers adhere to their old, as-is behavior while others follow the new, to-be behavior. Here the extra transition, or rather extra decoration, from the as-is state `hasLF` to the as-is state `think` allowed in the mR mode, and its counterpart in the mL mode, come into play. As an extreme situation, suppose in migration mode all four philosophers are holding a fork in their left hand. Philosopher 1 and philosopher 3 do so rightly but philosopher 2 and philosopher 4 still exhibit as-is behavior and reside in the as-is state `hasLF` which doesn't exist in their to-be behavior. See Figure 7. The mR mode provides philosopher 2 and philosopher 4 a transition from `hasLF` back to `think`, putting back the fork on the table and allowing philosopher 1 and philosopher 3 to proceed. When philosopher 2 and philosopher 4 are picking up a fork again, they have moved to to-be behavior excluding the situation of taking a left fork without having the right fork already.

The dining philosophers example shows a more elaborate pattern for dynamic reconfiguration where the behaviors of both coordinator and components are augmented and gradually, via a switch in their modes, evolve from as-is into to-be behavior. The machinery responsible for the augmentation of behavior is not within in the scope of the modelling. However, one may require from such a machinery that obsolete behavior, i.e. transitions that are phased out and isolated states, is garbage-collected, i.e. removed eventually. The main concern of our modelling here is to make the necessary coordination and the change thereof explicit in order to be able to reason about it and establish its correctness.

4. Festo MPS

The Festo MPS casestudy is a benchmark for reconfigurable production systems that is frequently used in the literature [22]. In this paper, Festo MPS will be used to further explain and illustrate the concepts as proposed for the modelling of dynamically reconfigurable systems. In particular, the case study has a controller that can work at different levels of performance, and if an error happens in a component, the controller is able to adapt its control strategy to cope with the emerged situation. Thus, the section reports on a larger casestudy of modeling dynamic adaptation with Petri nets.

In the set-up of the present paper, Festo MPS is composed of two units: (i) a transportation unit called **EBR**, consisting of an elevator **E** largely ignored in this paper, two belts **B1** and **B2**, and a rotator **R**, and (ii) a processing unit composed of two drills **D1** and **D2**. Objects, available in sufficient supply, are transported, via elevator, belts, and the rotator if needed, to one of the drills for drilling. The system can operate at three levels:

(i) at high performance level **H**, where both drills are active;

(ii) at light performance level **L1**, where drill **D1** is inactive while drill **D2** continues to work;

(iii) at light performance level **L2**, where symmetrically drill **D2** is inactive while drill **D1** continues.

Initially, the system operates at high performance **H**: objects are delivered by the **EBR** to be drilled by **D1** or **D2**. If one of the drills breaks down, the system reconfigures from high performance operation to the appropriate performance level **L1** or **L2**. If both drills break down, the system cannot operate until at least one of the drills has been fixed.

We model Festo MPS as a component-based system as illustrated in Figure 9. Each component has: (i) a local behavior which is not visible to other components, (ii) a behavior that is visible to the controller, and (iii) a behavior that is set by the controller and is influencing the local behavior of the component. For instance, signal *D1_Ready* is set by **D1** and is visible to the controller; mode *D1_Perm* is set by the controller and influences the behavior of **D1**. If drill **D1** breaks down, the controller triggers a reconfiguration indirectly affecting the control of other components.

Since we are modelling dynamically reconfigurable systems, reconfiguration can happen in the middle of the working process of each component. Suppose that an object is on transportation belt **B1** on its way to drill **D1**. If **D1** breaks down, then, the rotator **R** should be activated to redirect the object to belt **B2**, so that the object is delivered at drill **D2**. Another scenario is the case where an object is being drilled by drill **D1**, just when the drill breaks down. In this case, the object should be evacuated from the drilling unit before actual reconfiguration happens.

For the modelling of each component, a number of general concepts are taken as starting point. A component has (i) a normal behavior that is executed when no error occurs and no reconfiguration is needed, (ii) adaptation behavior that is triggered between an error report and the subsequent reconfiguration, and (iii) critical behavior that should be triggered immediately, e.g. reporting that a component is in error. Overall, all functional components have a permission mode *Perm* that is set by the controller. If the *Perm* mode is active, the component

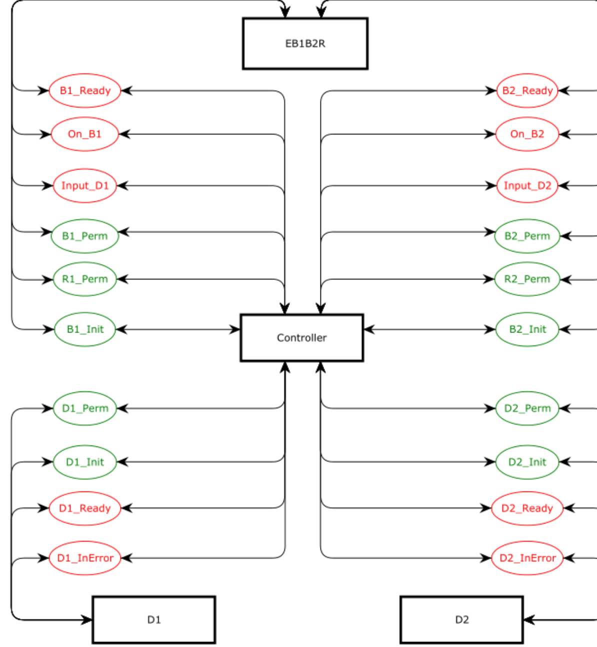


Figure 9: Festo MPS top-level model (created with CPN tools).

can run its normal working process. If the *Perm* mode is not active, then the component can only run the adaptive behavior, after which it awaits the controller for activation, i.e. to have set its *Perm* mode again.

In the Festo MPS system a component can only receive one object at a time. Thus, for instance drill **D1** can either be free or busy, holding just one object. If done with an object, drill **D1** communicates its ready status to the controller through signal *D1_Ready*. Initially, the signals *B1_Ready*, *B2_Ready*, *D1_Ready*, and *D2_Ready* of belts and drills are set and the modes B_1*_Init*, B_2*_Init*, D_1*_Init*, D_2*_Init*, and H are active for belts, drills, and controller.

We use ordinary Petri nets to model Festo MPS. For the sake of readability, we make some assumptions regarding the illustrations of the Petri nets of our models. We assume that the capacity of places is exactly 1. In some model checkers, this can be explicitly specified by setting a specific property in each place. Other model checkers, including CPN tools, do not support this concept directly. However, the semantics thereof can be achieved using the anti-places pattern, cf. [23, 24]. This approach adds a new place (called an anti-place) that corresponds to the original place, such that for each incoming/outgoing arc to/from the original place, there is an outgoing/incoming arc from/to the corresponding anti place moving the same number of tokens. This way, the

initial marking of the anti-place specifies the capacity of its original place.

Next we describe the local behavior of the system components as well as their communication with the controller. In this case study, Festo MPS has two similar drills. Therefore, we focus on the details of the transportation unit **EBR**, the first drill **D1**, and the controller in the sequel. We distinguish between two kinds of components: (i) components with potential failures that can be the cause of a reconfiguration in the system (such as **D1** and **D2**), and (ii) components with normal behavior only and that don't trigger a reconfiguration. However, all components can be influenced by a reconfiguration that is triggered in the system.

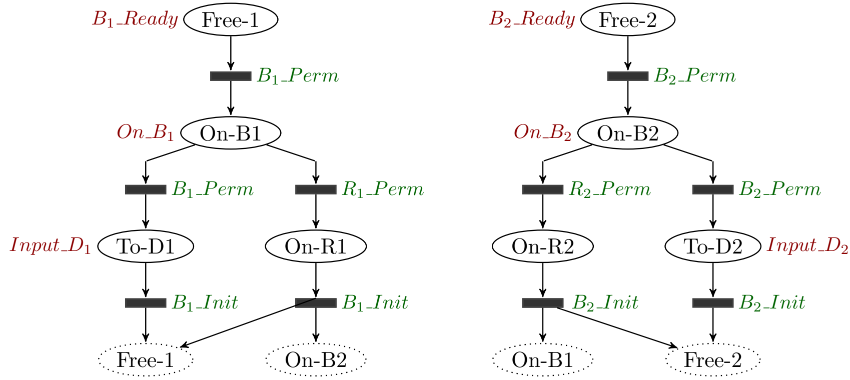


Figure 10: Component net of **EBR** unit.

The transportation unit **EBR** as modeled in Figure 10 specifies the local behavior of the transporter belts **B1** and **B2**, and the rotator **R**. Initially, both belts are free and ready for transportation of objects toward the drills with signals B_1_*Ready* and B_2_*Ready* being set. An object can therefore be transported to either belt. Once an object is on one of the belts, a signal is set (*On*_B_1 and *On*_B_2, respectively). Then, the object can either be transported to the corresponding drill or may end up via the rotator to the other belt. In the former case, a signal is set (*Input*_D_1) to indicate that an object has been given to the drill by the **EBR**, and the belt can move back to its initial state again. In the latter case, the object is transported to the second belt, and the current belt can resume from its initial state.

Drill **D1** as modeled in Figure 11, is a component with potential failures. Initially, drill **D1**, is free and has its signal D_1_*Ready* switched on. Next, it starts drilling, assuming an object is made available, and may yield two possible results: a normal exit (place `NmlExit`) or an error occurs (place `InError`). The former means that the drill did not break down when drilling. So, it returns back to its

initial state, i.e. with place `Free` marked. The latter indicates that the drill broke while processing the object. In this case, the object is evacuated and the drill sets its signal *D1_InError* to inform the controller. After that, the drill waits to be fixed before going back to its initial state.

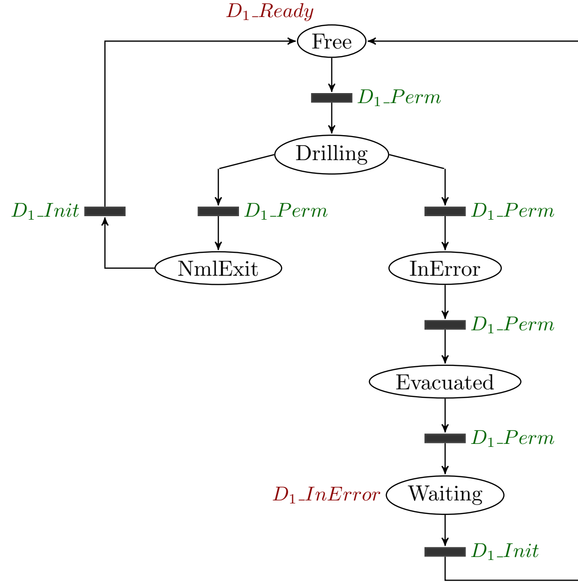


Figure 11: Component net of drill **D1**.

Without the controller, components cannot handle external failures. For instance, when drill **D1** is in error, the transportation unit **EBR** continues if not interrupted to transport objects to drill **D1**, which is inactive though. In our case study the controller has two main roles: synchronizing between components, i.e. coordination of behavior, and adapting behavior in case of failures, i.e. imposing reconfigurations. The controller receives the information regarding the state of a component via signals that are set by the components themselves. The controller is described in Figure 12. Its behavior is composed of two parts. Figure 12a shows the submodel controlling the **EBR** component, Figure 12b shows the submodel controlling the drill components. According to these signals, the controller may trigger a mode transfer for the components. Note that in Figure 12, we write for brevity decorations like $\rightarrow M$ for a transfer from any current mode to the mode M, and we do not write the controller mode if the transition can be fired in all modes. In such a case, the component adopts the new mode as imposed. As a result, the controller has exclusive access to the performance operation of the system.

When operating at performance level **H** or **L2**, the controller awaits a signal

from drill **D1** indicating that it is ready and a signal from the transportation unit **EBR** indicating that it has an input for **D1**. Drill **D1** is prepared to drill the object, and the input from the **EBR** is consumed (place `In1Free` becomes marked again). The controller sets the D_1*_Perm* mode for **D1** so that it can start drilling. At this stage, the controller can end up in one of the three following states: (i) if the system is operating at the **H** level and the D_1*_InError* signal is set, then a reconfiguration to **L1** is triggered. At the **L1** level, drill **D1** cannot operate until the controller activates its D_1*_Init* mode. (ii) If the system is operating at the **L2** level and the D_1*_InError* signal is set, both drills are in error. So, the drills are stalled until the D_1*_Init* and/or the D_2*_Init* mode are activated. (iii) If the system is operating at the **H** or **L2** level of performance and the D_1*_Ready* signal is set, then the exit is normal and the drill can resume from its initial state.

The controller also handles changes in the system operation triggered by the user. In this case study, we distinguish two kinds of reconfigurations: (i) automatic reconfiguration that is triggered when an error occurs, and (ii) reconfigurations on user request.

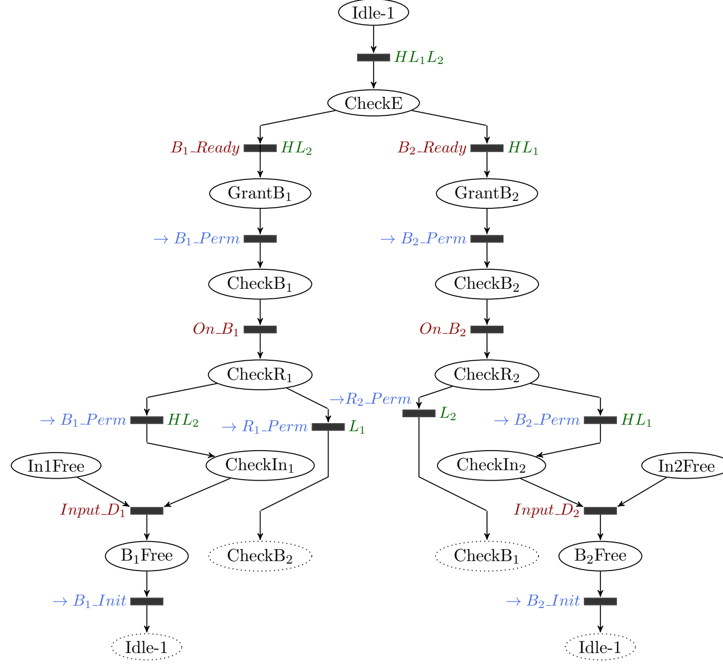
The mode transfer $H \to L_1$ for a change from a high level of operation to the light level with drill **D1** inactive, is an example of the former. As an instance of the latter, the mode transfer $L_1 \to H$ is triggered where both places `Idle_1` and `Idle_2` are marked, which means that both drills are working and available. Such a reconfiguration is coordinated by the user, the local behavior of which we didn't specify.

The complete model of our Festo MPS casestudy has been modeled with CPN tools. In particular the possibility for stepwise simulation that CPN tools provide is helpful when going through various scenarios. However, the organization instigated by Paradigm, especially regarding the interfaces between controller on the one hand and components equipped with subnets for signals and modes for sub-behavior and transfer on the other hand greatly structures the task.

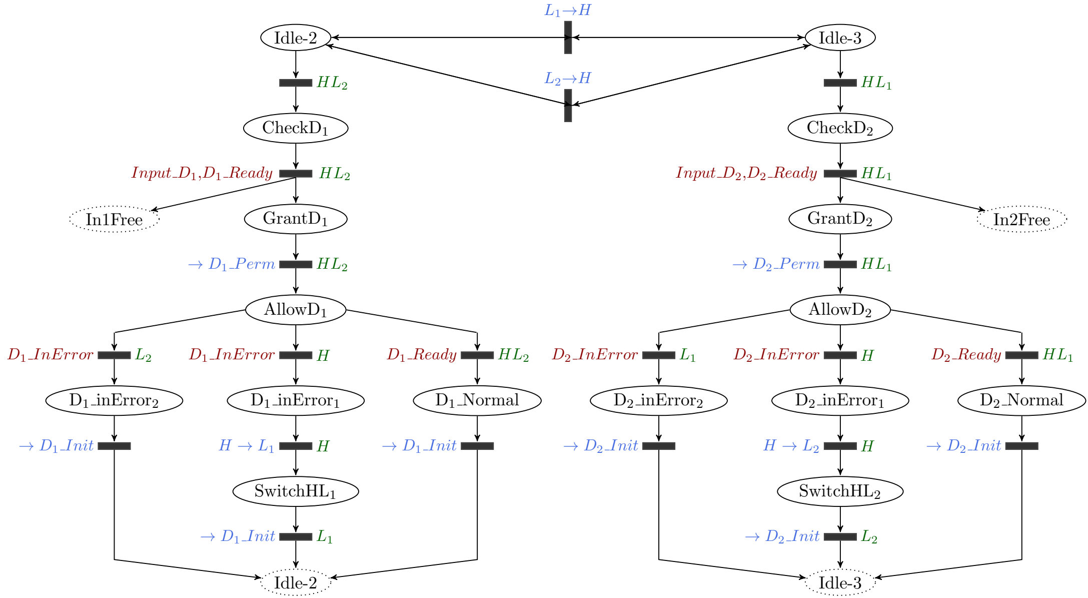
5. Formalization

The decorated Petri sub-nets used in the Paradigm approach, modelling component nets and coordinator nets, can be represented as well as standard Petri nets; the decoration are decorations only. In this section we provide a formal definition of a Paradigm model in terms of Petri nets and formalize the restrictions that fololow from the way coordination is governed within Paradigm.

1. Components execute local transitions restrictedly, in interaction with the coordinator. This interaction is regulated via modes and signals. Transitions of components belong to a mode and can only fire if the mode is active.



(a) Submodel controlling the **EBR** component.



(b) Submodel controlling drillers **D1** and **D2**.

Figure 12: Coordinator net of the controller.

2. The transitions of a coordinator may depend on the modes and signals of the components. However, per component at most one mode and one signal is input to the transition and correspondingly at most one mode and one signal is output of the transition.

A *component net* is an occurrence net that has its places and transitions decorated by so-called signals and modes. More precisely, a component net is a 7-tuple $\mathcal{N}_{comp} = (P, T, F, M, m, S, s)$ with places P, transitions T, and flow F as usual, sets M and S of elements called modes and signals, and decorations $m : T \to M$ and $s : P \to S$ of transitions and places, respectively. A *coordinator net* is an occurrence net where transitions are decorated by a number of signals and/or a number of pairs of modes, the latter referred to as mode transfers. However, the signals and modes belong to the components that the coordinator is controlling. Thus, a coordinator net is an occurrence net $\mathcal{N}_{coord} = (P, T, F, M, t, S, s)$ where the sets P, T, F, M, and S are as before, and the maps $t : T \to \mathcal{P}(M \times M)$ and $s : T \to \mathcal{P}(S)$ decorate transitions with mode transfers and signals, respectively. The definition below formalizes the notion of a Paradigm model that integrates component and coordinator nets.

Definition 1. *Let $\mathcal{N} = (P, T, F, M_0)$ be a safe Petri net and $n > 0$. The subnets $\mathcal{N}_i = (P_i, T_i, F_i)$ for $i = 1, \ldots, n$ and $\mathcal{N}' = (P', T', F')$ are called a Paradigm model with components $\mathcal{N}_1$ to $\mathcal{N}_n$ and coordinator $\mathcal{N}'$ for $\mathcal{N}$ if the following is satisfied:*

1. *(component conditions) For all $i = 1, \ldots, n$ it holds that*

   a. *$P_i \subseteq P$, $T_i \subseteq T$, and $F_i = (P_i \times T_i \cup T_i \times P_i) \cap F$ such that $P_i = oP_i \cup mP_i \cup sP_i$ for some non-empty and disjoint sets oP_i, mP_i, and sP_i containing places referred to as states or ordinary places, modes, and signals, respectively;*

   b. *for each transition $t \in T_i$, exists some $m \in mP_i$, such that ${}^{\bullet}t \cap mP_i = \{m\}$ and $t^{\bullet} \cap mP_i = \{m\}$, and t is said to be a local transition of $\mathcal{N}_i$ in mode m;*

   c. *for each transition $t \in T \backslash T_i$ at least one of the sets ${}^{\bullet}t \cap P_i$ and $t^{\bullet} \cap P_i$ is empty.*

2. *(coordinator conditions)*

   a. *$P' \subseteq P$, $T' \subseteq T$, and*

   $$F' = \big((P' \cup \textstyle\bigcup_{i=1}^{n} mP_i \cup sP_i) \times T' \cup T' \times (P' \cup \bigcup_{i=1}^{n} mP_i \cup sP_i)\big) \cap F;$$

   b. *for each transition $t \in T'$ and each $i = 1, \ldots, n$, the four sets ${}^{\bullet}t \cap mP_i$, $t^{\bullet} \cap mP_i$, ${}^{\bullet}t \cap sP_i$, and $t^{\bullet} \cap sP_i$ are all singletons or are all empty;*

c. *for each transition* $t \in T \backslash T'$ *at least one of the sets* ${}^\bullet t \cap P'$ *and* $t^\bullet \cap P'$ *is empty.*

Regarding components, the first condition of Definition 1 distinguishes the ordinary places from those for the decorations, i.e. the modes and signals of the component. The second condition reflects that a transition belongs to a mode. The unique place belonging to a mode is both input and output to the transition. As a consequence, the transition can only be taken when the place for the mode contains a token. The last condition for components forbids that a component shares a transition with other components or with the coordinator. It is allowed that another subnet needs to provide a token to the transition or will obtain a token from the transition, but the subnet is not allowed to do both. This leads to a clear separation between the component and the rest of the model.

Transitions of the coordinator involve the places of the coordinator itself, but can also input and output from the mode and signal places of the component as reflected by the first coordinator condition of the definition. If so, the second coordinator condition requires, the transition of the coordinator both to consume from a mode place and from a signal place and to produce a token to a mode place and to a signal place. Regarding the modes this is relevant for the consistency of the component; it can only be in one mode at the same time. Regarding the signals, it is requested here for convenience. More refined policies for signalling are possible as well, e.g. removing the signal and setting no signal upon a transition as components are capable to set the signals themselves. The last condition of the definition is for drawing the borders between the coordinator, components, and possibly other subnets in the model.

We illustrate Definition 1 for the case of the workers and scheduler example. The subnet for an individual worker as discussed in Section 2 is depicted in Figure 13. For its places we have the four states `nCrit`, `pre`, `crit`, and `post`, the two modes *NoPm* and *Perm*, and the three signals *request*, *none*, and *ready*. So, with reference to condition 1a of Definition 1, for worker i, for $i = 1, 2, 3$, we have a subnet $\mathcal{N}_i = (P_i, T_i, F_i)$ say, where $P_i = oP_i \cup mP_i \cup sP_i$ with respectively $oP_i = \{\texttt{nCrit}, \texttt{pre}, \texttt{crit}, \texttt{post}\}$, $mP_i = \{\mathit{NoPm}, \mathit{Perm}\}$, and $sP_i = \{\mathit{request}, \mathit{none}, \mathit{ready}\}$.

The transitions from `post` to `nCrit` (called `finish`) and from `nCrit` to `pre` (called `prepare`) can only be fired in mode *NoPm* (i.e. when the place *NoPm* has a token), while the transitions from `pre` to `crit` (called `enter`) and from `crit` to `post` (called `finish`) can only fire in mode *Perm*. Firing of the local transitions doesn't change the mode. Thus, for worker i, we have the local transitions $T_i = \{\texttt{prepare}_i, \texttt{enter}_i, \texttt{leave}_i, \texttt{finish}_i\}$. As noted, with respect to condition 1b of Definition 1, we have for the transition `prepare` that ${}^\bullet\texttt{prepare} \cap mP_i = \{\mathit{NoPm}\}$ and $\texttt{prepare}^\bullet \cap mP_i = \{\mathit{NoPm}\}$ and similarly for the other transitions.

As to signalling, for example the transition from `nCrit` to `pre`, which is only possible in mode *NoPm*, moves a token from the signal *none* to the signal *request*; the transition from state `pre` to state `crit`, only allowed in mode *Perm*, moves the token back from signal *request* to signal *none*.

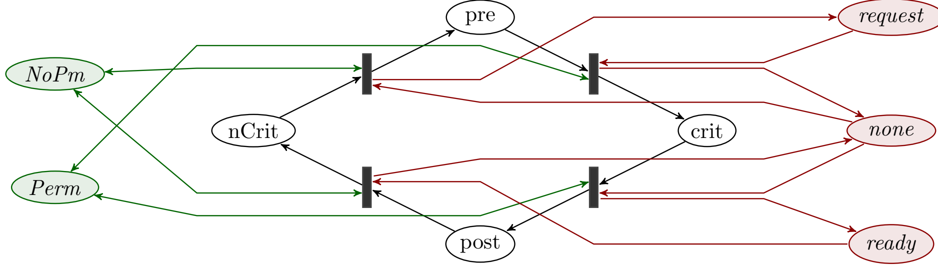


Figure 13: Component net for individual worker

The subnet for the scheduler, say $\mathcal{N}' = (P', T', F')$ is partially depicted in Figure 14. The figure focusses on the interaction of the scheduler with worker 1, surpressing the similar places and transitions for the other two workers.

The scheduler has, corresponding to the coordinator net of the non-migrating scheduler of Figure 2, places `idle` and the places $\texttt{check}_i$, $\texttt{granted}_i$, $\texttt{allowed}_i$, and $\texttt{withhold}_i$ for each of the workers, i.e. $i = 1, 2, 3$ as well as their modes NoPm_i and Perm_i, and the signals $\mathit{request}_i$, none_i, and ready_i, thus 28 places in total, 15 places shared with the workers. In the set-up of the workers and scheduler example, transitions involve places exclusively belonging the scheduler and modes of workers, or places of the scheduler and signals of the worker. This is done for clarity. For the example, the transitions from `check1` to `granted1` and from `granded1` to `allowed1` can be identified, as do the transitions from `allowed1` to `withold1` and from `withhold1` to `idle`. However, condition 2b of Definition 1 is met (also when identifying the transitions mentioned). The transitions of the subnet $\mathcal{N}'$ of the scheduler respects the restriction of the pre-set's and post-set's intersection with the sets of modes and the sets of signals to consistenty be either one or no place.

The Petri net capturing the workers and scheduler example consists of the three subnets for the three workers and the subnet for scheduler only. Its initial marking assigns to the subnet of each of the workers a token to the places `nCrit`, *NoPm*, and *none*, reflecting that each worker is supposed to start with non-critical activity without a need nor permission to enter the critical section. The subnet of the scheduler has a token at the place `idle` (as well as the tokens in the interface obtained from the initial marking of the subnets of the workers, i.e. three times the modes *NoPm* and the signals *none*). The local transitions of the workers don't involve places of other subnets. Also, the transitions of the scheduler involves places from these four subnets. Conditions 1c and 2c of

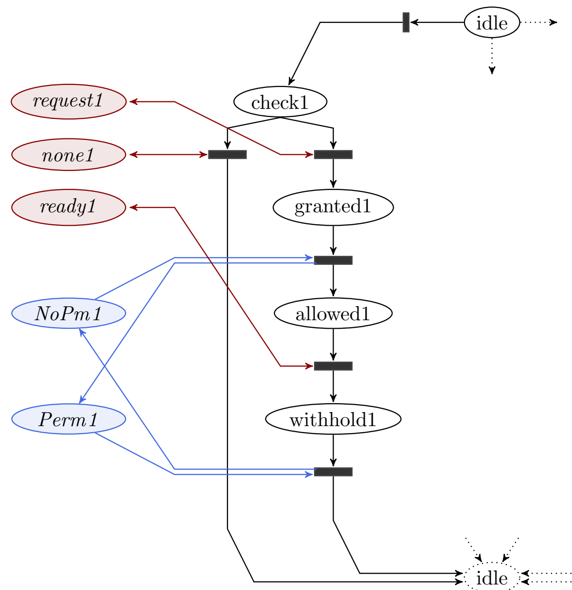


Figure 14: Petri net of the scheduler (parts for worker 2 and worker 3 omitted).

Definition 1 also applies when transitions involve places of other subnets other than the transitions typical for a coordinator net. An example of such a situation arises for the dining philosopers, although we have kept this out of the discussion. The ability of philosopher to grep their right fork involves a transition changing the availability of the fork as left fork of one of the other philosophers. To encode the exclusivity, a fork can be represented by a single place, marked in the initial marking to represent that the fork is on the table. A transition that picks the fork up consumes the token; a transition that puts the fork down returns the token.

The subnet for the migrating scheduler is (partially) depicted in Figure 15. Note the modes *NoPm1* and *Perm1* of worker 1 on the left and the modes *asIs* and *toBe* of the migrating scheduler itself on the right. Taking the places of workers 2 and 3 involved in the subnet into account, the migrating scheduler has a total of 33 places, viz. the state `idle` and 9 places for each of the three workers as before plus the new states `idle1`, `idle2`, and `idle3` together with the two newly created modes *asIs* and *toBe*.

The formalization of a Paradigm model for a Petri net as given by Definition 1 provides conditions that regulate the interaction of the subsets that are be interpreted as components and the subnet that is seen as the coordinator. The definition strives for a clean interface among these parts: Components are in one mode at a time and restrict their behaviour to the local transitions belong to that

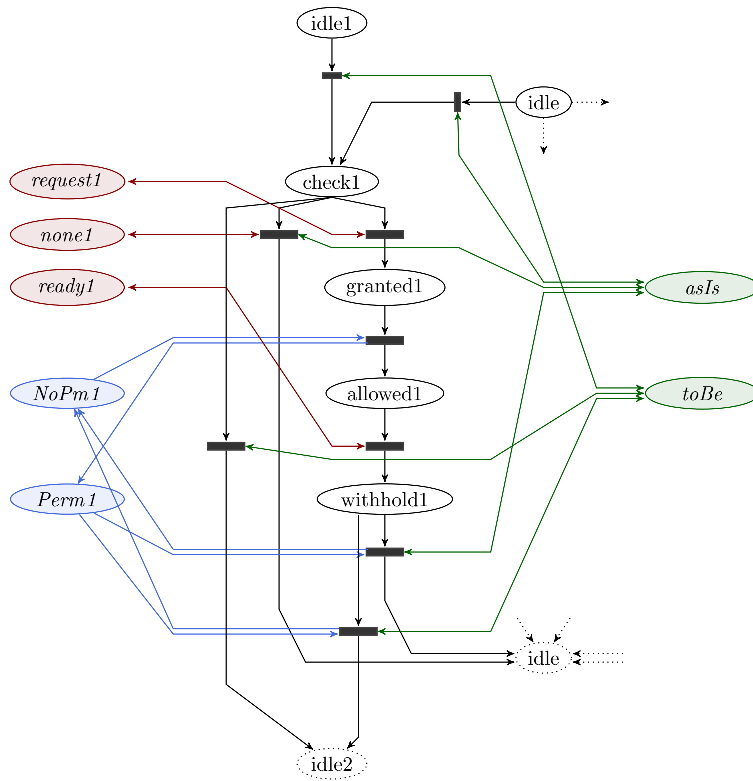


Figure 15: Petri net of the migrating scheduler (parts for worker 2 and worker 3 omitted): signals *request1*, *none1*, *ready1* and modes *NoPm* and *Perm* of worker 1; modes *asIs* and *toBe* of the scheduler. Bi-directional arcs of transitions permitted in both *asIs* and *toBe* mode to the respective mode places not drawn.

mode. Meanwhile components can set signals as one-side asynchronous communication with the coordinator. In situations specific to the application, when one or mode components are in certains modes with certain signal set, the coordinator may take a transition and thereby update the modes of the components involved.

It is noted, that the definition doesn't require the Petri net under consideration to be comprised of the subsets of the components and the coordinator. Although tacitly, in the modelling of the dining philosophers in Section 3 the forks where not involved directly in the discussion of the coordination and migration thereof. On another note, one can have multiple Paradigm models of the same Petri net, each of them highlighting different coordination aspects of the overall net (as we have seen for dynamic adaptation). This way various separate views on the coordination in the net are created. The benefit of this is that such a decomposition provides a better focus on the coordination within a model and helps to identifying possible underdesired interaction between the models.

6. Concluding remarks

In this work we propose to exploit the concepts of the coordination modeling language Paradigm in the setting of Petri nets in order to model the coordination and reconfiguration in component-based systems. Using three examples, a workers and scheduler example, a variant of the dining philosophers, and the Festo MPS casestudy, we illustrate how the design of component-based systems in general, but more importantly especially that of reconfigurable systems, becomes more straightforward when separating levels of coordination.

Starting point of our modeling is the decomposition of the whole system into component nets representing the actual behavior glued together by a coordinator net that handles the cooperativity of the system. This is achieved by a coarse-grained allowing and blocking of the modes of components by the coordinator based on the signals the components provide. Iterating the abstraction, it is observed that such a coarse-grained view on behavior can help to describe dynamic reconfiguration.

In future work, we aim to continue our current investigation of coordination modeling with CPN Tools and to make the presented approach amendable for formal verification techniques, in particular using the mCRL2 toolset. The formalization of our description of coordinator and components with Petri nets can be further refined to support asynchronous coordination (a change of mode for a component not being in effect immediate after its change by the coordinator to model delay of such communication), in line with Pardigm for labelled transition systems. We expect, the notion of a so-called trap in Paradigm can be arranged for by having a default signal for a component (like signal *none* in the workers

and scheduler example) and that can only once be changed in a mode to a non-default signal (the signals *request* and *ready* for the example mentioned). When a mode transfer occurs, the signal is reset to the default value again.

Acknowledgements We are indebted to the reviewers of the manuscript for their constructive suggestions to improve the paper. We thank Luuk Groenewegen for his comment on an early draft of the paper and Hossem Eddine Hafidi for his help in drawing a large part of the figures of this paper.

The idea of modeling reconfiguration within Paradigm, in the context of labeled transition systems with a special pre-formatted reconfigurator called McPal, has its origin in the paper [25] presented at the conference on Formal Aspects of Component Software held in Guimarães, co-chaired by Luís Soares Barbosa. The second author of this paper acknowledges the pleasant interaction and cooperation with Luís over the years.

References

[1] C. Petri, Kommunikation mit Automaten, Ph.D. thesis, Technische Hochschule Darmstadt (1962).

[2] W. Reisig, Petri Nets: An Introduction, Vol. 4 of EATCS Monographs on Theoretical Computer Science, Springer, 1985.

[3] D. Buchs, J. Carmona (Eds.), Application and Theory of Petri Nets and Concurrency, 42nd International Conference, LNCS 12734, 2021.

[4] R. Valk, Self-modifying nets, a natural extension of Petri nets, in: G. Ausiello, C. Böhm (Eds.), Proc. ICALP'78, LNCS 62, 1978, pp. 464–476.

[5] E. Badouel, J. Oliver, Reconfigurable nets, a class of high level Petri nets supporting dynamic changes within workflow systems, Tech. Rep. RR–3339, INRIA (1998).

[6] J. Zhang, M. Khalgui, Z. Li, O. Mosbahi, A. Al-Ahmari, R-TNCES: A novel formalism for reconfigurable discrete event control systems, IEEE Transactions on Systems, Man, and Cybernetics: Systems 43 (4) (2013) 757–772.

[7] A. Ratzer, L. Wells, H. Lassen, M. Laursen, J. Qvortrup, M. Stissing, M. Westergaard, S. Christensen, K. Jensen, CPN tools for editing, simulating, and analysing coloured Petri nets, in: W. van der Aalst, E. Best (Eds.), Proc. ICATPN 2003, Vol. 2679 of Lecture Notes in Computer Science, Springer, 2003, pp. 450–462.

[8] M. Heiner, M. Herajy, F. Liu, C. Rohr, M. Schwarick, Snoopy: a unifying Petri net tool, in: S. Hadded, L. Pomello (Eds.), Proc. Petri Nets 2012, Vol. 7347 of Lecture Notes in Computer Science, Springer, 2012, pp. 398–407.

[9] R. Davidrajuh, B. Skolud, D. Krenczyk, Performance evaluation of discrete event systems with GPenSIM, Computers 7 (1) (2018) 8.

[10] L. Groenewegen, N. van Kampenhout, E. de Vink, Delegation modeling with Paradigm, in: J.-M. Jacquet, G. Picco (Eds.), Proc. Coordination 2005, LNCS 3454, 2005, pp. 94–108.

[11] L. Groenewegen, E. de Vink, Evolution-on-the-fly with Paradigm, in: P. Ciancarini, H. Wiklicky (Eds.), Proc. Coordination 2006, LNCS 4038, 2006, pp. 97–112.

[12] A. Stam, Interaction protocols in Paradigm, Ph.D. thesis, Leiden University (2009).

[13] S. Andova, L. Groenewegen, E. de Vink, Dynamic adaptation with distributed control in Paradigm, Science of Computer Programming 94 (2014) 333–361.

[14] L. Groenewegen, J. Verschuren, E. de Vink, Extending Paradigm with data, in: F. de Boer, M. Bonsangue, J. Rutten (Eds.), It's All About Coordination, LNCS 10865, 2018, pp. 224–244.

[15] L. Groenewegen, E. de Vink, Operational semantics for coordination in Paradigm, in: F. Arbab, C. Talcott (Eds.), Proc. Coordination 2002, LNCS 2315, 2002, pp. 191–206.

[16] D. Clarke, Coordination: Reo, nets, and logic, in: F. d. Boer, M. Bonsangue, S. Graf, W. de Roever (Eds.), Proc. FMCO 2007, LNCS 5382, 2007, pp. 226–256.

[17] C. Krause, Integrated structure and semantics for Reo connectors and Petri nets, in: F. Bonchi, D. Grohmann, P. Spoletini, E. Tuosto (Eds.), Proc. ICE 2009, Vol. 12 of EPTCS, 2009, pp. 57–69.

[18] E. Ardeshir-Larijani, A. Farhadi, F. Arbab, Simulation of hybrid Reo connectors, in: Proc. RTEST 2020, IEEE, Tehran, 2020, p. 10pp.

[19] J. Padberg, L. Kahloul, Overview of reconfigurable Petri nets, in: R. Heckel, G. Täntzer (Eds.), Graph Transformation, Specifications, and Nets, LNCS 10800, 2018, pp. 201–222.

[20] Y. Hafidi, L. Kahloul, M. Khalgui, Z. Li, K. Alnowibet, T. Qu, On methodology for the verification of reconfigurable timed net condition/event systems, IEEE Transactions on Systems, Man, and Cybernetics: Systems 50 (10) (2018) 3577–3591.

[21] O. Bunte, J. Groote, J. Keiren, M. Laveaux, T. Neele, E. de Vink, A. Wijs, J. Wesselink, T. Willemse, The mCRL2 toolset for analysing concurrent systems, in: T. Vojnar, L. Zhang (Eds.), Proc. TACAS 2019, LNCS 11428, 2019, pp. 21–39.

[22] R. Roman, R. Holubek, M. Janíček, K. Velíšek, O. Tirian, Analysis of the industry 4.0 key elements and technologies implementation in the FESTO didactic educational systems mps 203 i4.0, Journal of Physics: Conference Series 1781 (2021) 012030.

[23] Z. Su, M. Qiu, Airport surface modelling and simulation based on timed coloured Petri nets, Promet-Traffic and Transportation 31 (5) (2019) 479–490.

[24] N. Vizovitin, V. Nepomniaschy, A. Stenenko, Application of colored Petri nets for verification of scenario control structures in UCM notation, Automatic Control and Computer Sciences 51 (7) (2017) 489–497.

[25] S. Andova, L. Groenewegen, E. de Vink, Distributed adaptation of dining philosophers, in: L. Barbosa, M. Lumpe (Eds.), Proc. FACS 2010, Guimarães, LNCS 6921, 2012, pp. 125–144.