

# Bachelor Project: Paradigm

Jasper Stafleu, Universiteit Leiden

September 13, 2006

## 1 Introduction

It has recently been shown that the coordination language Paradigm (see [1]) can describe a dynamically consistent way to let a Paradigm model evolve on-the-fly using self-adaptation, see [2]. This bachelor project will change the examples in [2] to clarify some alterations made in the `McPal` component, which is the actual driving force behind the migration. We'll also give an alternative for the negative side rule  $P(\pi) : S \xrightarrow{\theta}$ .

Section 2 presents the new `McPal` component, which has been changed into a clearer version compared to the original version in [2]. Section 3 repeats the original migrations, changing the models to allow for the new `McPal` component. Besides this revised migration, the migration model is adapted in Section 4 to allow the two migrations from Section 3 to be done in any order. Using this extension, we will reveal some undesired effects of the negative side rule, for which we will present two solutions. The first solution is done by remodeling the negative side rule, explained in Section 5. The second solution changes the standard `McPal` given in Section 2 to allow for migrations in which the new rules are added gradually instead of altogether in one step.

## 2 The new McPal model

In this section we will present the alterations in the `McPal` component and the reasons for altering them. The new `McPal` process is visualized in Figure 2.1.

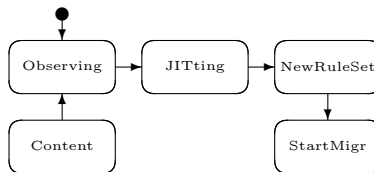


Figure 2.1: Process `McPal`

In this model, the initial state, **Observing**, will be left at some moment, after which the migration rules are passed on to **McPal** in state **JITting**, filling the sets  $Cr_{smig}$  —the set of rules that apply only during the migration— and  $Cr_{stoBe}$  —the set of rules that should apply after the migration is done— to values chosen by the modeler. The corresponding consistency rules responsible for these steps are:

$$\begin{aligned}
 McPal : \quad & \textit{Observing} \longrightarrow \textit{JITting} \\
 McPal : \quad & \textit{JITting} \longrightarrow \textit{NewRuleSet} \\
 * \quad & McPal[CRS := CRS \cup Cr_{smig} \cup Cr_{stoBe}]
 \end{aligned}$$

Another addition to the generic **McPal** component is the partition **EvolMcPal**, see Figure 2.2, which formalizes **McPal**'s global behavior. This is useful to make an easy-to-find distinction between the model being migrated and the model being stable. Since this version has no JIT-extended migration rules yet, the

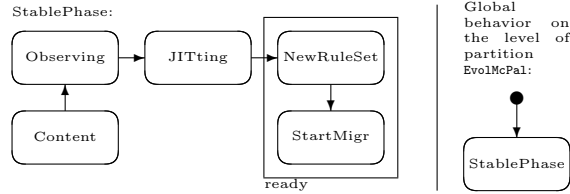


Figure 2.2: **McPal**'s partition **EvolMcPal**

global behavior of **EvolMcPal** is trivial, containing only the **StablePhase**. As soon as state **JITting** is left though,  $Cr_{smig}$  and  $Cr_{stoBe}$  are added. In particular,  $Cr_{smig}$  should always contain the two rules which state the transitions from **NewRuleSet** to **StartMigr** and from **Content** to **Observing**. The first of these rules sets **EvolMcPal** to an unstable phase, allowing for the migration to take place:

$$\begin{aligned}
 McPal : \quad & \textit{NewRuleSet} \longrightarrow \textit{StartMigr} \\
 * \quad & McPal(\textit{EvolMcPal}) : \textit{StablePhase} \xrightarrow{\textit{ready}} \textit{MigrPhase}
 \end{aligned}$$

It is important to note the unstable phase has been named **MigrPhase**, but, as we will see later, the actual name is different for each migration (and could even exist out of multiple phases), being decided upon during the JIT-extension. The second rule stabilizes the model, setting  $CRS$  to  $Cr_{stoBe}$ , thus removing the migration and old rules. In this rule, the trap in **MigrPhase** is named **migrDone**, but this will actually be named by the modeler as well:

$$\begin{aligned}
 McPal : \quad & \textit{Content} \longrightarrow \textit{Observing} \\
 * \quad & McPal(\textit{EvolMcPal}) : \textit{MigrPhase} \xrightarrow{\textit{migrDone}} \textit{StablePhase}, \\
 & McPal[CRS := Cr_{stoBe}]
 \end{aligned}$$

A few possible global behaviors of `McPal` (`EvoMcPal`) will be presented in Section 3 and 4. Section 3 will cover the alterations to the migrations from [2] and shows `McPal` now has complete control over the migration, as opposed to the former models in which merely adding the  $Crsmig$  and  $CrstoBe$  sets to  $CRS$  started the migration. The additional requirement of `McPal` (`EvoMcPal`) having left the `StablePhase` will ensure the complete control of the `McPal` component over the migration.

### 3 Updating the migrations

In this Section we will change the migrations given in [2] to ensure their consistency with the revised `McPal`. For the sake of completeness, the visual representations of `Workeri` and `Scheduler` are repeated in Figure 3.1.

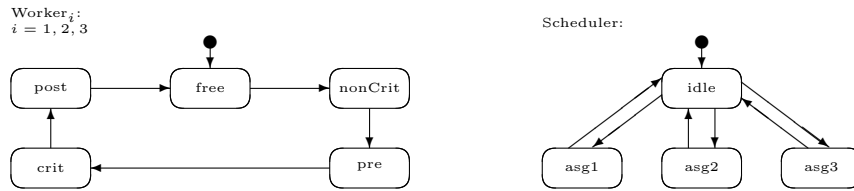


Figure 3.1: The processes `Workeri` and `Scheduler`

This model covers a version of the critical section problem; the three workers want access to the critical section `crit` when they reach the state `nonCrit` and have made their request clear in the `pre` state. If these steps have been taken they can enter the critical section. The permission to enter the critical section is then returned in state `post` and finally they restabilize themselves having left the critical section when they reach state `free`.

The `Scheduler` is the manager of this process, deciding which worker can enter the critical section in the `idle` state and then moving to the appropriate `asgi` state, giving `Workeri` permission to enter the critical section. The partition CSM of process `Workeri` and the global process `Workeri` (CSM), are visualized in Figure 3.2.

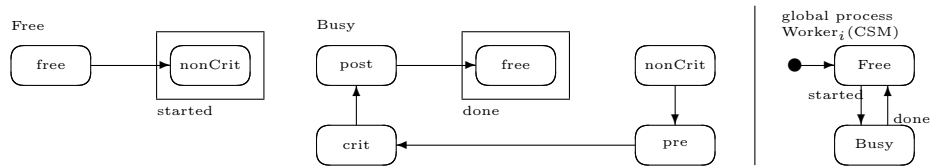


Figure 3.2: The partition CSM and global `worker` process at the level of CSM

Partition  $Worker_i(CSM)$  shows the worker enters the trap **started**, thus being able to go to the **Busy** subprocess, which means getting the permission to enter the critical section. From the trap **Done**, which contains only state **free**, the worker can return to the **Free** subprocess, losing the permission in the process. This behavior should be the basis for each of the migrations given in this Section. Adding to these the new **McPal** rules yields the following consistency rules **CRS**:

$$\begin{array}{l}
 \text{CRS :} \\
 Worker_i : free \longrightarrow nonCrit \\
 Worker_i : nonCrit \longrightarrow pre \\
 Worker_i : pre \longrightarrow crit \\
 \\
 Scheduler : \quad \quad \quad idle \longrightarrow asg_i \\
 * Worker_i(CSM) : \quad \quad Free \xrightarrow{started} Busy \\
 Scheduler : \quad \quad \quad asg_i \longrightarrow idle \\
 * Worker_i(CSM) : \quad \quad Busy \xrightarrow{done} Free \\
 \\
 McPal : \quad \quad \quad Observing \longrightarrow JITting \\
 McPal : \quad \quad \quad JITting \longrightarrow NewRuleSet \\
 * \quad \quad \quad McPal[CRS \quad := \quad CRS \cup Crs_{mig} \cup Crs_{toBe}]
 \end{array}$$

The first migration will be to change the global behavior of the worker. We wish to change it so the worker won't have to wait for the scheduler to remove its permission before starting to work on the non-critical part of his activities. To facilitate this, the worker should be able to reach the state **nonCrit** from **free** in both subprocesses. This change and the global migration of  $Worker_i(CSM)$  are visualised in Figure 3.3.

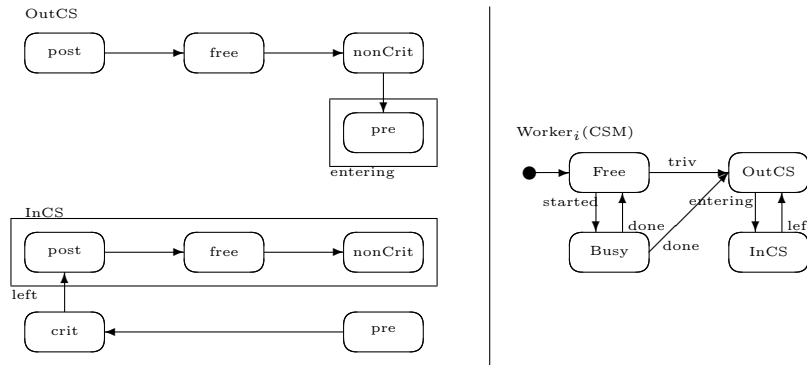


Figure 3.3: Extension of partition CSM and global process  $Worker_i(CSM)$  during the migration

The trap **started** will be changed to the trap **entering**, which contains state **pre**, and the trap **done** will be changed to the trap **left**, which contains the states **post**, **free** and **nonCrit**. This behavior satisfies the criteria and leads to the **firstWorkerToBe** input to the  $\mathbf{Crs}_{toBe}$  set:

*firstWorkerToBe* :

$Worker_i : free \longrightarrow nonCrit$	$Worker_i : crit \longrightarrow post$
$Worker_i : nonCrit \longrightarrow pre$	$Worker_i : post \longrightarrow free$
$Worker_i : pre \longrightarrow crit$	

<i>Scheduler</i> :	<i>idle</i>	$\longrightarrow$	<i>asg<sub>i</sub></i>
* $Worker_i(CSM)$ :	<i>OutCS</i>	$\xrightarrow{entering}$	<i>InCS</i>
<i>Scheduler</i> :	<i>asg<sub>i</sub></i>	$\longrightarrow$	<i>idle</i>
* $Worker_i(CSM)$ :	<i>InCS</i>	$\xrightarrow{left}$	<i>OutCS</i>

<i>McPal</i> :	<i>Observing</i>	$\longrightarrow$	<i>JITting</i>
<i>McPal</i> :	<i>JITting</i>	$\longrightarrow$	<i>NewRuleSet</i>
*	$McPal[CRS$	$:=$	$CRS \cup Crs_{mig} \cup Crs_{toBe}]$

A set of rules which can be entered into  $\mathbf{Crs}_{mig}$  to reach this form is given in **firstWorkerMigr**. In a few of these rules (and in future rules) we will use the trap **triv**; this trap always contains all states of the subprocess and thus allows the transition to take place at any moment while in that subprocess.

*firstWorkerMigr* :

<i>McPal</i> :	<i>NewRuleSet</i>	$\longrightarrow$	<i>StartMigr</i>
* $McPal(EvolMcPal)$ :	<i>StablePhase</i>	$\xrightarrow{ready}$	<i>MigrWorker</i>
<i>McPal</i> :	<i>StartMigr</i>	$\longrightarrow$	<i>Content</i>
* $Worker_1(CSM)$ :	<i>OutCS</i>	$\xrightarrow{triv}$	<i>OutCS</i>
<i>McPal</i> :	<i>Content</i>	$\longrightarrow$	<i>Observing</i>
* $McPal(EvolMcPal)$ :	<i>MigrWorker</i>	$\xrightarrow{migrDone}$	<i>StablePhase,</i>
	$McPal[CRS$	$:=$	$Cr_{stoBe}]$

<i>Scheduler</i> :	<i>asg<sub>i</sub></i>	$\longrightarrow$	<i>idle</i>
* $Worker_i(CSM)$ :	<i>Busy</i>	$\xrightarrow{done}$	<i>OutCS,</i>
$Worker_{i-1}(CSM)$ :	<i>Free</i>	$\xrightarrow{triv}$	<i>OutCS,</i>
$Worker_{i+1}(CSM)$ :	<i>Free</i>	$\xrightarrow{triv}$	<i>OutCS,</i>
$McPal(EvolMcPal)$ :	<i>MigrWorker</i>	$\xrightarrow{triv}$	<i>MigrWorker</i>

The only changes besides the new **McPal** rules in this set compared to the rules in [2] is in the last rule. The addition to this rule is the requirement for the **McPal(EvolMcPal)** partition to be in the subprocess **MigrWorker**. The requirement of being in the **MigrWorker** subprocess ensures the migration to

the new model doesn't take place until `McPal` has left the `StablePhase`. This is not the only way of doing this, we could replace the last rule by

$$\begin{array}{lcl}
 \text{McPal} : & \text{StartMigr} & \longrightarrow \text{Content} \\
 * \text{Worker}_1(\text{CSM}) : & \text{Free} & \xrightarrow{\text{triv}} \text{OutCS}, \\
 \text{Worker}_2(\text{CSM}) : & \text{Free} & \xrightarrow{\text{triv}} \text{OutCS}, \\
 \text{Worker}_3(\text{CSM}) : & \text{Free} & \xrightarrow{\text{triv}} \text{OutCS}
 \end{array}$$

This would also enable the migration, since the requirement of the scheduler being in state `idle` would be met by all workers being in state `Free` on the level of `(CSM)`. Then from `MigratedWorker`, `McPal` could continue immediately to state `Content` and finish the migration in the usual manner; we could force the migration to occur by not allowing the `Workeri(CSM)` to enter the state `Busy`, as done in [2]. It is even possible to migrate the workers in order by adding two states in `McPal`, using each step ( $\text{StartMigr} \rightarrow \text{First}$ ,  $\text{First} \rightarrow \text{Second}$  and  $\text{Second} \rightarrow \text{Content}$ ) to migrate a worker. Another possibility in which the workers and the scheduler are migrated simultaneously is presented in Section 4. `FirstWorkerMigr` leads to an `McPal(EvolMcPal)` partition with the `StablePhase` as in Figure 2.2 and the new `McPal` and `MigrWorker` visualized in Figure 3.4.

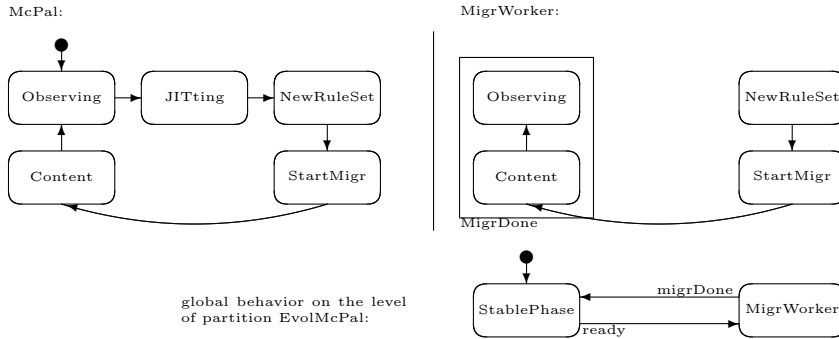


Figure 3.4: `McPal` and `McPal(EvolMcPal)` partition during the `Workeri(CSM)` migration

The second migration given in [2] changes the `Scheduler` from the current non-deterministic approach to a deterministic round-robin strategy. As done in [2] we will detail this migration assuming the `CRS` given on Page 4, that is, the non-migrated `CRS`. The new `Scheduler` is visualized in Figure 3.5.

In this model, the `Scheduler` checks whether `Workeri` wants permission in state `checki`. If so, it assigns permission to enter the critical section to `Workeri` by entering state `asgi`. If not, the model moves on to state `checki+1` and continues the process. The set of rules (using the `CRS` as given earlier in this chapter) to set `CrstoBe` to have been captured in `firstSchedToBe`:

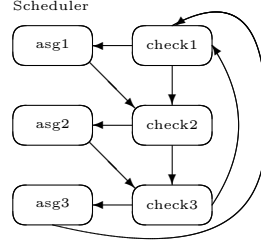


Figure 3.5: The new deterministic Scheduler

*firstSchedToBe* :

$Worker_i : free \longrightarrow nonCrit$   
 $Worker_i : nonCrit \longrightarrow pre$   
 $Worker_i : pre \longrightarrow crit$

$Worker_i : crit \longrightarrow post$   
 $Worker_i : post \longrightarrow free$

*Scheduler* :

$check_i \longrightarrow asg_i$   
 $* Worker_i(CSM) : Free \xrightarrow{started} Busy$   
 $Scheduler : asg_i \longrightarrow check_{i+1}$   
 $* Worker_i(CSM) : Busy \xrightarrow{done} Free$   
 $Scheduler : check_i \longrightarrow check_{i+1}$   
 $* Worker_i(CSM) : Free \xrightarrow{started}$

*McPal* :

$Observing \longrightarrow JITting$

*McPal* :

$JITting \longrightarrow NewRuleSet$

\*

$McPal[CRS := CRS \cup Crs_{mig} \cup Crs_{toBe}]$

Now we wish to obtain a rule set for the migration which leads to this model. Reasoning from the objective, we gain the trap **finished**, which shows the migration is done. This trap contains all the states and transitions from  $Crs_{toBe}$ . Since the  $asg_i$  states are also in  $Crs_{toBe}$ , no migration needs to be modeled to leave those states. **Idle** is not in  $Crs_{toBe}$  though, so a rule to leave that state and enter the trap needs to be added. We also need to discard the rules which leave this trap (per trap definition, see [2]). This migration is visualized in Figure 3.6, **McPal** and the partition **McPal (EvolMcPal)** during this migration are shown in Figure 3.7 during the **Scheduler** migration.

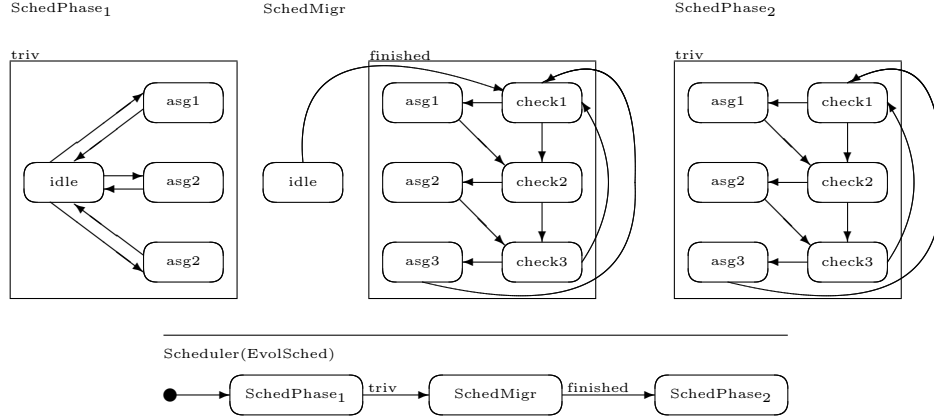


Figure 3.6: Partition EvolSched of Scheduler

The migration can be reached by `firstSchedMigr`:

*firstSchedMig* :

<i>Scheduler</i> :	<i>idle</i>	$\longrightarrow$	<i>check<sub>1</sub></i>
<i>McPal</i> :	<i>NewRuleSet</i>	$\longrightarrow$	<i>StartMigr</i>
* <i>McPal(EvolMcPal)</i> :	<i>StablePhase</i>	$\xrightarrow{\text{ready}}$	<i>MigrSched</i>
<i>McPal</i> :	<i>StartMigr</i>	$\longrightarrow$	<i>Sched1to2</i>
* <i>Scheduler(EvolSched)</i> :	<i>SchedPhase<sub>1</sub></i>	$\xrightarrow{\text{triv}}$	<i>SchedMigr</i>
<i>McPal</i> :	<i>Sched1to2</i>	$\longrightarrow$	<i>Content</i>
* <i>Scheduler(EvolSched)</i> :	<i>SchedMigr</i>	$\xrightarrow{\text{finished}}$	<i>SchedPhase<sub>2</sub></i>
<i>McPal</i> :	<i>Content</i>	$\longrightarrow$	<i>Observing</i>
* <i>McPal(EvolMcPal)</i> :	<i>MigrSched</i>	$\xrightarrow{\text{migrDone}}$	<i>StablePhase,</i>
	<i>McPal[CRS</i>	$:=$	<i>CrstoBe]</i>

In Section 4 we'll put the two migrations after each other, first in order and finally in no specific order. To reach the latter, we'll change the worker migration slightly.

## 4 Combining the Migrations

In this section we are going to combine the two migrations given in Section 3 into one migration set. First of all, we will begin our migrations from the starting `CRS` on Page 4, and migrate to `firstWorkerToBe` or `firstSchedToBe` as given in Section 3, depending on the order of migrations we choose. After this, we will continue with a second migration, which we will give in this Section. Independent of the choice, Figure 4.1 visualizes `McPal` and its global behavior on the level of `EvolMcPal` during the complete migration. The `MigrWorker` and `MigrSched` subprocesses are the same as given earlier in Figure 3.4 and Figure 3.7.

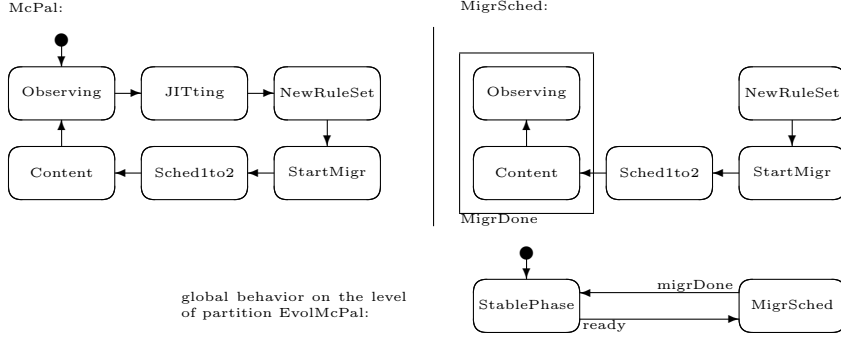


Figure 3.7: McPal and McPal(EvolMcPal) partition during the  $Worker_i$  (CSM) migration

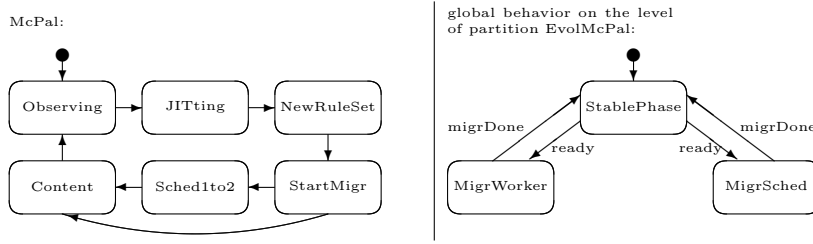


Figure 4.1: McPal and its global behavior during both migrations in any order

While the first migrations can be done as in Section 3, we can't execute the subsequent migrations as given without some changes. To determine what needs to be altered, we first construct the final desired result, given in the set  $bothToBe$ .

$$\begin{aligned}
 &bothToBe : \\
 &Worker_i : free \longrightarrow nonCrit && Worker_i : crit \longrightarrow post \\
 &Worker_i : nonCrit \longrightarrow pre && Worker_i : post \longrightarrow free \\
 &Worker_i : pre \longrightarrow crit \\
 \\ 
 &Scheduler : && check_i \longrightarrow asg_i \\
 &* Worker_i(CSM) : OutCS \xrightarrow{entering} InCS \\
 &Scheduler : && asg_i \longrightarrow check_{i+1} \\
 &* Worker_i(CSM) : InCS \xrightarrow{left} OutCS \\
 &Scheduler : && check_i \longrightarrow check_{i+1} \\
 &* Worker_i(CSM) : OutCS \xrightarrow{entering}
 \end{aligned}$$

$$\begin{array}{lcl}
McPal : & Observing & \longrightarrow JITting \\
McPal : & JITting & \longrightarrow NewRuleSet \\
* & McPal[CRS & := CRS \cup Crs_{mig} \cup Crs_{toBe}]
\end{array}$$

This set differs from the sets `firstWorkerToBe` and `firstSchedToBe` given in Section 3 only by the rules which contain references to the partitions that alter, such as the `Workeri` (CSM) partition, or the detailed behavior that changes, such as `Scheduler`.

To alter the migration sets `firstWorkerMigr` and `firstSchedMigr` we again need to change the rules which contain parts of the changing components. When we first execute the `Worker` migration, we need to construct a set `nextSchedMigr` which successfully migrates the `Scheduler` after the `Worker` has migrated. We can do this by changing the rules from `firstSchedMigr` which concern the `Workeri` (CSM) partition, which changed during the first migration. None of those exist in the `firstSchedMigr` set, so `nextSchedMigr = firstSchedMigr`. When we reverse the order, we need to construct a `nextWorkerMigr` set from `firstWorkerMigr` in which all the detailed behavior of `Scheduler` is altered. This time, we do need to change the rule

$$\begin{array}{lcl}
Scheduler : & asg_i & \longrightarrow idle \\
* Worker_i(CSM) : & Busy & \xrightarrow{done} OutCS, \\
Worker_{i-1}(CSM) : & Free & \xrightarrow{triv} OutCS, \\
Worker_{i+1}(CSM) : & Free & \xrightarrow{triv} OutCS, \\
McPal(EvolMcPal) : MigrWorker & \xrightarrow{triv} & MigrWorker
\end{array}$$

because it will not fire, since the transition  $asg_i \rightarrow idle$  is not possible; the first migration changed this to  $asg_i \rightarrow check_{i+1}$ . This suggests to substitute this rule with

$$\begin{array}{lcl}
Scheduler : & asg_i & \longrightarrow check_{i+1} \\
* Worker_i(CSM) : & Busy & \xrightarrow{done} OutCS, \\
Worker_{i-1}(CSM) : & Free & \xrightarrow{triv} OutCS, \\
Worker_{i+1}(CSM) : & Free & \xrightarrow{triv} OutCS, \\
McPal(EvolMcPal) : MigrWorker & \xrightarrow{triv} & MigrWorker
\end{array}$$

which is indeed enough to make a valid set `nextWorkerMigr` which migrates the `Worker` after the `Scheduler` has already migrated. One problem does arise when using this version; during the migration, `CRS` contains two negative side rules on the same partition, both allowing the transition  $check_i \rightarrow check_{i+1}$ . Since a negative side rule is satisfied if the model is not in the relevant subprocess (or not in the relevant trap), one of these rules can always fire; the model can never be in `OutCS` and `Busy` at the same time. This means the `Scheduler` can continue to cycle through the `checki` states without ever giving any worker the permission

to enter the critical section and, consequently, the migration might never take place. We will construct two solutions to this problem in Section 5 and 6. For now, we'll use the assumption that the migration takes place eventually.

Finally, we will now construct another set **bothMigr** such that we can migrate the complete model in one migration; we once again use the set **CRS** from Page 3 as a starting point and try to construct a migration set **bothMigr** such that at the end of the migration we reach the consistency rules set given by **bothToBe**.

Because of the problems stated earlier in this Section, we do not want to have two negative side rule allowing the  $check_i \rightarrow check_{i+1}$  transition during the migration. This means the worker process needs to be migrated before the scheduler reaches any of the **check<sub>i</sub>** states. The original method of migration only fails this requirement if the  $idle \rightarrow check_1$  rule fires, which means also executing the migration when this rule fires is enough. If we do this, the scheduler migration can take place whenever the trap **finished** is reached, which can be before or after the worker migration has been done. We will need to be ensure the the complete migration is not finished before both migrations are done though. The set **bothToMigr** meets all of these requirements.

*bothMigr* :

<i>Scheduler</i> :	<i>idle</i>	$\longrightarrow$	<i>check<sub>1</sub></i>
* <i>Worker<sub>1</sub>(CSM)</i> :	<i>Free</i>	$\xrightarrow{triv}$	<i>OutCS</i> ,
<i>Worker<sub>2</sub>(CSM)</i> :	<i>Free</i>	$\xrightarrow{triv}$	<i>OutCS</i> ,
<i>Worker<sub>3</sub>(CSM)</i> :	<i>Free</i>	$\xrightarrow{triv}$	<i>OutCS</i>
<i>Scheduler</i> :	<i>asg<sub>i</sub></i>	$\longrightarrow$	<i>check<sub>i+1</sub></i>
* <i>Worker<sub>i</sub>(CSM)</i> :	<i>Busy</i>	$\xrightarrow{done}$	<i>OutCS</i> ,
<i>Worker<sub>i-1</sub>(CSM)</i> :	<i>Free</i>	$\xrightarrow{triv}$	<i>OutCS</i> ,
<i>Worker<sub>i+1</sub>(CSM)</i> :	<i>Free</i>	$\xrightarrow{triv}$	<i>OutCS</i>
<i>McPal</i> :	<i>NewRuleSet</i>	$\longrightarrow$	<i>StartMigr</i>
* <i>McPal(EvolMcPal)</i> :	<i>StablePhase</i>	$\xrightarrow{ready}$	<i>MigrBoth</i>
<i>McPal</i> :	<i>StartMigr</i>	$\longrightarrow$	<i>Both1to2</i>
* <i>Scheduler(EvolSched)</i> :	<i>SchedPhase<sub>1</sub></i>	$\xrightarrow{triv}$	<i>newSchedMigr</i>
<i>McPal</i> :	<i>Both1to2</i>	$\longrightarrow$	<i>Content</i>
* <i>Scheduler(EvolSched)</i> :	<i>newSchedMigr</i>	$\xrightarrow{finished}$	<i>SchedPhase<sub>2</sub></i> ,
<i>Worker<sub>1</sub>(CSM)</i> :	<i>OutCS</i>	$\xrightarrow{triv}$	<i>OutCS</i>
<i>McPal</i> :	<i>Content</i>	$\longrightarrow$	<i>Observing</i>
* <i>McPal(EvolMcPal)</i> :	<i>MigrBoth</i>	$\xrightarrow{migrDone}$	<i>StablePhase</i> ,
	<i>McPal[CRS</i>	$:=$	<i>CRStoBe]</i>

The rule *McPal* : *Both1to2*  $\rightarrow$  *Content* ensures the complete migration is finished by testing for both, thus the complete migration is done in a deterministic manner regardless of the order of the underlying migrations. The migration is visualized in Figure 4.2.

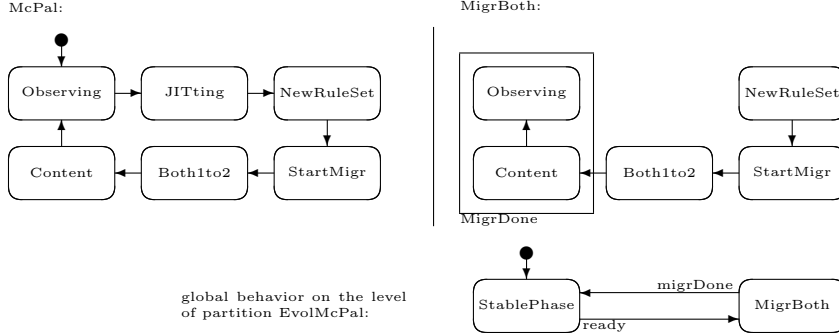


Figure 4.2: McPal and McPal(EvoMcPal) partition during the combined migration

Another possible solution could be to reconstruct the `schedMigr` phase to ensure the only way to reach a `checki` state is through an `asgi` state. Two different solutions will be given in Sections 5 and 6, where we will also solve the problem stated earlier in this Section about the negative side rule.

## 5 An alternative for the Negative Side Rule

In this Section we will first alter the consistency rules set `bothToBe` from 4 to display the same behavior, but without using the negative side rule  $P(\pi) : S \xrightarrow{\theta}$ . Secondly, we will construct a new worker migration which can take place after the scheduler has migrated, yet is deterministic, unlike the migration given by the set `nextWorkerMigr`.

We begin with the set `bothToBe` given on Page 9. To replace the `Scheduler : checki → checki+1` rule without losing the internal consistency ensured by this consistency rule, we need to construct one or more equivalent consistency rules in which the global behavior displayed by the negative side rule, not being in a trap, is captured in a different manner. Not using the negative side rule, we can only ensure global consistency in one way; the manager can only fire if all of it's employees are in the right trap. Therefore we need to add a subprocess to the same partition the negative side rule applies to, which has (at least) two traps; one trap which contains all states in the original trap and one with all other states the subprocess could be in which are not in the original trap. After we have constructed this we can replace the rule containing the negative trap with two rules, one which does not change the detailed state, but changes the subprocess to the new one, and one rule which changes the subprocess back to the former subprocess if the original trap was not entered. We also need to change any other rules which contain that particular trap to test for the trap in the new process, instead of the old one. This is done for the worker in partition

$Worker_i(CSM)$ , visualized in Figure 5.1 and the new CRS is given in `newToBe`.

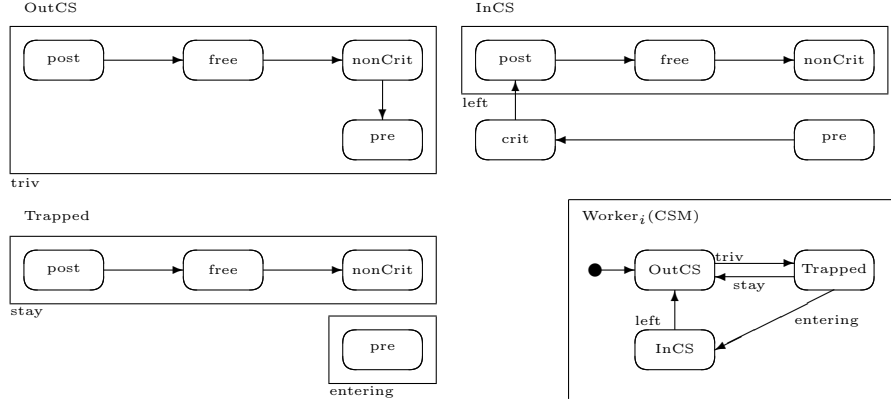


Figure 5.1: New partition  $Worker_i(CSM)$  with subprocess `Trapped`

`newToBe` :

$Worker_i : free \rightarrow nonCrit$   
 $Worker_i : nonCrit \rightarrow pre$   
 $Worker_i : pre \rightarrow crit$

$Worker_i : crit \rightarrow post$   
 $Worker_i : post \rightarrow free$

<i>Scheduler</i> :	$check_i$	$\rightarrow$	$asg_i$
* $Worker_i(CSM)$ :	<i>Trapped</i>	$\xrightarrow{entering}$	<i>InCS</i>
<i>Scheduler</i> :	$asg_i$	$\rightarrow$	$check_{i+1}$
* $Worker_i(CSM)$ :	<i>InCS</i>	$\xrightarrow{left}$	<i>OutCS</i> ,
$Worker_{i+1}(CSM)$ :	<i>OutCS</i>	$\xrightarrow{triv}$	<i>Trapped</i>
<i>Scheduler</i> :	$check_i$	$\rightarrow$	$check_{i+1}$
* $Worker_i(CSM)$ :	<i>Trapped</i>	$\xrightarrow{stay}$	<i>OutCS</i> ,
$Worker_{i+1}(CSM)$ :	<i>OutCS</i>	$\xrightarrow{triv}$	<i>Trapped</i>
<i>McPal</i> :	<i>Observing</i>	$\rightarrow$	<i>JITting</i>
<i>McPal</i> :	<i>JITting</i>	$\rightarrow$	<i>NewRuleSet</i>
*	$McPal[CRS$	$:=$	$CRS \cup Crs_{mig} \cup Crs_{toBe}]$

When we consider the migrations from Section 4, we find all three migrations need changes to facilitate the migration to `newToBe`. The most important change to `bothToBe` is that exactly two of the worker subprocesses are in `OutCS`, while the remaining subprocess is either in `Trapped` or `InCS`. Beginning with the migration starting in `firstWorkerToBe`, we find the migration needs only one change to ensure this. If the scheduler is in one of the  $asg_i$  states, one of the  $Worker_i(CSM)$  is in the `InCS` subprocess, and the migration can take place

without changes. If the scheduler is in state `idle` though, all workers are in the `OutCS` subprocess, so we need to have the  $Scheduler : idle \rightarrow check_1$  ensure  $Worker_1$  enters subprocess `Trapped`; therefore the rule should be replaced by

$$\begin{aligned} Scheduler : & \quad idle \longrightarrow check_1 \\ * Worker_1(CSM) : & \quad OutCS \xrightarrow{triv} Trapped \end{aligned}$$

While `nextWorkerMigr` could be changed in the same manner to yield a migration from `firstSchedToBe`, the model will still not be deterministic and could cause a deadlock. The negative side rule in `firstSchedToBe` creates this problem; when the workers have all left the `Free` subprocess, the  $check_i \rightarrow check_{i+1}$  rule could continuously fire until `McPal` removes the rules from `firstSchedToBe` from the set. This could lead to a situation where the worker being checked by the scheduler is not the same as the worker which has entered the `Trapped` subprocess. We therefore need to substitute the negative side rule in `firstSchedToBe` in the same manner given earlier in this Section. This is done in figure 5.2 and the following consistency rule set `newFirstSchedToBe`

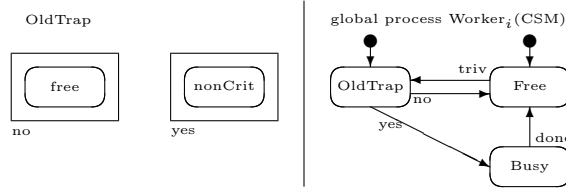


Figure 5.2:  $Worker_i(CSM)$  and subprocess `oldTrap`; `firstSchedulerToBe` without the negative side rule

$$\begin{aligned} newFirstSchedToBe : \\ Worker_i : free & \longrightarrow nonCrit & Worker_i : crit & \longrightarrow post \\ Worker_i : nonCrit & \longrightarrow pre & Worker_i : post & \longrightarrow free \\ Worker_i : pre & \longrightarrow crit \\ \\ Scheduler : & \quad check_i \longrightarrow asg_i \\ * Worker_i(CSM) : & \quad OldTrap \xrightarrow{yes} Busy \\ Scheduler : & \quad asg_i \longrightarrow check_{i+1} \\ * Worker_i(CSM) : & \quad Busy \xrightarrow{done} Free, \\ & \quad Worker_{i+1}(CSM) : Free \xrightarrow{triv} OldTrap \\ Scheduler : & \quad check_i \longrightarrow check_{i+1} \\ * Worker_i(CSM) : & \quad OldTrap \xrightarrow{no} Free, \\ & \quad Worker_{i+1}(CSM) : Free \xrightarrow{triv} OldTrap \\ \\ McPal : Observing & \longrightarrow JITting \\ McPal : JITting & \longrightarrow NewRuleSet \\ * McPal[CRS & := CRS \cup Crs_{mig} \cup Crs_{toBe}] \end{aligned}$$

Using this model, a new consistency rule set is constructed below, which successfully migrates to **newToBe**; figure 5.3 visualizes this migration;

*newNextWorkerMig* :

<i>Scheduler</i> :	<i>asg<sub>i</sub></i>	$\longrightarrow$	<i>check<sub>i+1</sub></i>
* <i>Worker<sub>i</sub>(CSM)</i> :	<i>Busy</i>	$\xrightarrow{\text{done}}$	<i>OutCS</i> ,
<i>Worker<sub>i-1}(CSM)</sub></i> :	<i>Free</i>	$\xrightarrow{\text{triv}}$	<i>OutCS</i> ,
<i>Worker<sub>i+1}(CSM)</sub></i> :	<i>Free</i>	$\xrightarrow{\text{triv}}$	<i>Trapped</i> ,
<i>McPal(EvolMcPal)</i> :	<i>MigrWorker</i>	$\xrightarrow{\text{triv}}$	<i>MigrWorker</i>

<i>McPal</i> :	<i>NewRuleSet</i>	$\longrightarrow$	<i>StartMigr</i>
* <i>McPal(EvolMcPal)</i> :	<i>StablePhase</i>	$\xrightarrow{\text{ready}}$	<i>MigrWorker</i>
<i>McPal</i> :	<i>StartMigr</i>	$\longrightarrow$	<i>Content</i>
* <i>Worker<sub>1</sub>(CSM)</i> :	<i>OutCS</i>	$\xrightarrow{\text{triv}}$	<i>OutCS</i>
<i>McPal</i> :	<i>Content</i>	$\longrightarrow$	<i>Observing</i>
* <i>McPal(EvolMcPal)</i> :	<i>MigrWorker</i>	$\xrightarrow{\text{migrDone}}$	<i>StablePhase</i> ,
	<i>McPal[CRS]</i>	$:=$	<i>CrstoBe</i> ]

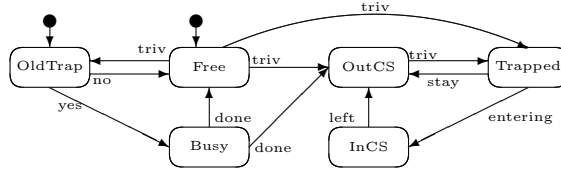


Figure 5.3: Migration starting from **newFirstSchedMig** to **newToBe**

Finally we consider the migration from **CRS** to **newToBe**; here we find replacement of the scheduler rules in **bothMigr** by

<i>Scheduler</i> :	<i>idle</i>	$\longrightarrow$	<i>check<sub>1</sub></i>
* <i>Worker<sub>1</sub>(CSM)</i> :	<i>Free</i>	$\xrightarrow{\text{triv}}$	<i>Trapped</i> ,
<i>Worker<sub>2</sub>(CSM)</i> :	<i>Free</i>	$\xrightarrow{\text{triv}}$	<i>OutCS</i> ,
<i>Worker<sub>3</sub>(CSM)</i> :	<i>Free</i>	$\xrightarrow{\text{triv}}$	<i>OutCS</i>
<i>Scheduler</i> :	<i>asg<sub>i</sub></i>	$\longrightarrow$	<i>check<sub>i+1</sub></i>
* <i>Worker<sub>i</sub>(CSM)</i> :	<i>Busy</i>	$\xrightarrow{\text{done}}$	<i>OutCS</i> ,
<i>Worker<sub>i-1}(CSM)</sub></i> :	<i>Free</i>	$\xrightarrow{\text{triv}}$	<i>OutCS</i> ,
<i>Worker<sub>i+1}(CSM)</sub></i> :	<i>Free</i>	$\xrightarrow{\text{triv}}$	<i>Trapped</i>

is enough to ensure a consistent migration. This change is comparable to the change used to create a new migration set from **firstWorkerToBe** to **newToBe**. The migration is visualized in Figure 5.4.

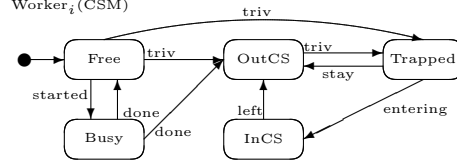


Figure 5.4: New form of  $Worker_i$  (CSM) during the migration

## 6 Using a Phased McPal

Until now, we have held on to the idea of one step which adds the new consistency rules and one which remove the old ones. When we examine the problem with multiple negative side rules given in Section 4 though, we find splitting these steps into multiple steps solves the problem too. We can construct a McPal which adds the rules of the migration to the set first:

$$\begin{aligned}
 McPal : \quad & JITting \longrightarrow MigrRuleSet \\
 * \quad & McPal[CRS := CRS \cup Crs_{mig}]
 \end{aligned}$$

It then uses  $Crs_{mig}$  to add  $Crs_{toBe}$  only when those rules are needed. This can solve the multiple negative side rule problem by adding the new negative side rule simultaneous with removing the old one. This can be done using the rules below for the migration from  $firstSchedToBe$  to  $BothToBe$  (remember,  $firstSchedToBe$  is slightly altered; the  $JITting \rightarrow NewRuleSet$  is replaced by the rule above):

$$\begin{aligned}
 & phasedNextWorkerMigr : \\
 McPal : \quad & MigrRuleSet \longrightarrow StartMigr \\
 * \quad & McPal(EvolMcPal) : StablePhase \xrightarrow{ready} MigrWorker \\
 McPal : \quad & StartMigr \longrightarrow Content \\
 * \quad & Worker_1(CSM) : OutCS \xrightarrow{triv} OutCS \\
 McPal : \quad & Content \longrightarrow Observing \\
 * \quad & McPal(EvolMcPal) : MigrWorker \xrightarrow{migrDone} StablePhase, \\
 & \quad \quad \quad McPal[CRS := Crs_{toBe}] \\
 \\ 
 Scheduler : \quad & asg_i \longrightarrow check_{i+1} \\
 * \quad & Worker_i(CSM) : Busy \xrightarrow{done} OutCS, \\
 & Worker_{i-1}(CSM) : Free \xrightarrow{triv} OutCS, \\
 & Worker_{i+1}(CSM) : Free \xrightarrow{triv} OutCS, \\
 McPal(EvolMcPal) : \quad & MigrWorker \xrightarrow{triv} MigrWorker, \\
 & \quad \quad \quad McPal[CRS := Crs_{mig} \cup Crs_{toBe}]
 \end{aligned}$$

## 7 Conclusion and Future Research

We showed the new McPal component was capable of reproducing the migrations from [2], only needing to add rules to prevent the model from migrating before McPal has left `stablePhase`, one of the requirements of the new McPal component. After these changes, we combined the migrations; first by executing them in any order, then by creating a new migration set which migrates the complete model in one step. We found these migrations needed little changes from the ones given in Section 3, but one of the migrations was not consistent during the migration. This showed us that even if the three consistency rules sets  $CRS$ ,  $Crsmig$  and  $CrstoBe$  are internally consistent, the unification of the three sets need not be. In this specific case, we found a situation in which both the current  $CRS$  and  $CrstoBe$  contained a negative side rule; requiring the same partition to either not be in state `Free` (from  $CRS$ ) or not in state `OutCS` (from  $CrstoBe$ ). During the migration, at least one of these rules could always fire, which was not consistent with the required functionality of the model.

We presented two solutions for this problem, the first being an alternative for the negative side rule, using a new subprocess to determine whether the model is or isn't in the desired trap. This model has the advantage that the unification of the three consistency rule sets is more likely to behave as desired, while the negative side rule will always display this problem if the subprocesses in the relevant partition change during migration. This solution has the disadvantage of not being a simple "default" option; the negative side rule was a simple rule to allow a model to proceed if it was not in the relevant subprocess, modeling this using the new form would require one rule per subprocess in the dimension.

Another solution to this problem was not to have the old and new consistency rules sets be contained in  $CRS$  simultaneously. To do this, we created a multi-phase migration, in which during the first phase the  $CRS$  contains the old rules and the migration rules, while in the second phase,  $CRS$  consists of the migration rules and the new rules. This model has the advantage of solving every possible problem with the unification of sets during the migration, but the disadvantage of possibly needing multiple migration sets if more than one problem exists with the non-phased model. This method of generally solving similar problems could be researched in more detail.

A third solution was not presented, because the current Paradigm system requires all models to be solid-frame; adding, removing and renaming dimensions in the cartesian product space of the model is currently not possible. If it were, we could construct a migration which uses a temporary dimension to store the  $CRS$  and  $CrstoBe$  set during the migration, and having the global behavior completely removed from the original dimension. In the `scheduler` migration, this would mean the `schedMigr` phase in the partition `Scheduler(EvolSched)` becomes empty, preventing all scheduler rules dependent on it from firing. Meanwhile, the migration set contains the required migrational transitions which are dependent on the temporary dimensions. This prevents the problems mentioned above by not having the two negative side rules in the same dimension, possibly even requiring both rules to apply before executing the rule.

## References

- [1] L.P.J. Groenewegen, A.W. Stam, P.J. Toussaint and E.P. de Vink. *Paradigm as Organization-Oriented Coordination Language*.
- [2] L.P.J. Groenewegen and E. de Vink. *Evolution On-the-Fly with Paradigm*.