# Formalizing Adaptation On-the-Fly

S. Andova[a], L.P.J. Groenewegen[1,b], J. Stafleu[b] & E.P. de Vink[a]

[a] *Department of Mathematics and Computer Science, TU/e, Eindhoven, the Netherlands*

[b] *FaST Group, LIACS, Leiden University, The Netherlands*

**Abstract**

Paradigm models specify coordination of collaborating components via constraint control. Component McPal allows for later addition of new constraints and new control in view of unforeseen adaptation. After addition McPal starts coordinating migration accordingly, adapting the system towards to-be collaboration. Once done, McPal removes obsolete control and constraints. All coordination remains ongoing while migrating on-the-fly, being deflected without any quiescence. Through translation into process algebra, supporting formal analysis is arranged carefully, showing that as-is and to-be processes are proper abstractions of the migrating process. A canonical critical section problem illustrates the approach.

## 1 Introduction

Coordination language Paradigm [1] models the dynamics of collaborating components. Collaboration is specified by loosely coupling detailed local dynamics of participants to protocol dynamics via role dynamics. In a two-sided way, a role dynamically imposes a current constraint both on a participant's next steps (phase) and on a protocol's next steps (trap). Figure 1 gives such collaborations in UML 2.0 style as dashed ovals.

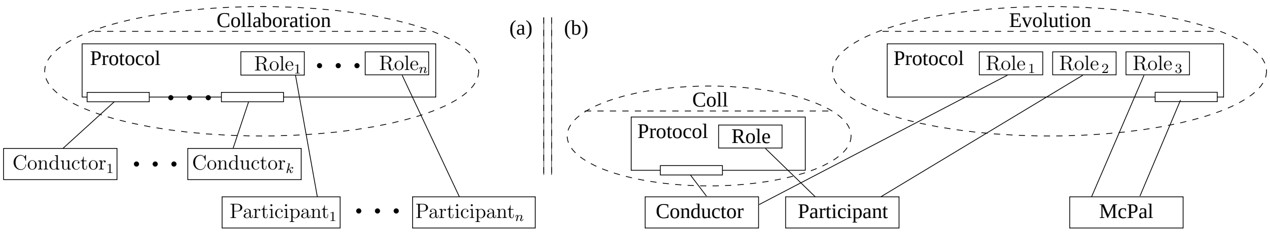


Fig. 1. Collaboration, protocol, roles, participants and conductors.

In Figure 1a, Collaboration presents the general structure of Paradigm collaborations. Participants contribute via Roles, in turn composed into a Protocol by synchronizing role steps. Conductors can be involved too, recognizable by a thin box

[1] Corresponding author, email `luuk@liacs.nl`.

*This paper is electronically published in*
*Electronic Notes in Theoretical Computer Science*
*URL:* `www.elsevier.nl/locate/entcs`

across the protocol border. A Conductor conducts synchronization of role steps in a single step of the protocol. In UML 2.0 dynamic consistency is still problematic, see [13,11]. Particularly for general UML collaborations, dynamic consistency between participants, roles and collaboration interaction is not clear. If, moreover, such a collaboration has to change, dynamic consistency is even more problematic, particularly so *during* migration. For a similar reason, the notion of *quiescence* has been introduced [12] for adaptive systems: a system part, having to change while the system is ongoing, is isolated first from its environment, then it is changed, e.g. by replacing it, and finally, the part in its renewed form is reconnected. Thus, quiescence circumvents dynamic consistency problems in ongoing collaborations during the actual change, by separating a part from what remains ongoing.

Paradigm models for collaborations are dynamically consistent [8,1]: so-called *phase* and *trap* constraints guarantee consistency between a Participant and any Role of it (vertical consistency); so-called *consistency rules*, defining a Protocol, guarantee consistency between the Roles and the Conductors (horizontal consistency). Interestingly, adaptation in Paradigm can be formulated as coordination of a once-only migration collaboration from ongoing as-is collaboration to to-be collaboration aimed at. In particular, migration can be done without any quiescence, thus maintaining dynamical consistency before, during and after migration. Merely *structurally*, Figure 1b visualizes a schematic and simplified, far from general migration example: both as-is and to-be collaboration, named Coll, are identical, with only one participant, one role, and one conductor. (Though not specified in the –merely structural– diagram, their as-is and to-be dynamics do differ.) In addition, a separate collaboration Evolution has every participant and every conductor of Coll as participant, via one separate role each. A special but generally applicable component McPal is involved too [1], both as participant and as conductor of Evolution. Its specialty lies in its dynamics.

McPal's dynamics and the interplay thereof with the larger Paradigm model are organized as follows. Initially, as long as a given as-is coordination situation remains stable, a Paradigm model specifies and performs as-is coordination between model components with their as-is dynamics ongoing. Nevertheless, special component McPal is in place in so-called *hibernating* form, not involved in the ongoing as-is coordination at all, but having the ability to extend the model with to-be coordination as well as with migration coordination from as-is to to-be. Only after such an extension has been specified well and subsequently installed, McPal awakes from hibernating to start adapting dynamics and coordination gradually, from as-is into to-be, as specified in terms of the migration coordination just added. Once done, McPal retires into hibernation, removing model specification parts no longer needed, while the to-be coordination situation remains stable until further notice, as the Paradigm model now specifies and performs to-be coordination between its components with their to-be dynamics ongoing. In fact, we have a form of quiescence for McPal, but activity of other components is not interrupted. Here, the quiescence is mirrored, as McPal is active during migration.

Process algebra (PA) provides a specification formalism for describing Paradigm

[1] The name McPal is short for Managing Changing Processes Ad Libitum.

models in a precise and structural way [1]. Collaborating components are represented in PA by recursive specifications. Dynamic constraints and consistency rules are reflected in the synchronizing function of the parallel operator of the process algebra we consider, defining how components communicate. Thus, a Paradigm model of an adapting system, including the special component McPal, is translated into PA. So, using a well established abstraction technique of PA, we can formally analyze the adaptation process. For instance, we can prove that as-is collaboration indeed migrates to to-be collaboration. In particular, the PA model makes the adaptation dynamics explicit. Therefore, for every migration trajectory, progress properties can be verified.

To clarify the above, the paper has four sections. Section 2 recapitulates Paradigm through a nondeterministic critical section solution, with McPal in place, going to migrate the example. In addition, the section addresses the suitability of the same McPal for general unforeseen migration of arbitrary Paradigm models. In Section 3, PA analysis of the adaptation is presented, for the example first and subsequently for the general case. Section 4 closes with comparing McPal to earlier versions, with variants of McPal in form and performance, with related work and with ideas for future work.

## 2 On-the-fly migration through coordination

In view of explaining McPal, this section first repeats Paradigm's basic notions. Second, it presents a concrete as-is Paradigm model, with McPal in place in hibernating form. Third, it presents a concrete to-be model, with McPal returned to hibernating. Fourth, given the to-be goal, it presents migration coordination from as-is to to-be, conducted by McPal, only while not hibernating. Fifth, we abstract from the example by discussing general adaptation of Paradigm models through migration coordination conducted by McPal, with its hibernating form the same. Except for McPal we shall keep our explanation brief.

The following definitions present Paradigm's basic notions: state-transition diagram, phase, (connecting) trap, partition and global process, see also [1].

- A *state-transition diagram* (*STD*) is a triple $\langle \mathtt{ST}, \mathtt{AC}, \mathtt{TS} \rangle$ with $\mathtt{ST}$ the set of states, $\mathtt{AC}$ the set of actions and $\mathtt{TS} \subseteq \mathtt{ST} \times \mathtt{AC} \times \mathtt{ST}$ the set of transitions or steps. A step $(x, a, x') \in \mathtt{TS}$, denoted by $x \xrightarrow{a} x'$, is said to be from $x$ to $x'$.
- A *phase* of STD $\langle \mathtt{ST}, \mathtt{AC}, \mathtt{TS} \rangle$ is an STD $S = \langle \mathtt{st}, \mathtt{ac}, \mathtt{ts} \rangle$ such that $\mathtt{st} \subseteq \mathtt{ST}$, $\mathtt{ac} \subseteq \mathtt{AC}$ and $\mathtt{ts} \subseteq \{ (x, a, x') \in \mathtt{TS} \mid x, x' \in \mathtt{st}, a \in \mathtt{ac} \}$.
- A *trap* $t$ of phase $S = \langle \mathtt{st}, \mathtt{ac}, \mathtt{ts} \rangle$ is a non-empty set of states $t \subseteq \mathtt{st}$ such that $x \in t$ and $x \xrightarrow{a} x' \in \mathtt{ts}$ imply $x' \in t$. A trap *connects* phase $S$ it belongs to, to another phase $S' = \langle \mathtt{st}', \mathtt{ac}', \mathtt{ts}' \rangle$ if $t \subseteq \mathtt{st}'$, notation $S \xrightarrow{t} S'$, called *phase transfer*. If $t = \mathtt{st}$, trap $t$ is called *trivial*, denoted as $\mathsf{triv}(S)$.
- A *partition* $\pi = \{ (S_i, T_i) \mid i \in I \}$ of an STD $Z$ is a set of phases $S_i$ of $Z$ and a set of traps $T_i$ of $S_i$, typically $\mathsf{triv}(S_i) \in T_i$. A *role* or *global STD* at the level of partition $\pi$ is an STD $Z(\pi) = \langle \mathtt{GST}, \mathtt{GAC}, \mathtt{GTS} \rangle$ with $\mathtt{GST} \subseteq \{ S_i \mid i \in I \}$, $\mathtt{GAC} \subseteq \bigcup_{i \in I} T_i$ and $\mathtt{GTS} \subseteq \{ S_i \xrightarrow{t} S_j \mid i, j \in I, t \in \mathtt{GAC} \}$ a set of phase transfers. $Z$ is called the *detailed* STD underlying *global* STD $Z(\pi)$, the $\pi$-role of $Z$.

A phase, when being current state of a role, is a dynamic constraint imposed on the detailed STD underlying the role, containing all transitions allowed by the role in that phase. A connecting trap of a phase is a further dynamic constraint committed to by the detailed STD, serving as guard for a phase transfer, often to be carried out in combination with simultaneous phase transfers in other roles.

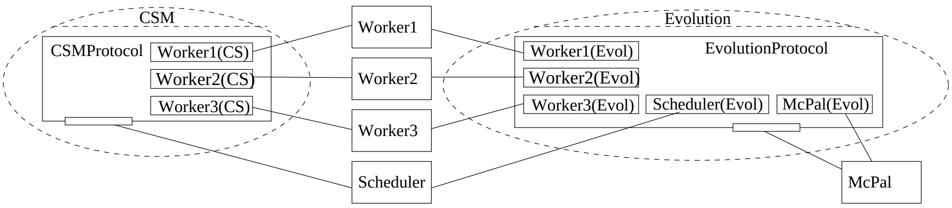


Fig. 2. Collaborations CSM and Evolution.

The as-is model we want to present, is a variant of the nondeterministic server solution for a critical section problem, with three Workers and with Scheduler serving them, see Figure 2. Each $\mathsf{Worker}_i(\mathsf{CS})$ role is contributed to CSM by $\mathsf{Worker}_i$. Moreover, Scheduler is involved too, as the only conductor. (Collaboration Evolution is addressed separately below.)

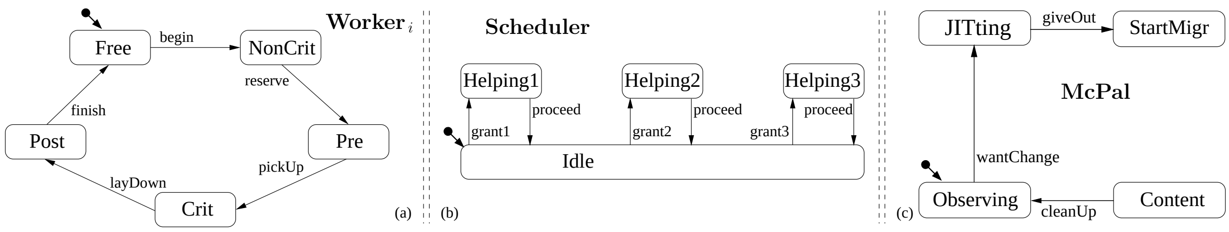


Fig. 3. Participant dynamics: (a) Worker (b) Scheduler (c) McPal.

STDs for Workers and Scheduler, covering the as-is situation only, are given in Figure 3ab. Being in as well as going to and leaving state Crit constitute a Worker's critical section activities. Therefore, Figure 4a presents phase NotHaving of a Worker as detailed STD fragment, reflecting a Worker's allowed dynamics when *not having* the permission for doing its critical work[2]. Similarly, phase Having reflects a Worker's dynamics when *having* that permission. Additional polygons indicate a phase' trap, containing the trap's states: request is connecting to phase Having and done is connecting to NotHaving, paving the way for three roles $\mathsf{Worker}_i(\mathsf{CS})$, see Figure 4b.

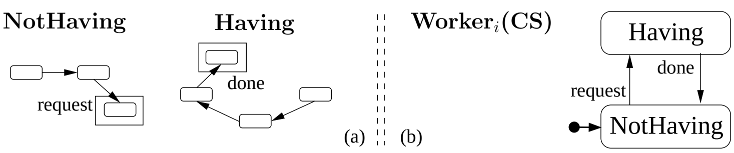


Fig. 4. CS constraints: (a) phases and traps (b) role for $\mathsf{Worker}_i$.

Note, *starting states* of Workers, of their roles and of Scheduler, pointed at by a dot-and-arrow in UML-style, are consistent. Starting state Free belongs to NotHaving, the 'starting constraint'. Moreover, Scheduler is supposed to regulate each Worker's critical section entrance. Thus, in its starting state Idle, with each Worker in

[2] For reason of space, state and action names from Figure 3a are not repeated in 4a, but the form of the original is kept in the fragments.

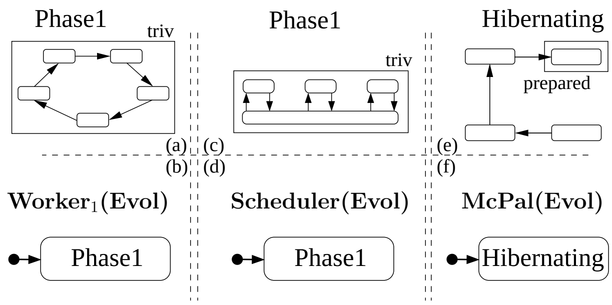


Fig. 5. Evol constraints for single $\mathsf{Worker}_i$, Scheduler and McPal.

NotHaving, it starts refusing permission to each Worker. The question then is, how their combined dynamics stay consistent once started. Informally, Scheduler is giving the permission to $\mathsf{Worker}_i$ only by going to $\mathsf{Helping}_i$ and it withdraws permission by returning to Idle.

$$\mathsf{Scheduler}\colon \mathsf{Idle} \overset{grant_i}{\rightarrow} \mathsf{Helping}_i \ * \ \mathsf{Worker}_i(\mathsf{CS})\colon \mathsf{NotHaving} \overset{request}{\rightarrow} \mathsf{Having}$$
$$\mathsf{Scheduler}\colon \mathsf{Helping}_i \overset{proceed}{\rightarrow} \mathsf{Idle} \ * \ \mathsf{Worker}_i(\mathsf{CS})\colon \mathsf{Having} \overset{done}{\rightarrow} \mathsf{NotHaving}$$

The above two consistency rules let Scheduler conduct the CS roles of the Workers. The roles, by imposing the current phase, in turn dynamically constrain the five detailed steps of each Worker. Some of these steps actually lead to entering a connecting trap. Via such a trap entered, detailed STDs dynamically constrain coordination steps of Scheduler. In general, a consistency rule synchronizes single steps from *different* STDs: zero or one detailed steps, zero or more role steps and a so-called *change clause* to update the consistency rules within a protocol. If present in one rule, the role steps together constitute one protocol step. If synchronized with a detailed step of an STD $M$, this $M$ is referred to as the *conductor* of that protocol step.

So the first rule says, by its step to $\mathsf{Helping}_i$, Scheduler conducts $\mathsf{Worker}_i$'s CS role step to phase Having, provided trap request has been entered within NotHaving. Similarly the second rule says, by returning to Idle, Scheduler conducts $\mathsf{Worker}_i$'s CS role to return to NotHaving, provided trap done has been entered within phase Having.

$$\mathsf{Worker}_i\colon \mathsf{Free} \overset{begin}{\rightarrow} \mathsf{NonCrit} \qquad \mathsf{Worker}_i\colon \mathsf{Pre} \overset{pickUp}{\rightarrow} \mathsf{Crit} \qquad \mathsf{Worker}_i\colon \mathsf{Post} \overset{finish}{\rightarrow} \mathsf{Free}$$
$$\mathsf{Worker}_i\colon \mathsf{NonCrit} \overset{reserve}{\rightarrow} \mathsf{Pre} \qquad \mathsf{Worker}_i\colon \mathsf{Crit} \overset{layDown}{\rightarrow} \mathsf{Post}$$

The second group has five, more simple rules. No marker '$*$' means, isolated detailed steps only, without any synchronization. But current phases do restrict the actual taking of a step. E.g., regarding the third rule, a Worker can do action pickUp only if its current phase is Having as the phase NotHaving does not allow this transition.

Until now we did not take the Evol roles into account. Figure 5bd specifies them in terms of one phase $\mathsf{Phase}_1$ each. Each $\mathsf{Phase}_1$ does not really restrict underlying detailed dynamics, as it contains every detailed step, see Figure 5ac. As each role $\mathsf{Worker}_i(\mathsf{Evol})$ and Scheduler(Evol) starts as well as remains residing in its only global state $\mathsf{Phase}_1$, as-is dynamics presented above are not influenced (yet). Moreover, each $\mathsf{Phase}_1$ has a trivial trap, intended to be connecting to a next phase unknown as yet, but at least allowing for future interruption at any moment. McPal is in place as conductor of the Evolution protocol, however. See Figure 2. According to

Figures 3c and 5ef its detailed STD starts in Observing and its own Evol role starts in Hibernating. The phase Hibernating has trap prepared, intended to be connecting to a next phase unknown for a while, but known indeed *when* trap prepared, i.e. state StartMigr, is entered, in view of the change clause in the second consistency rule below.

$$\mathsf{McPal}\colon \mathsf{Observing} \overset{wantChange}{\rightarrow} \mathsf{JITting}$$
$$\mathsf{McPal}\colon \mathsf{JITting} \overset{giveOut}{\rightarrow} \mathsf{StartMigr} \quad * \quad \mathsf{McPal}\colon [\mathsf{Crs}\colon = \mathsf{Crs} + \mathsf{Crs}_{migr} + \mathsf{Crs}_{toBe}]$$
$$\mathsf{McPal}\colon \mathsf{Content} \overset{clearUp}{\rightarrow} \mathsf{Observing} \quad * \quad \mathsf{McPal}\colon [\mathsf{Crs}\colon = \mathsf{Crs}_{toBe}]$$

According to the above three consistency rules for McPal's detailed steps, the first is without any conducting. When in state Observing, McPal can start private preparation of still unknown migration at leasure. Preparation occurs when in JITting, through input –e.g. from a modeler– or through McPal's own activity. It results in a new Paradigm model, covering a to-be situation, as well as migration trajectories towards it. Thus, the second rule too is without any conducting, but here the change clause couples McPal's detailed step giveOut to an update of the consistency rules, extending Crs with new rules both for a to-be situation, collected in set $\mathsf{Crs}_{toBe}$, and for a migration situation from as-is to to-be, collected in set $\mathsf{Crs}_{migr}$. The original content of Crs consists of the as-is situation as specified through the above ten rules. Thus, a Paradigm model with a hibernating McPal in place is *reflective* as the model contains its own specification. In addition, it extends its specification while keeping its dynamics unchanged, ongoing as before: McPal's second rule. The third rule specifies, once migration has been done, by returning to Observing, all model specification fragments obsolete by then, are removed.

The to-be situation aimed at is a variant solution for CSM: pursuing a round robin strategy augmented with more efficient permission withdrawal, by asking for withdrawal sooner and by delaying the necessity to wait for it. Figure 2 remains the same, as collaborations and protocol structures do not change. But detailed STDs for Workers and Scheduler are different, see Figure 6. By spanning as-is, migration and to-be situations together, the figures get less clear however, missing a historical overview in the details. Figure 7abcd alleviates this via Evol phases, traps and roles. It shows in particular, the Workers suddenly get more dynamic freedom as more direct steps from Post towards Pre can be taken, whereas Scheduler exhibits special intermediate dynamics in phase NDetToRoRo before conducting in mere round robin fashion. Note, the round robin fashion emerges from Scheduler's cycling through states $\mathsf{Checking}_i$, possibly alternated with going to $\mathsf{Helping}_i$ if $\mathsf{Worker}_i$ asks for it.

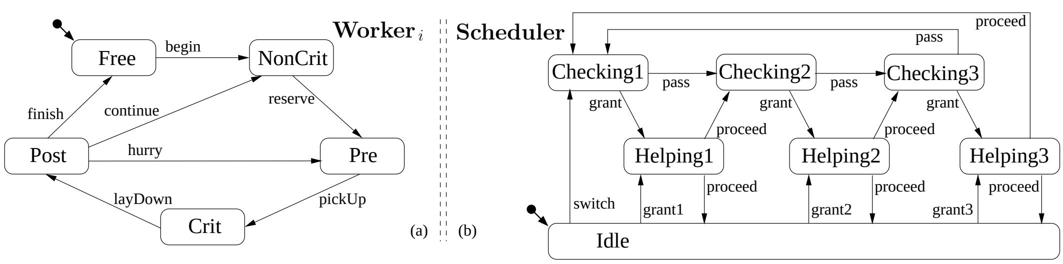


Fig. 6. Detailed STDs for Workers and Scheduler for migration in its entirety.

The CS role of Worker, see Figure 8, changes too: ($i$) New CS phases and traps must cover the newly added dynamics in Worker's Evol phase $\mathsf{Phase}_2$. Thus, Without is the new version of NotHaving and With is the new version of Having. (*ii*) In view of the round robin approach, Interrupt is needed as interrupted form of Without, enabling discrimination between needing permission for critical work or not, on the basis of different traps entered.

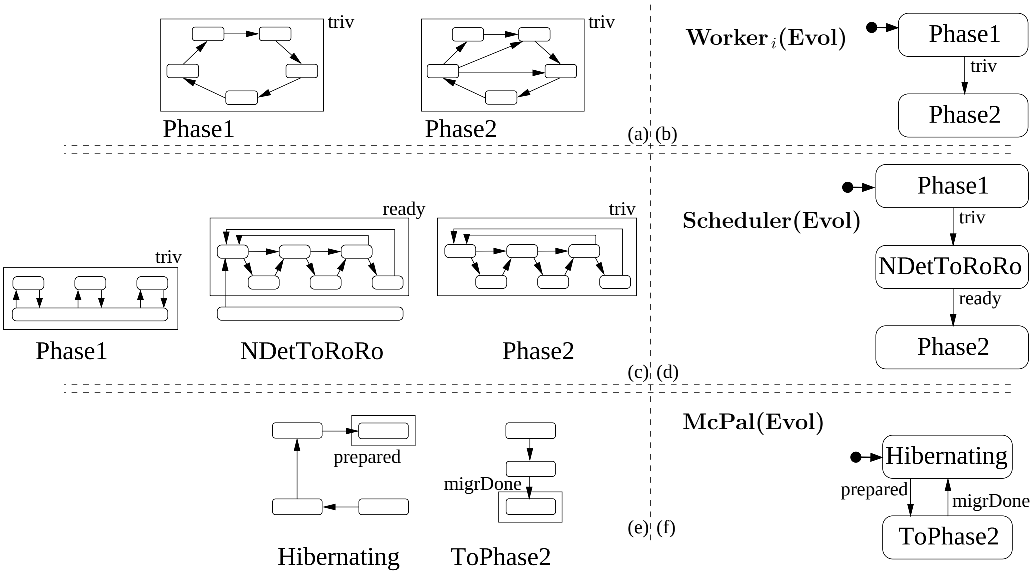


Fig. 7. Evol constraints and role for any participant's entire migration.

The ten consistency rules below specify this, using cyclic indexing. They belong to the set $\mathsf{Crs}_{toBe}$. Note, some rules from Crs, originally specifying as-is dynamics, are still there.

$$\mathsf{Scheduler}\colon \mathsf{Checking}_i \xrightarrow{grant} \mathsf{Helping}_i \;\;*\;\; \mathsf{Worker}_i(\mathsf{CS})\colon \mathsf{Interrupt} \xrightarrow{request} \mathsf{With}$$

$$\mathsf{Scheduler}\colon \mathsf{Helping}_i \xrightarrow{proceed} \mathsf{Checking}_{i+1} \;\;*$$
$$\mathsf{Worker}_i(\mathsf{CS})\colon \mathsf{With} \xrightarrow{done} \mathsf{Without},\;\; \mathsf{Worker}_{i+1}(\mathsf{CS})\colon \mathsf{Without} \xrightarrow{triv} \mathsf{Interrupt}$$

$$\mathsf{Scheduler}\colon \mathsf{Checking}_i \xrightarrow{pass} \mathsf{Checking}_{i+1} \;\;*$$
$$\mathsf{Worker}_i(\mathsf{CS})\colon \mathsf{Interrupt} \xrightarrow{notYet} \mathsf{Without},\;\; \mathsf{Worker}_{i+1}(\mathsf{CS})\colon \mathsf{Without} \xrightarrow{triv} \mathsf{Interrupt}$$

$$\mathsf{Worker}_i\colon \mathsf{Free} \xrightarrow{begin} \mathsf{NonCrit} \qquad \mathsf{Worker}_i\colon \mathsf{Pre} \xrightarrow{pickUp} \mathsf{Crit} \qquad \mathsf{Worker}_i\colon \mathsf{Post} \xrightarrow{finish} \mathsf{Free}$$
$$\mathsf{Worker}_i\colon \mathsf{NonCrit} \xrightarrow{reserve} \mathsf{Pre} \qquad \mathsf{Worker}_i\colon \mathsf{Crit} \xrightarrow{layDown} \mathsf{Post} \qquad \mathsf{Worker}_i\colon \mathsf{Post} \xrightarrow{continue} \mathsf{NonCrit}$$
$$\mathsf{Worker}_i\colon \mathsf{Post} \xrightarrow{hurry} \mathsf{Pre}$$

Also, the same three rules for McPal discussed above, belong to $\mathsf{Crs}_{toBe}$.

Suggested already in Figure 7e, the STD of McPal, covering migration in its entirety, is given in Figure 9. Apparently, for conducting the migration it has two originally unforeseen detailed steps, leading from StartMigr via StartRoRo to Content. In addition, separating these two unforeseen steps from the three foreseen steps in Hibernating, role McPal(Evol) has to perform two steps, one swapping from Hibernating to $\mathsf{ToPhase}_2$, reflecting 'awakening', and the other swapping back, reflecting 'retiring', both global steps being unforeseen too as McPal(Evol)'s actual migration phase $\mathsf{ToPhase}_2$ was originally unknown. Swapping between McPal's own Evol phases is specified through two choreography steps without a conductor instead of through orchestration steps having a conductor. According to the choreography, ‘awakening’ comes first, ‘retiring’ comes second. The notion of choreography for Paradigm has been adopted from [16].

$* \ \mathsf{McPal(Evol)\colon Hibernating} \stackrel{prepared}{\rightarrow} \mathsf{ToPhase_2} \qquad * \ \mathsf{McPal(Evol)\colon ToPhase_2} \stackrel{migrDone}{\rightarrow} \mathsf{Hibernating}$

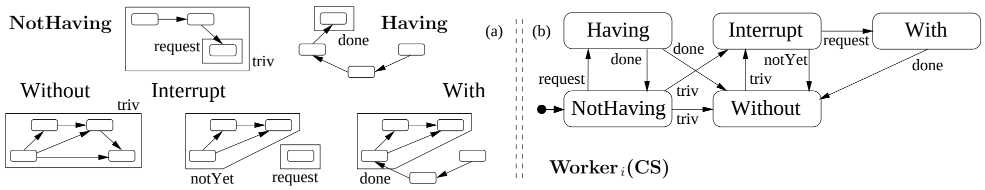


Fig. 8. $\mathsf{CS}$ constraints and role for any $\mathsf{Worker}$’s entire migration.

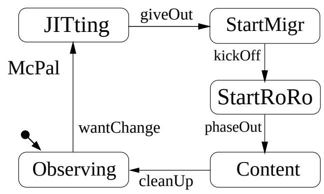


Fig. 9. STD of $\mathsf{McPal}$ for the entire migration.

The two choreography rules above together with the six rules below constitute the set $\mathsf{Crs}_{migr}$. Two rules with Scheduler conducting, address Scheduler’s first step to whatever Checking state, thereby synchronously and consistently transferring all Workers from their as-is CS phases to their to-be CS phases.

$$\begin{aligned}
&\mathsf{Scheduler\colon Idle} \stackrel{switch}{\rightarrow} \mathsf{Checking_1} \ * \ \mathsf{Worker_1(CS)\colon NotHaving} \stackrel{triv}{\rightarrow} \mathsf{Interrupt},\\
&\quad \mathsf{Worker_2(CS)\colon NotHaving} \stackrel{triv}{\rightarrow} \mathsf{Without},\ \mathsf{Worker_3(CS)\colon NotHaving} \stackrel{triv}{\rightarrow} \mathsf{Without}\\
&\mathsf{Scheduler\colon Helping}_i \stackrel{proceed}{\rightarrow} \mathsf{Checking}_{i+1} \ * \ \mathsf{Worker}_i\mathsf{(CS)\colon Having} \stackrel{done}{\rightarrow} \mathsf{Without},\\
&\quad \mathsf{Worker}_{i+1}\mathsf{(CS)\colon NotHaving} \stackrel{triv}{\rightarrow} \mathsf{Interrupt},\ \mathsf{Worker}_{i-1}\mathsf{(CS)\colon NotHaving} \stackrel{triv}{\rightarrow} \mathsf{Without}
\end{aligned}$$

Four rules have McPal coordinating migration by conducting Evol phase transfers of all other participants. In the first rule, McPal starts Scheduler migrating and, simultaneously, it transfers the three Workers to their new full dynamics. As long as Scheduler has not transferred their as-is CS phases to to-be ones, such new dynamics will remain excluded, although allowed by McPal already. The remaining three rules transfer Scheduler from migrating to round robin scheduling only, depending on whether $\mathsf{Worker_1}$ has arrived in whatever new CS phase; the other two Workers then must have arrived there too.

$$\begin{aligned}
&\mathsf{McPal\colon StartMigr} \stackrel{kickOff}{\rightarrow} \mathsf{StartRoRo} \ *\\
&\quad \mathsf{Scheduler(Evol)\colon Phase_1} \stackrel{triv}{\rightarrow} \mathsf{NDetToRoRo},\ \mathsf{Worker_1(Evol)\colon Phase_1} \stackrel{triv}{\rightarrow} \mathsf{Phase_2},\\
&\quad \mathsf{Worker_2(Evol)\colon Phase_1} \stackrel{triv}{\rightarrow} \mathsf{Phase_2},\ \mathsf{Worker_3(Evol)\colon Phase_1} \stackrel{triv}{\rightarrow} \mathsf{Phase_2}\\
&\mathsf{McPal\colon StartRoRo} \stackrel{phaseOut}{\rightarrow} \mathsf{Content} \ *\\
&\quad \mathsf{Scheduler(Evol)\colon NDetToRoRo} \stackrel{ready}{\rightarrow} \mathsf{Phase_2},\ \mathsf{Worker_1(Evol)\colon Without} \stackrel{triv}{\rightarrow} \mathsf{Without}\\
&\mathsf{McPal\colon StartRoRo} \stackrel{phaseOut}{\rightarrow} \mathsf{Content} \ *\\
&\quad \mathsf{Scheduler(Evol)\colon NDetToRoRo} \stackrel{ready}{\rightarrow} \mathsf{Phase_2},\ \mathsf{Worker_1(Evol)\colon Interrupt} \stackrel{triv}{\rightarrow} \mathsf{Interrupt}\\
&\mathsf{McPal\colon StartRoRo} \stackrel{phaseOut}{\rightarrow} \mathsf{Content} \ *\\
&\quad \mathsf{Scheduler(Evol)\colon NDetToRoRo} \stackrel{ready}{\rightarrow} \mathsf{Phase_2},\ \mathsf{Worker_1(Evol)\colon With} \stackrel{triv}{\rightarrow} \mathsf{With}
\end{aligned}$$

Please note, it is on the basis of the constraining character of phases and traps, the example of the critical section model with the above McPal succeeds in specifying a self-adapting model via migration coordination, even for unforeseen adaptation. No quiescence of components is necessary, as coordination remains ongoing although changing gradually. Once migration has finished, the original critical section model is working in a completely new way, but McPal, not unlike a catalyst, is still in place, having returned to its original appearance represented by its phase Hibernating only.

So far, we have established unforeseen adaptation without quiescence for our nondeterministic critical section example only, migrating to the above round robin solution. We abstract from the example as follows. For the general situation, let an arbitrary, well-defined Paradigm model $\mathsf{PM}_1$ be given. Assume, at some later point in time, we prefer to have another well-defined Paradigm model $\mathsf{PM}_2$ instead of $\mathsf{PM}_1$. Moreover assume, once $\mathsf{PM}_2$ is known, another well-defined Paradigm model $\mathsf{PM}_{1to2}$ can be constructed, specifying how to migrate as smoothly as required from $\mathsf{PM}_1$ performance to $\mathsf{PM}_2$ performance. Then, Paradigm model $\mathsf{PM}_1$ extended with $(i)$ the above McPal in hibernating form and with $(ii)$ trivially suitable Evol roles for each component, can coordinate its own migration to the originally unforeseen model $\mathsf{PM}_2$ performance, with McPal in place afresh, in hibernating form again and with new but similar trivially suitable Evol roles for each component. This means in particular, after whatever migration done in this manner, McPal is still in place for yet another unforeseen adaptation via yet another migration coordination done in this manner.

So, in view of modeling unforeseen change, the special component McPal is included in all Paradigm models. During the original, stable collaboration stage of the executing Paradigm model, McPal is stand-by only, not influencing the rest of the model at all. This is McPal 's hibernating form. But, by being there, McPal provides the means for preparing the migration as well as for conducting its coordination accordingly. To that aim, connections between McPal and the rest of the model are in place, realizing rudimentary interfacing for later purposes; in Paradigm terms, one Evol role per component *without* dynamics, as there is exactly one global state as *nonrestrictive* phase per Evol role. As soon as, via McPal, the new way of working as well as the migration towards it have been developed, McPal takes step giveOut thereby installing the relevant extension to the original model. Trap prepared of phase Hibernating having been entered by then, a choreography step is taken as Evol protocol step, only now enabling McPal to conduct the various Evol roles from their once stable $\mathsf{Phase}_m$, for some $m$ say, eventually to a next, until-further-notice stable $\mathsf{Phase}_{m+1}$. In this manner, McPal's own migration begins through choreography, the migration of the others is started thereafter by McPal as conductor taking a kickOff-like step. Finishing migration is done in reversed order. The others are explicitly left to their new stable collaboration, restricting them to their $\mathsf{Phase}_{m+1}$ phases, by McPal, as conductor, taking a phaseOut-like step, thus reaching a migrDone-like trap. Thereupon McPal ceases to influence the others, as a choreography step transfers role McPal(Evol) back to phase Hibernating. As a last step, the first within phase Hibernating, McPal shrinks the recently extended model, by removing model fragments no longer needed, keeping the new model only, McPal really emerging as a catalyst amidst a completely renewed model.

# 3 Process algebra translation of McPal

In this section, McPal and the other example components from Section 2 are expressed as PA processes, following the translation of [1]. Using the translation we prove formally that the system migrates indeed from the as-is to to-be behaviour. Moreover, the PA specification can directly be taken as an input into the mCRL2 modelchecker, to be used for further analysis of the migration model.

Recall that the system in migration originally has the as-is dynamics, to become the to-be behaviour once it has migrated. Thus, the Paradigm migration model comprises both, as-is and to-be behaviour, as well as the dynamicity of the migration, McPal included. As long as the system behaves as-is or to-be, McPal is in its hibernating from, and the behaviour of each component is constrained by its trivial Evol phase. Thus, the rich complex dynamics of the migrating system is restricted, by McPal and the Evol roles, to relatively more simple as-is behaviour (and similar for the to-be behaviour). We show that, indeed, the as-is behaviour, SysAsIs, is an abstracted version of the overall behaviour of the migrating system (Theorem 3.1). Moreover, the process algebraic compositional mechanisms allow us to take another perspective on the as-is behaviour. Namely, the as-is behaviour (same for to-be) can be considered as a standalone system, SWSysAsIs, not "connected" to any McPal and without Evol roles per component, and thus, not in the context of any migration, only as an isolated interacting composition of the relevant components. Nevertheless, we show that the presence of McPal and the Evol roles in the former SysAsIs as-is model does not add any behaviour. Namely we show, by establishing a relation between their PA specifications, that the two as-is models, SysAsIs as a part of the bigger migration model and SWSys as as-is system in isolation, essentially have the same behaviour (Theorem 3.2).

In the sequel, each STD from the Paradigm model is specified by a process algebraic recursive specification. Components are composed into larger systems by means of parallel composition and synchronization. Here we only specify the Scheduler and McPal STDs of the migration model, a Mig suffix in process names relating them to the migration model. Appendix A specifies the other migration STDs. Similarly suffixed, the AsIs and ToBe models are in Appendices B and C.

The recursive specifications of Scheduler's STDs of the migration model are given below. The migration Scheduler mimics both as-is and to-be Schedulers (see Fig. 6b). This is made explicit by naming the mimicking transitions *name*AsIs and *name*ToBe, respectively In addition, once conducted by McPal to NDetToRoRo –the right phase at the right time– Scheduler exhibits extended behaviour conducting other components towards their to-be behaviours. These transitions, typical for the migration model, have the extension Mig. Thus, the proceed transition is now represented by three different transitions: proceedAsIs, proceedMig and proceedToBe. This is essential, as each proceed action synchronizes differently in the three models. While in Paradigm this differentiation is implicit, in the process algebraic translation this has to be made clear. E.g., switch is a migration transition, hence it is denoted by switchMig. As described in [1], to capture vertical consistency, processes are augmented with the actions at?, at!, ok? and ok!. (Via the at communication, information whether a phase change can take place is passed from the local to the global level of a process; via the $\mathsf{ok}$ communication, information whether a local step is allowed by a current phase is exchanged.) Horizontal consistency is captured by the communication function '$|$' and process synchronization.

We introduce the following short-hand. For a component $\mathsf{C}$, we use $\mathsf{LAct(C)}$ to denote the set of all names of local transitions of that component. For instance, $\mathsf{LAct(Sch)} = \{\, \mathsf{grantAsIs}_1, \ldots, \mathsf{passToBe}_3\}$. $\mathsf{LAct(C){\downarrow}AsIs}$ denotes the subset of names in $\mathsf{LAct(C)}$ tagged as $\mathsf{AsIs}$ actions. Thus, $\mathsf{LAct(Sch){\downarrow}AsIs} = \{\mathsf{grantAsIs}_i,\ \mathsf{proceedAsIs}_i \mid i = 1, 2, 3\}$. Similar for other extensions, $\mathsf{ToBe}$ and $\mathsf{Mig}$. $\mathsf{Act(C)}$ denotes the set of all actions names in the process algebraic specification of $\mathsf{C}$.

The $\mathsf{Scheduler}$ of the migration model is specified as given below. Note, to emphasize that, via action $\mathsf{proceedAsIs}$, $\mathsf{Scheduler}$ conducts the $\mathsf{CS}$ roles of $\mathsf{Workers}$ (see consistency rules on page 5), we rather write $\mathsf{man(proceedAsIs)}$ instead of $\mathsf{ok?(proceedAsIs)}$. Similar for other cases of the $\mathsf{man}$ actions in the sequel.

$$
\begin{aligned}
\mathsf{SchedulerMig} &= \mathsf{IdleMig}\\
\mathsf{IdleMig} &= \sum_i \mathsf{man(grantAsIs}_i\mathsf{)} \cdot \mathsf{HelpingMig}_i + \mathsf{man(switchMig)} \cdot \mathsf{CheckingMig}_1\\
\mathsf{HelpingMig}_i &= \mathsf{man(proceedAsIs}_i\mathsf{)} \cdot \mathsf{Idle} + \mathsf{man(proceedToBe}_i\mathsf{)} \cdot \mathsf{CheckingMig}_{i+1} +\\
&\quad\ \mathsf{man(proceedMig}_i\mathsf{)} \cdot \mathsf{CheckingMig}_{i+1} + \mathsf{at!(HelpingMig}_i\mathsf{)} \cdot \mathsf{HelpingMig}_i\\
\mathsf{CheckingMig}_i &= \mathsf{man(grantToBe}_i\mathsf{)} \cdot \mathsf{HelpingMig}_i + \mathsf{man(passToBe}_i\mathsf{)} \cdot \mathsf{CheckingMig}_{i+1} +\\
&\quad\ \mathsf{at!(CheckingMig}_i\mathsf{)} \cdot \mathsf{CheckingMig}_i
\end{aligned}
$$

The specification of $\mathsf{Scheduler(Evol)}$ is

$$
\begin{aligned}
\mathsf{SchedulerEvolMig} &= \mathsf{SchEvolPhase1TrivMig}\\
\mathsf{SchEvolPhase1TrivMig} &= \sum_i \mathsf{ok!(grantAsIs}_i\mathsf{)} \cdot \mathsf{SchEvolPhase1TrivMig} +\\
&\quad\ \sum_i \mathsf{ok!(proceedAsIs}_i\mathsf{)} \cdot \mathsf{SchEvolPhase1TrivMig} +\\
&\quad\ \mathsf{emp(Phase1, NDetToRoRo, trivMig)} \cdot \mathsf{SchEvolNDetToRoRoTrivMig}\\[1ex]
\mathsf{SchEvolNDetToRoRoTrivMig} &= \sum_{t \in \mathsf{LAct(Sch)}} \mathsf{ok!}(t) \cdot \mathsf{SchEvolNDetToRoRoTrivMig} +\\
&\quad\ \sum_i \mathsf{at?(HelpingMig}_i\mathsf{)} \cdot \mathsf{SchEvolNDetToRoRoReadyMig} +\\
&\quad\ \sum_i \mathsf{at?(CheckingMig}_i\mathsf{)} \cdot \mathsf{SchEvolNDetToRoRoReadyMig}\\[1ex]
\mathsf{SchEvolNDetToRoRoReadyMig} &= \sum_{t \in \mathsf{LAct(Sch)} \setminus \mathsf{switchMig}} \mathsf{ok!}(t) \cdot \mathsf{SchEvolNDetToRoRoReadyMig} +\\
&\quad\ \mathsf{emp(NDetToRoRoReady, Phase2, readyMig)} \cdot \mathsf{SchEvolPhase2TrivMig}\\
\mathsf{SchEvolPhase2TrivMig} &= \sum_{t \in \mathsf{LAct(Sch)} \downarrow \mathsf{ToBe}} \mathsf{ok!}(t) \cdot \mathsf{SchEvolPhase2TrivMig}
\end{aligned}
$$

Translation of $\mathsf{McPal}$ and of $\mathsf{McPal(Evol)}$ is done similarly.

$$
\begin{aligned}
\mathsf{McPalMig} &= \mathsf{McPalObserving}\\
\mathsf{McPalObserving} &= \mathsf{ok?(wantChange)} \cdot \mathsf{McPalJITting}\\
\mathsf{McPalJITting} &= \mathsf{ok?(giveOut)} \cdot \mathsf{McPalStartMigr}\\
\mathsf{McPalStartMigr} &= \mathsf{ok?(kickOff)} \cdot \mathsf{McPalStartRoRo} + \mathsf{at!(McPalStartMigr)} \cdot \mathsf{McPalStartMigr}\\
\mathsf{McPalStartRoRo} &= \mathsf{ok?(phaseOut)} \cdot \mathsf{McPalContent}\\
\mathsf{McPalContent} &= \mathsf{ok?(cleanUp)} \cdot \mathsf{McPalObserving} + \mathsf{at!(McPalContent)} \cdot \mathsf{McPalContent}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{McPalEvolMig} &= \mathsf{McPalEvolHibTriv} \\
\mathsf{McPalEvolHibTriv} &= \textstyle\sum_{t \in \{\mathsf{wantChange}, \mathsf{giveOut}, \mathsf{cleanUp}\}} \mathsf{ok!}(t_i) \cdot \mathsf{McPalEvolHibTriv} + {} \\
&\quad \mathsf{at?}(\mathsf{StartMigr}) \cdot \mathsf{McPalEvolHibPrepared} \\
\mathsf{McPalEvolHibPrepared} &= \mathsf{emp}(\mathsf{Hib}, \mathsf{Phase2}, \mathsf{prepared}) \cdot \mathsf{McPalEvolToPhase2Triv} \\
\mathsf{McPalEvolToPhase2Triv} &= \mathsf{ok!}(\mathsf{kickOff}) \cdot \mathsf{McPalEvolToPhase2Triv} + {} \\
&\quad \mathsf{ok!}(\mathsf{phaseOut}) \cdot \mathsf{McPalEvolToPhase2Triv} + \mathsf{at?}(\mathsf{Content}) \cdot \mathsf{McPalEvolToPhase2MigDone} \\
\mathsf{McPalEvolToPhase2MigDone} &= \mathsf{emp}(\mathsf{Hib}, \mathsf{Phase2}, \mathsf{migrDone}) \cdot \mathsf{McPalEvolHibTriv}
\end{aligned}
$$

The communication function '$|$' is derived from the consistency rules. As for the translation in general, we put $\mathsf{at!}(s) \mid \mathsf{at?}(s) = \mathsf{at}(s)$ and $\mathsf{ok?}(t) \mid \mathsf{ok!}(t) = \mathsf{ok}(t)$. We present further synchronization in three parts, following the consistency rules. The first two communications pertain to the migration process exhibiting as-is behaviour (corresponding to the first two consistency rules on page 5).

$$
\begin{aligned}
\mathsf{grantAsIs}_i &= \mathsf{man}(\mathsf{grantAsIs}_i) \mid \mathsf{ok!}(\mathsf{grantAsIs}_i) \mid \mathsf{emp}(\mathsf{NotHaving}_i, \mathsf{Having}_i, \mathsf{requestAsIs}) \\
\mathsf{proceedAsIs}_i &= \mathsf{man}(\mathsf{proceedAsIs}_i) \mid \mathsf{ok!}(\mathsf{proceedAsIs}_i) \mid \mathsf{emp}(\mathsf{Having}_i, \mathsf{NotHaving}_i, \mathsf{doneAsIs})
\end{aligned}
$$

The next six clauses reflect what happens while Workers and Scheduler are migrating to their new behaviour. Note, migration of Workers from as-is to the to-be behaviour is clearly marked by moving from NotHaving and Having to Without, Interrupt or With (corresponding to the last six consistency rules on page 8).

$$
\begin{aligned}
\mathsf{switchMig} &= \mathsf{man}(\mathsf{switchMig}) \mid \mathsf{ok!}(\mathsf{switchMig}) \mid \mathsf{emp}(\mathsf{NotHaving}_1, \mathsf{Interrupt}_1, \mathsf{trivMig}) \mid \\
&\quad \mathsf{emp}(\mathsf{NotHaving}_2, \mathsf{Without}_2, \mathsf{trivMig}) \mid \mathsf{emp}(\mathsf{NotHaving}_3, \mathsf{Without}_3, \mathsf{trivMig}) \\
\mathsf{proceedMig}_i &= \mathsf{man}(\mathsf{proceedMig}_i) \mid \mathsf{ok!}(\mathsf{proceedMig}_i) \mid \mathsf{emp}(\mathsf{Having}_i, \mathsf{Without}_i, \mathsf{doneMig}) \mid \\
&\quad \mathsf{emp}(\mathsf{NotHaving}_{i+1}, \mathsf{Interrupt}_{i+1}, \mathsf{trivMig}) \mid \mathsf{emp}(\mathsf{NotHaving}_{i-1}, \mathsf{Without}_{i-1}, \mathsf{trivMig})
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{kickOff} &= \mathsf{man}(\mathsf{kickOff}) \mid \mathsf{emp}(\mathsf{Phase1}, \mathsf{NDetToRoRo}, \mathsf{triv}) \mid \\
&\quad \mathsf{emp}(\mathsf{Phase1}_1, \mathsf{Phase2}_1, \mathsf{triv}_1) \mid \mathsf{emp}(\mathsf{Phase1}_2, \mathsf{Phase2}_2, \mathsf{triv}_2) \mid \mathsf{emp}(\mathsf{Phase1}_3, \mathsf{Phase2}_3, \mathsf{triv}_3) \\
\mathsf{phaseOut}(\mathsf{Without}) &= \mathsf{man}(\mathsf{phaseOut}) \mid \mathsf{emp}(\mathsf{NDetToRoRo}, \mathsf{Phase2}, \mathsf{ready}) \mid \mathsf{emp}(\mathsf{Without}_1, \mathsf{Without}_1, \mathsf{triv}_1) \\
\mathsf{phaseOut}(\mathsf{Interrupt}) &= \mathsf{man}(\mathsf{phaseOut}) \mid \mathsf{emp}(\mathsf{NDetToRoRo}, \mathsf{Phase2}, \mathsf{ready}) \mid \mathsf{emp}(\mathsf{Interrupt}_1, \mathsf{Interrupt}_1, \mathsf{triv}_1) \\
\mathsf{phaseOut}(\mathsf{With}) &= \mathsf{man}(\mathsf{phaseOut}) \mid \mathsf{emp}(\mathsf{NDetToRoRo}, \mathsf{Phase2}, \mathsf{ready}) \mid \mathsf{emp}(\mathsf{With}_1, \mathsf{With}_1, \mathsf{triv}_1)
\end{aligned}
$$

The last clauses of the communication function capture the synchronization in the to-be behaviour (corresponding to the first three consistency rules on page 7).

$$
\begin{aligned}
\mathsf{grantToBe}_i &= \mathsf{man}(\mathsf{grantToBe}_i) \mid \mathsf{ok!}(\mathsf{grantToBe}_i) \mid \mathsf{emp}(\mathsf{Interrupt}_i, \mathsf{With}_i, \mathsf{requestToBe}) \\
\mathsf{proceedToBe}_i &= \mathsf{man}(\mathsf{proceedToBe}_i) \mid \mathsf{ok!}(\mathsf{proceedToBe}_i) \mid \\
&\quad \mathsf{emp}(\mathsf{With}_i, \mathsf{Without}_i, \mathsf{doneToBe}) \mid \mathsf{emp}(\mathsf{Without}_{i+1}, \mathsf{Interrupt}_{i+1}, \mathsf{trivToBe}) \\
\mathsf{passToBe}_i &= \mathsf{man}(\mathsf{passToBe}_i) \mid \mathsf{ok!}(\mathsf{passToBe}_i) \mid \\
&\quad \mathsf{emp}(\mathsf{Interrupt}_i, \mathsf{Without}_i, \mathsf{notYetToBe}) \mid \mathsf{emp}(\mathsf{Without}_{i+1}, \mathsf{Interrupt}_{i+1}, \mathsf{trivToBe})
\end{aligned}
$$

Combining the processes to express their collaboration requires parallel composition only. Thus, the whole migration process conducted by McPal is then specified by

$$\mathsf{SysMig} = \partial_{\mathsf{H}} \left( \mathsf{McPalMig} \parallel \mathsf{McPalEvolMig} \parallel \mathsf{SysMig}' \right)$$

$$\mathsf{SysMig}' = \parallel_i \left( \mathsf{WorkerMig}_i \parallel \mathsf{WorkerCSMig}_i \parallel \mathsf{WorkerEvolMig}_i \right) \parallel \mathsf{SchedulerMig} \parallel \mathsf{SchedulerEvolMig}$$

where the encapsulation operator $\partial_{\mathsf{H}}$ enforces all communicating actions to synchronize[3]. Similarly, using the translation of the proper STDs, we can derive both as-is behaviours of the service system: as-is in the presence of McPal in hibernation, SysAsIs, and as-is standalone service system, SWSysAsIs, of Workers and Scheduler. And similarly for the to-be behaviours. We thus define

$$\mathsf{SysAsIs} \;=\; \partial_{\mathsf{H}}\,(\,\mathsf{McPalAsIs} \parallel \mathsf{McPalEvolAsIs} \parallel \mathsf{SysAsIs}'\,)$$

$$\mathsf{SysAsIs}' \;=\; \parallel_i (\,\mathsf{WorkerAsIs}_i \parallel \mathsf{WorkerCSAsIs}_i \parallel \mathsf{WorkerEvolAsIs}_i\,) \parallel \mathsf{SchedulerAsIs} \parallel \mathsf{SchedulerEvolAsIs}$$

and

$$\mathsf{SysToBe} \;=\; \partial_{\mathsf{H}}\,(\,\mathsf{McPalToBe} \parallel \mathsf{McPalEvolToBe} \parallel \mathsf{SysToBe}'\,)$$

$$\mathsf{SysToBe}' \;=\; \parallel_i (\,\mathsf{WorkerToBe}_i \parallel \mathsf{WorkerCSToBe}_i \parallel \mathsf{WorkerEvolToBe}_i\,) \parallel \mathsf{SchedulerToBe} \parallel \mathsf{SchedulerEvolToBe}$$

For the standalone variants we have the following specifications:

$$\mathsf{SWSysAsIs} \;=\; \partial_{\mathsf{H}}\,(\parallel_i (\,\mathsf{WorkerAsIs}_i \parallel \mathsf{WorkerCSAsIs}_i\,) \parallel \mathsf{SchedulerAsIs}\,) \quad \text{and}$$

$$\mathsf{SWSysToBe} \;=\; \partial_{\mathsf{H}}\,(\parallel_i (\,\mathsf{WorkerToBe}_i \parallel \mathsf{WorkerCSToBe}_i\,) \parallel \mathsf{SchedulerToBe}\,)$$

Having formalized the separate components and the systems they compose, we are able to relate the models. See Theorem 3.1 to 3.3 below. The first result states, as long McPal(Evol) is not allowed to perform the choreography step prepared, meaning it cannot start migration, the larger migration system has the same behaviour as the as-is system, up to branching bisimulation [6,1]. This is specified, first by blocking action prepared and second, through abstraction from all actions McPal can perform in the Hibernating phase; thus all actions $\mathsf{Act}(\mathsf{McPalHib}) = \{\mathsf{ok}(\mathsf{wantChange}),\ \mathsf{ok}(\mathsf{giveOut}),\ \mathsf{ok}(\mathsf{cleanUp})\,\}$ are renamed into silent action $\tau$ by means of the abstraction operator $\tau_{\mathsf{Act}(\mathsf{McPalHib})}$. Note, blocking the phase change of McPal from Hibernating to ToPhase2, directly disables McPal to execute any action not allowed in the Hibernating phase. Thus, it is sufficient to abstract away only from Act(McPalHib) actions.

**Theorem 3.1** *SWSysAsIs is branching bisimilar to*

$$\tau_{\mathit{Act(McPalHib)}} \circ \partial_{\mathsf{H}}\,(\,\mathit{McPalMig} \parallel \partial_{\mathit{emp}(\_,\_,\mathit{prepared})}\,(\,\mathit{McPalEvolMig}\,) \parallel \mathit{SWSysMig}\,).$$

The second theorem states that both as-is models are equivalent, i.e. McPal in hibernation and the trivial Evol roles of Workers and Scheduler do not essentially change the as-is behaviour.

**Theorem 3.2** *SWSysAsIs is branching bisimilar to* $\tau_{\mathsf{Act(McPalHib)}}\,(\,\mathit{SysAsIs}\,)$.
*SWSysToBe is branching bisimilar to* $\tau_{\mathsf{Act(McPalHib)}}\,(\,\mathit{SysToBe}\,)$.

In view of the next theorem, a recursive specification is interpreted as a labelled transition system (LTS). Every state in an LTS corresponds to a process variable specified. The state space of the parallel composition is the product of the component state spaces, restricted to the subset of the states reachable from the initial state of the composition. In our example, every reachable state $\tilde{s}$ in the LTS of SWSysToBe is a tuple $(s_1, s_2, \ldots, s_7)$, where $s_i$ is a state in the LTS of the cor-

[3] For the sake of simplicity we assume that the set H is always properly chosen for the parallel composition of synchronizing processes to which the encapsulation operator is applied.

responding $i$-th component (2 states per $\mathsf{Worker}$, 1 state for $\mathsf{Scheduler}$). For $\tilde{s}$, we write $\mathsf{SWSysToBe}(\tilde{s})$. Let $S$ be the set of all (reachable) states of the LTS of $\mathsf{SWSysToBe}$. Let $S_i \subseteq S$, $i = 1, 2, 3$, contain all states in $S$ having currently $\mathsf{SchedulerToBe}$ in state $\mathsf{CheckingToBe}_i$, $\mathsf{WorkerCSToBe}_i$ in state $\mathsf{InterruptTrivToBe}_i$, and for $j \neq i$, $\mathsf{WorkerCSToBe}_j$ in state $\mathsf{WithoutTrivToBe}_j$. Finally, let $\mathsf{NoToBe}$, defined as $\mathsf{Act}(\mathsf{SysMig}) \setminus \mathsf{Act}(\mathsf{SysToBe})$, denote the set of actions from $\mathsf{SysMig}$ not in $\mathsf{SysToBe}$.

The last result states: Once the migration has been started, i.e. after McPal has executed the kickOff step, the migration process will evolve into to-be behaviour, independent of what the process was executing before. This is specified by hiding all NoToBe actions by means of abstraction, renaming them into $\tau$. The theorem implicitly confirms the progress of the migration process (conducted by McPal): eventually to-be behaviour is reached.

**Theorem 3.3** *Processes $\tau_G(\mathit{SysMig})$ is branching bisimilar to process*

$$\tau \cdot \mathit{kickOff} \cdot \big( \textstyle\sum_{\tilde{s} \in S_1} \tau \cdot \mathit{SWSysToBe}(\tilde{s}) + \sum_{i=1}^{3} \sum_{\tilde{s} \in S_i} \tau \cdot \mathit{SWSysToBe}(\tilde{s}) \big)$$

*where* $G = \mathit{NoToBe} \setminus \{\mathit{kickOff}\}$.

The four inner summands in the process description cover the four possible migration trajectories: the first one via switchMig migration step and the other three via $\mathsf{proceedMig}_i$, $i = 1, 2, 3$ migration steps. Intuitively, as long as kickOff is not executed, SysMig behaves as the as-is system (Theorem 3.1). Eventually kickOff is executed (under the fairness assumption), moving Scheduler into NDetToRoRo and Workers into Phase2. Between kickOff and either switchMig or $\mathsf{proceedMig}_i$, the system continues behaving as-is. However, in addition to behaving as-is, any reachable state in this phase can execute exactly one action out of switchMig and $\mathsf{proceedMig}_i$, $i = 1, 2, 3$. The current as-is state determines which is enabled. Thus, if in the current state $\tilde{s}$ of SysMig, SchedulerMig is in IdleMig state, namely $s_7 = \mathsf{IdleMig}$, then switchMig is enabled, but $\mathsf{proceedMig}_i$ is not. By execution of any of these four actions, the system is migrated to to-be behaviour. Essential is, these transitions change the global states of Workers only, not their detailed states. The three sets $S_i$ reflect this, each one containing states differing only in Workers' detailed states.

Migration as provided by the Paradigm migration model, which does not require any quiescence, is reflected in this theorem. Namely, components are continuously active, and the system can be in any allowed state at the moment the kickOff action is executed. As a result, the components are silently moved to their to-be behaviour essentially without changing their current local states, from where the system continues without any interruption to execute to-be transitions. This implies smooth migration, with ongoing component dynamics indeed.

The theorems presented above for the critical section running example can be generalized to any Paradigm migration model. As explained already in Section 2, a Paradigm migration model consists of three models, as-is model $\mathsf{PM}_1$, to-be model $\mathsf{PM}_2$, and migration model $\mathsf{PM}_{1to2}$. Note, due to the specific role of McPal in the migration process, its specification in hibernating form remains the same for any migration model, as well as for as-is and to-be models. Thus, in a similar manner as for the example above, McPal and its Evol role, McPalEvol, are components in the three models. Therefore, $\mathsf{PM}_{1to2} = \partial_{\mathsf{H}}(\,\mathsf{McPalMig} \parallel \mathsf{McPalEvolMig} \parallel \mathsf{PMMig})$ where $\mathsf{PMMig}$ is the composition of the other system components. Similar, $\mathsf{PM}_1 = \partial_{\mathsf{H}_1}(\,\mathsf{McPalAsIs} \parallel \mathsf{McPalEvolAsIs} \parallel \mathsf{PMAsIs}\,)$, and $\mathsf{PM}_2 = \partial_{\mathsf{H}_2}(\,\mathsf{McPalToBe} \parallel \mathsf{McPalEvolToBe} \parallel \mathsf{PMToBe}\,)$. Subsequently, the result of Theorem 3.1 can be generalized: $\tau_{\mathsf{Act(McPalHib)}}(\mathsf{PM}_1)$ is branching bisimilar to

$$\tau_{\mathsf{Act(McPalHib)}} \circ \partial_{\mathsf{H}}\ (\,\mathsf{McPalMig} \parallel \partial_{\mathsf{emp(\_,\_,prepared)}}\,(\,\mathsf{McPalEvolMig}\,) \parallel \mathsf{PMMig}\,).$$

where $\mathsf{H}$, as for $\mathsf{H}_1$ and $\mathsf{H}_2$ above, are properly chosen sets of actions to be forced to synchronize.

In general, in any Paradigm migration model, the migration of the system components is unleashed once McPal performs a kickOff-like action. Consequently, the components are silently moved to their to-be behaviour, possible via different trajectories. Assuming that there are $n$ different trajectories $t_k$, $k = 1, \ldots, n$. Assume that state $s_k$ is the first state on trajectory $t_k$ that is a state in the (LTS of the) to-be model $\mathsf{PM}_2$. And assume that the set $I$ contains all actions occurring in $\mathsf{PM}_{1to2}$ but not in $\mathsf{PM}_2$, except the kickOff-like action. Then the generalization of Theorem 3.3 states branching bisimilarity of $\tau_I(\mathsf{SysMig})$ and the process $\tau \cdot \mathsf{kickOff} \cdot \sum_{s_k} \tau \cdot \mathsf{PM}_2(s_k)$.

# 4 Variants, related and future work

The above McPal is reminiscent of two other McPal versions from earlier work [9,10]. Compared with [9], the above McPal is far more general, since the older one, lacking a McPal(Evol) role, has exactly two fixed migration steps between Crs extension and Crs reduction only. The older version does allow for quite some freedom in unforeseen migration, however, as both fixed migration steps can be adorned, lazily but just-in-time, with new conducting, even repeatedly so for later migrations. Nevertheless, more than two migration steps, alternative migration steps or iterated migration steps cannot be covered at once, which for the above McPal are no problem at all. The concrete migrations in [9] are also less comprising, more cautious than the above example combining change of all detailed and role dynamics within one migration cycle.

Compared with [10], the above McPal has a rather more elegant Hibernating phase: complete symmetry in initial model extension and final model reduction via actions giveOut and cleanUp, respectively. Moreover, the choreography steps coordinating McPal(Evol) steps are more simple than McPal's self-conducting in [10]. The actual migration in [10] is completely different, however, in two respects. A round robin strategy as above serves as as-is situation and the to-be situation is a pipeline architecture, with four Units collaborating pair-wise in producer-consumer fashion. So, Workers and Scheduler as above gradually become a Unit, with different dynamics each, without quiescence. Moreover, McPal decides on-the-fly of the migration, which Worker becomes which Unit. Another difference is, the migration is specified at a suitable architectural level: suggestively clear but incomplete, thus being not amenable to PA analysis yet.

In addition to variants for migration coordination, the Hibernating phase of McPal is open to variation too. Although such variants should not influence migration, being internal to Hibernating, they might unravel the preparation of the migration, by refining what could happen in state JITting. The following variant

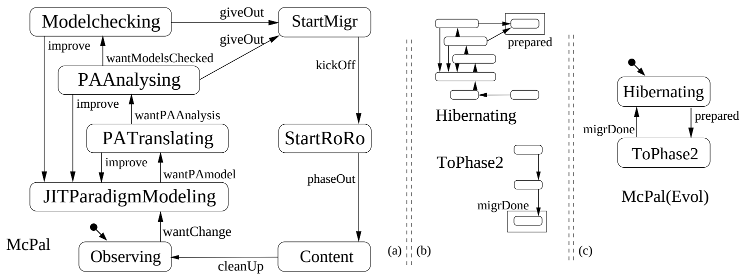


Fig. 10. Revisiting McPal: a new way of Hibernating.

particularly illustrates how translating a Paradigm model into a PA model and analyzing it, fit into McPal's life cycle, providing instant formal verification of a proposed migration trajectory. See Figure 10. State JITting is refined into cycling through four modeling-related states: from JITParadigmModeling to Modelchecking. At arrival in JITParadigmModeling the as-is Paradigm model is the only model known, specified as current value of Crs, comprising consistency rules and corresponding STD definitions. On leaving the state, generally two more Paradigm models are known, specified as current values of $\mathsf{Crs}_{toBe}$ and of $\mathsf{Crs} \cup \mathsf{Crs}_{migr} \cup \mathsf{Crs}_{toBe}$, respectively, allowing for analysis and improvement. Model analysis and checking can be abandoned via step giveOut from two states, always leading to StartMigr and thereby into trap prepared, only then enabling awakening from Hibernating.

The broad variability of McPal in its details, both for specifying concrete migration coordination and for unraveling migration support given by a concrete model engineering process, underlines McPal's *reusability* potential, effectively providing a pattern for adaptation.

There is much research addressing dynamic system adaptation. Generally, formal analysis of the migration trajectory is ignored. Exceptions to this are mainly found in the WCAT community. In the setting of component-based software engineering, process languages and mobile calculi are used to express run-time adaptor modification for coupled COTS components [3,4,5,15]. However, tool support towards formal analysis of run-time adaptation has not been addressed so far. Moreover, whereas adaptors do change, components cannot, unless by replacement: they are from on-the-shelf.

Various studies, e.g. [18,2,7,17], rely on high-level flexibility in an architectural setting, allowing low-level variability of components only. This boils down to rearranging existent or foreseen component behaviors. New behavior can only be achieved by replacing the existing component by a new version, requiring halting that component if not a larger part of the system. Even in case of adaptation at a detailed level and towards originally unforeseen behaviour, similar halting of the component to be adapted is generally required. Thus, actual adaptation is achieved by quiescence. In this manner it is not addressed how to adapt component behavior gradually, i.e., how to modify in detail ongoing behavior in an originally unforeseen direction, really on-the-fly. A wide perspective is discussed in [14], as yet without theory (or enough operational details) enabling formal trajectory analysis, but pointing out the relevance of five techniques: reflection, probes, decomposition, generation and reification. It is interesting to see these mirrored in Paradigm-McPal. Reflection is present through the consistency rules in Crs. Probes as feed forward and feedback stimuli are present through traps and phases. Generation is McPal's conducting, fully dynamically woven into dynamical decomposition (gradually fading out before phasing out) as well as into dynamical reification (gradually fading in after kick off): reification on-the-fly of decomposition constituting generation, conducted by McPal.

For future research topics we see great opportunities in investigating patterns for all kinds of dynamic change, by modeling and analyzing them in tandem. Such changes occur naturally where management, improvement or flexibility is relevant: reconfiguration, requirements change, alignment, etc. Concerning McPal as introduced here, we plan to study extensions concerning consistent creation and deletion of STDs, detailed and global, and also multiple McPals together, hierarchically organized or as a federation.

# References


[1] S. Andova, L.P.J. Groenewegen, and E.P. de Vink. Dynamic consistency in process algebra: From Paradigm to ACP. In P. Poizat C. Canal and M. Sirjani, editors, *Proc. FOCLASA'08*. 19pp. To appear, extended version submitted.

[2] O. Barais, A. Le Meur, L. Duchien, and J. Lawall. Software architecture evolution. In T. Mens and S. Demeyer, editors, *Software Evolution*, pages 233–262, 2008.

[3] A. Bracciali, A. Brogi, and C. Canal. A formal approach to component adaptation. *Journal of Systems and Software*, 74:45–54, 2005.

[4] A. Brogi, J. Cámera, C. Canal, J. Cubo, and E. Pimentel. Dynamic contextual adaptation. *ENTCS*, 175:81–95, 2007.

[5] J. Cámara, G. Saluaün, and C. Canal. Run-time composition and adaptation of mismatching behavioural transactions. In *Proc. SEFM, London*, pages 381–390. IEEE, 2007.

[6] R.J. van Glabbeek and P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43:555–600, 1996.

[7] I. Gorton, Yan Liu, and N. Trivedi. An extensible and leightweight architecture for adaptive server applications. *Software, Practice and Experience*, 38:853–883, 2008.

[8] L. Groenewegen, N. van Kampenhout, and E. de Vink. Delegation modeling with paradigm. In J.-M. Jacquet and G.P. Picco, editors, *Proc. COORDINATION 2005*, pages 94–108. LNCS 3454, 2005.

[9] L. Groenewegen and E. de Vink. Evolution on-the-fly with Paradigm. In P. Ciancarini and H. Wiklicky, editors, *Proc. COORDINATION 2006*, pages 97–112. LNCS 4038, 2006.

[10] L.P.J. Groenewegen and E.P. de Vink. Dynamic system adaptation by constraint orchestration. Technical Report CSR 08/29, Technische Universiteit Eindhoven, 2008. 20pp, arXiv:0811.3492v1.

[11] J.F. Jacob. *Domain Specific Modeling and Analysis*. PhD thesis, University of Leiden, 2008.

[12] J. Kramer and J. Magee. The evolving philosophers problem: dynamic change management. *IEEE Transactions on Software Engineering*, 16:1293–1306, 1990.

[13] J. Küster. *Consistency Management of Object-Oriented Behavioral Models*. PhD thesis, University of Paderborn, 2004.

[14] R. Morrison, D. Balasubramaniam, G. Kirby, K. Mickan, B. Warboys, R.M. Greenwood, I. Robertson, and B. Snowdon. A framework for supporting dynamic systems co-evolution. *Automated Software Engineering*, 14:261–292, 2007.

[15] P. Poizat and G. Salaün. Adaptation of open component-based systems. In M.M. Bonsangue and E.B. Johnsen, editors, *Proc. FMOODS 2007*, pages 141–156. LNCS 4468, 2007.

[16] A.W. Stam. *ParADE: A Conceptual Framework for Software Component Interaction*. PhD thesis, LIACS, Leiden University, 2009. Forthcoming.

[17] D. Sykes, W. Heaven, J. Magee, and J. Kramer. From goals to components: A combined approach to self-management. In *Proc. SEAMS'08, Leipzig*, pages 1–8. IEEE, 2008.

[18] J. Zhang and B.H.C. Cheng. Model-based development of dynamically adaptive software. In L.J. Osterweil, H.D. Rombach, and M.L. Soffa, editors, *Proc. ICSE'06*, pages 371–380. ACM, 2006.

# A Process algebraic specifications of other components in the migration model

The detailed behaviour of $\mathsf{Worker}_i$, $i = 1, 2, 3$, in the migration model:

$$
\begin{aligned}
\mathsf{WorkerMig}_i &= \mathsf{Free}_i \\
\mathsf{Free}_i &= \mathsf{at!}(\mathsf{Free}_i) \cdot \mathsf{Free}_i + \mathsf{ok?}(\mathsf{begin}_i) \cdot \mathsf{NonCrit}_i \\
\mathsf{NonCrit}_i &= \mathsf{ok?}(\mathsf{reserve}_i) \cdot \mathsf{Pre}_i \\
\mathsf{Pre}_i &= \mathsf{at!}(\mathsf{Pre}_i) \cdot \mathsf{Pre}_i + \mathsf{ok?}(\mathsf{pickUp}_i) \cdot \mathsf{Crit}_i \\
\mathsf{Crit}_i &= \mathsf{ok?}(\mathsf{layDown}_i) \cdot \mathsf{Post}_i \\
\mathsf{Post}_i &= \mathsf{ok?}(\mathsf{finish}_i) \cdot \mathsf{Free}_i + \mathsf{ok?}(\mathsf{continue}_i) \cdot \mathsf{NonCrit}_i + \mathsf{ok?}(\mathsf{hurry}_i) \cdot \mathsf{Pre}_i
\end{aligned}
$$

The roles $\mathsf{WorkerCS}_i$ and $\mathsf{WorkerEvol}_i$, $i = 1, 2, 3$, in the migration model:

$$
\begin{aligned}
\mathsf{WorkerCSMig}_i &= \mathsf{WorkerCSNotHavingTrivMig}_i \\
\mathsf{WorkerCSNotHavingTrivMig}_i &= \textstyle\sum_{t \in \{\mathsf{begin}, \mathsf{reserve}\}} \mathsf{ok!}(t_i) \cdot \mathsf{WorkerCSNotHavingTrivMig}_i + \\
&\quad \mathsf{at?}(\mathsf{PreMig}_i) \cdot \mathsf{WorkerCSNotHavingRequestMig}_i + \\
&\quad \mathsf{emp}(\mathsf{NotHaving}_i, \mathsf{Without}_i, \mathsf{trivMig}) \cdot \mathsf{WorkerCSWithoutTrivMig}_i + \\
&\quad \mathsf{emp}(\mathsf{NotHaving}_i, \mathsf{Interrupt}_i, \mathsf{trivMig}) \cdot \mathsf{WorkerCSInterruptTrivMig}_i \\
\mathsf{WorkerCSNotHavingRequestMig}_i &= \mathsf{emp}(\mathsf{NotHaving}_i, \mathsf{Having}_i, \mathsf{requestAsIs}) \cdot \mathsf{WorkerCSHavingMig}_i + \\
&\quad \mathsf{emp}(\mathsf{NotHaving}_i, \mathsf{Without}_i, \mathsf{trivMig}) \cdot \mathsf{WorkerCSWithoutTrivMig}_i + \\
&\quad \mathsf{emp}(\mathsf{NotHaving}_i, \mathsf{Interrupt}_i, \mathsf{trivMig}) \cdot \mathsf{WorkerCSInterruptTrivMig}_i \\
\mathsf{WorkerCSHavingTrivMig}_i &= \textstyle\sum_{t \in \{\mathsf{pickUp}, \mathsf{layDown}, \mathsf{finish}\}} \mathsf{ok!}(t_i) \cdot \mathsf{WorkerCSHavingTrivMig}_i + \\
&\quad \mathsf{at?}(\mathsf{FreeMig}_i) \cdot \mathsf{WorkerCSHavingDoneMig}_i \\
\mathsf{WorkerCSHavingDoneMig}_i &= \mathsf{emp}(\mathsf{Having}_i, \mathsf{NotHaving}_i, \mathsf{doneAsIs}) \cdot \mathsf{WorkerCSNotHavingMig}_i + \\
&\quad \mathsf{emp}(\mathsf{Having}_i, \mathsf{Without}_i, \mathsf{doneMig}) \cdot \mathsf{WorkerCSWithoutTrivMig}_i \\
\mathsf{WorkerCSWithoutTrivMig}_i &= \textstyle\sum_{t \in \{\mathsf{finish}, \mathsf{begin}, \mathsf{reserve}, \mathsf{continue}, \mathsf{hurry}\}} \mathsf{ok!}(t_i) \cdot \mathsf{WorkerCSWithoutTrivMig}_i + \\
&\quad \mathsf{emp}(\mathsf{Without}_i, \mathsf{Interrupt}_i, \mathsf{trivToBe}) \cdot \mathsf{WorkerCSInterruptTrivMig} + \\
&\quad \mathsf{emp}(\mathsf{Without}_i, \mathsf{Without}_i, \mathsf{trivMig}) \cdot \mathsf{WorkerCSWithoutTrivMig} \\
\mathsf{WorkerCSInterruptTrivMig}_i &= \textstyle\sum_{t \in \{\mathsf{finish}, \mathsf{begin}, \mathsf{continue}\}} \mathsf{ok!}(t_i) \cdot \mathsf{WorkerCSInterruptTrivMig}_i + \\
&\quad \textstyle\sum_{s \in \{\mathsf{Free}, \mathsf{NonCrit}, \mathsf{Post}\}} \mathsf{at?}(s_i) \cdot \mathsf{WorkerCSInterruptNotYetMig}_i + \\
&\quad \mathsf{at?}(\mathsf{PreMig}_i) \cdot \mathsf{WorkerCSInterruptRequestMig}_i + \\
&\quad \mathsf{emp}(\mathsf{Interrupt}_i, \mathsf{Interrupt}_i, \mathsf{trivMig}) \cdot \mathsf{WorkerCSInterruptTrivMig}_i \\
\mathsf{WorkerCSInterruptNotYetMig}_i &= \textstyle\sum_{t \in \{\mathsf{finish}, \mathsf{begin}, \mathsf{continue}\}} \mathsf{ok!}(t_i) \cdot \mathsf{WorkerCSInterruptNotYetMig}_i + \\
&\quad \mathsf{emp}(\mathsf{Interrupt}_i, \mathsf{Without}_i, \mathsf{notYetToBe}) \cdot \mathsf{WorkerCSWithoutTrivMig}_i + \\
&\quad \mathsf{emp}(\mathsf{Interrupt}_i, \mathsf{Interrupt}_i, \mathsf{trivMig}) \cdot \mathsf{WorkerCSInterruptTrivMig}_i \\
\mathsf{WorkerCSInterruptRequestMig}_i &= \mathsf{emp}(\mathsf{Interrupt}_i, \mathsf{With}_i, \mathsf{requestToBe}) \cdot \mathsf{WorkerCSWithTrivMig}_i + \\
&\quad \mathsf{emp}(\mathsf{Interrupt}_i, \mathsf{Interrupt}_i, \mathsf{trivMig}) \cdot \mathsf{WorkerCSInterruptTrivMig}_i \\
\mathsf{WorkerCSWithTrivMig}_i &= \textstyle\sum_{t \in \{\mathsf{finish}, \mathsf{begin}, \mathsf{continue}, \mathsf{pickUp}, \mathsf{layDown}\}} \mathsf{ok!}(t_i) \cdot \mathsf{WorkerCSWithTrivMig}_i + \\
&\quad \textstyle\sum_{s \in \{\mathsf{Free}, \mathsf{NonCrit}, \mathsf{Post}\}} \mathsf{at?}(s_i) \cdot \mathsf{WorkerCSWithDoneMig}_i + \\
&\quad \mathsf{emp}(\mathsf{With}_i, \mathsf{With}_i, \mathsf{trivMig}) \cdot \mathsf{WorkerCSWithTrivMig}_i
\end{aligned}
$$

$$\begin{aligned}
\mathsf{WorkerCSWithDoneMig}_i &= \textstyle\sum_{t\in\{\mathsf{finish},\mathsf{begin},\mathsf{continue}\}} \mathsf{ok!}(t_i)\cdot \mathsf{WorkerCSWithDoneMig}_i + \\
&\quad \mathsf{emp}(\mathsf{With}_i, \mathsf{Without}_i, \mathsf{doneToBe})\cdot \mathsf{WorkerCSWithoutTrivMig}_i + \\
&\quad \mathsf{emp}(\mathsf{With}_i, \mathsf{With}_i, \mathsf{trivMig})\cdot \mathsf{WorkerCSWithTrivMig}_i \\
\mathsf{WorkerEvolPhase1TrivMig}_i &= \textstyle\sum_{t\in\{\mathsf{finish},\mathsf{begin},\mathsf{reserve},\mathsf{pickUp},\mathsf{layDown}\}} \mathsf{ok!}(t_i)\cdot \mathsf{WorkerEvolPhase1TrivMig}_i + \\
&\quad \mathsf{emp}(\mathsf{Phase1}, \mathsf{Phase2}, \mathsf{trivMig})\cdot \mathsf{WorkerEvolPhase2TrivMig}_i \\
\mathsf{WorkerEvolPhase2TrivMig}_i &= \textstyle\sum_{t\in\mathsf{LAct}(\mathsf{WorkerMig}_i)} \mathsf{ok!}(t_i)\cdot \mathsf{WorkerEvolPhase2TrivMig}_i
\end{aligned}$$

Note, ternary synchronization $\mathsf{ok?}(t)|\mathsf{ok!}(t)|\mathsf{ok!}(t) = \mathsf{ok}(t)$ corresponds to vertical consistency of Workers.

# B Process algebraic specification of the as-is model

$\mathsf{Worker}_i$ and roles $\mathsf{WorkerCS}_i$, $\mathsf{WorkerEvol}_i$, $i = 1, 2, 3$, in the as-is model:

$$\begin{aligned}
\mathsf{WorkerAsIs}_i &= \mathsf{Free}_i \\
\mathsf{Free}_i &= \mathsf{at!}(\mathsf{Free}_i)\cdot \mathsf{Free}_i + \mathsf{ok?}(\mathsf{begin}_i)\cdot \mathsf{NonCritAsIs}_i \\
\mathsf{NonCrit}_i &= \mathsf{ok?}(\mathsf{reserve}_i)\cdot \mathsf{Pre}_i \\
\mathsf{Pre}_i &= \mathsf{at!}(\mathsf{Pre}_i)\cdot \mathsf{Pre}_i + \mathsf{ok?}(\mathsf{pickUp}_i)\cdot \mathsf{Crit}_i \\
\mathsf{Crit}_i &= \mathsf{ok?}(\mathsf{layDown}_i)\cdot \mathsf{Post}_i \\
\mathsf{Post}_i &= \mathsf{ok?}(\mathsf{finish}_i)\cdot \mathsf{Free}_i
\end{aligned}$$

$$\begin{aligned}
\mathsf{WorkerCSAsIs}_i &= \mathsf{WorkerCSNotHavingTrivAsIs}_i \\
\mathsf{WorkerCSNotHavingTrivAsIs}_i &= \textstyle\sum_{t\in\{\mathsf{begin},\mathsf{reserve}\}} \mathsf{ok!}(t_i)\cdot \mathsf{WorkerCSNotHavingTrivAsIs}_i + \\
&\quad \mathsf{at?}(\mathsf{PreAsIs}_i)\cdot \mathsf{WorkerCSNotHavingRequestAsIs}_i \\
\mathsf{WorkerCSNotHavingRequestAsIs}_i &= \mathsf{emp}(\mathsf{NotHaving}_i, \mathsf{Having}_i, \mathsf{requestAsIs})\cdot \mathsf{WorkerCSHavingAsIs}_i \\
\mathsf{WorkerCSHavingTrivAsIs}_i &= \textstyle\sum_{t\in\{\mathsf{pickUp},\mathsf{layDown},\mathsf{finish}\}} \mathsf{ok!}(t_i)\cdot \mathsf{WorkerCSHavingTrivAsIs}_i + \\
&\quad \mathsf{at?}(\mathsf{FreeAsIs}_i)\cdot \mathsf{WorkerCSHavingDoneAsIs}_i \\
\mathsf{WorkerCSHavingDoneAsIs}_i &= \mathsf{emp}(\mathsf{Having}_i, \mathsf{NotHaving}_i, \mathsf{doneAsIs})\cdot \mathsf{WorkerCSNotHavingAsIs}_i
\end{aligned}$$

$$\begin{aligned}
\mathsf{WorkerEvolAsIs}_i &= \mathsf{WorkerEvolPhase1TrivAsIs}_i \\
\mathsf{WorkerEvolPhase1TrivAsIs}_i &= \textstyle\sum_{t\in\mathsf{LAct}(\mathsf{WorkerAsIs_i})} \mathsf{ok!}(t_i)\cdot \mathsf{WorkerEvolPhase1TrivAsIs}_i
\end{aligned}$$

Scheduler and role SchedulerEvol in the as-is model:

$$\begin{aligned}
\mathsf{SchedulerAsIs} &= \mathsf{IdleAsIs} \\
\mathsf{IdleAsIs} &= \sum_i \mathsf{man}(\mathsf{grantAsIs}_i)\cdot \mathsf{HelpingAsIs}_i \\
\mathsf{HelpingAsIs}_i &= \mathsf{man}(\mathsf{proceedAsIs}_i)\cdot \mathsf{Idle}
\end{aligned}$$

$$\begin{aligned}
\mathsf{SchedulerEvolAsIs} &= \mathsf{SchEvolPhase1TrivAsIs} \\
\mathsf{SchEvolPhase1TrivAsIs} &= \sum_i \mathsf{ok!}(\mathsf{grantAsIs}_i)\cdot \mathsf{SchEvolPhase1TrivAsIs} + \\
&\quad \sum_i \mathsf{ok!}(\mathsf{proceedAsIs}_i)\cdot \mathsf{SchEvolPhase1TrivAsIs}
\end{aligned}$$

The above processes are composed in parallel to express the collaboration of Scheduler and Workers in the as-is model as a standalone process, as

$$\mathsf{SWSysAsIs} = \partial_{\mathsf{H}}\ (\|_i\ (\,\mathsf{WorkerAsIs}_i \parallel \mathsf{WorkerCSAsIs}_i\,) \parallel \mathsf{SchedulerAsIs}\,)$$

with the synchronization function defined by

$$\begin{aligned}\mathsf{grantAsIs}_i &= \mathsf{man}(\mathsf{grantAsIs}_i) \mid \mathsf{ok!}(\mathsf{grantAsIs}_i) \mid \mathsf{emp}(\mathsf{NotHaving}_i, \mathsf{Having}_i, \mathsf{requestAsIs}) \\ \mathsf{proceedAsIs}_i &= \mathsf{man}(\mathsf{proceedAsIs}_i) \mid \mathsf{ok!}(\mathsf{proceedAsIs}_i) \mid \mathsf{emp}(\mathsf{Having}_i, \mathsf{NotHaving}_i, \mathsf{doneAsIs})\end{aligned}$$

McPal can be also considered as a component in the as-is model, which is actually passive, as it is constrained by Evol phase Hibernating. Theorem 3.2 indeed states that McPal does not add to the behaviour of SWSysAsIs. To clarify this, we give the specification of McPal in the as-is model, where by $\delta$ we denote the deadlock process. It is clear from the Paradigm model of McPal that state StartMigr at the local level as well as trap prepared at the global role level induce deadlock.

$$\begin{aligned}\mathsf{McPalAsIs} &= \mathsf{McPalObserving} \\ \mathsf{McPalObserving} &= \mathsf{ok?}(\mathsf{wantChange}) \cdot \mathsf{McPalJITting} \\ \mathsf{McPalJITting} &= \mathsf{ok?}(\mathsf{giveOut}) \cdot \mathsf{McPalStartMigr} \\ \mathsf{McPalStartMigr} &= \delta \\ \mathsf{McPalContent} &= \mathsf{ok?}(\mathsf{cleanUp}) \cdot \mathsf{McPalObserving}\end{aligned}$$

$$\begin{aligned}\mathsf{McPalEvolAsIs} &= \mathsf{McPalEvolHibTriv} \\ \mathsf{McPalEvolHibTriv} &= \textstyle\sum_{t \in \{\mathsf{wantChange}, \mathsf{giveOut}, \mathsf{cleanUp}\}} \mathsf{ok!}(t_i) \cdot \mathsf{McPalEvolHibTriv} + \\ &\quad \mathsf{at?}(\mathsf{StartMigr}) \cdot \mathsf{McPalEvolHibPrepared} \\ \mathsf{McPalEvolHibPrepared} &= \delta\end{aligned}$$

The whole as-is process, including McPal and the Evol roles, is then specified by:

$$\mathsf{SysAsIs} = \partial_{\mathsf{H}} \left( \mathsf{McPalAsIs} \parallel \mathsf{McPalEvolAsIs} \parallel \mathsf{SysAsIs}' \right)$$

$$\mathsf{SysAsIs}' = \parallel_i \left( \mathsf{WorkerAsIs}_i \parallel \mathsf{WorkerCSAsIs}_i \parallel \mathsf{WorkerEvolAsIs}_i \right) \parallel \mathsf{SchedulerAsIs} \parallel \mathsf{SchedulerEvolAsIs}$$

where $\partial_{\mathsf{H}}$ enforces all communicating actions to synchronize. Note that McPal does not synchronize with any component, but executes local steps only, thus merely preparing for a next migration. So, its deadlocks at two levels are irrelevant for the other components.

# C Process algebraic specifications of to-be model

$\mathsf{Worker}_i$ and roles $\mathsf{WorkerCS}_i$, $\mathsf{WorkerEvol}_i$, $i = 1, 2, 3$, in the to-be model:

$$\begin{aligned}\mathsf{WorkerToBe}_i &= \mathsf{Free}_i \\ \mathsf{Free}_i &= \mathsf{at!}(\mathsf{Free}_i) \cdot \mathsf{Free}_i + \mathsf{ok?}(\mathsf{begin}_i) \cdot \mathsf{NonCrit}_i \\ \mathsf{NonCrit}_i &= \mathsf{at!}(\mathsf{NonCrit}_i) \cdot \mathsf{NonCrit}_i + \mathsf{ok?}(\mathsf{reserve}_i) \cdot \mathsf{Pre}_i \\ \mathsf{Pre}_i &= \mathsf{at!}(\mathsf{Pre}_i) \cdot \mathsf{Pre}_i + \mathsf{ok?}(\mathsf{pickUp}_i) \cdot \mathsf{Crit}_i \\ \mathsf{Crit}_i &= \mathsf{at!}(\mathsf{Crit}_i) \cdot \mathsf{Crit}_i + \mathsf{ok?}(\mathsf{layDown}_i) \cdot \mathsf{Post}_i \\ \mathsf{Post}_i &= \mathsf{at!}(\mathsf{Post}_i) \cdot \mathsf{Post}_i + \mathsf{ok?}(\mathsf{finish}_i) \cdot \mathsf{Free}_i + \\ &\quad \mathsf{ok?}(\mathsf{continue}_i) \cdot \mathsf{NonCrit}_i + \mathsf{ok?}(\mathsf{hurry}_i) \cdot \mathsf{Pre}_i\end{aligned}$$

$$
\begin{aligned}
\mathsf{WorkerCSToBe}_1 &= \mathsf{WorkerCSInterruptTrivToBe}_1\\
\mathsf{WorkerCSToBe}_2 &= \mathsf{WorkerCSWithoutTrivToBe}_2\\
\mathsf{WorkerCSToBe}_3 &= \mathsf{WorkerCSWithoutTrivToBe}_3\\
\mathsf{WorkerCSWithoutTrivToBe}_i &= \textstyle\sum_{t\in\{\mathsf{finish},\mathsf{begin},\mathsf{reserve},\mathsf{continue},\mathsf{hurry}\}} \mathsf{ok!}(t_i)\cdot \mathsf{WorkerCSWithoutTrivToBe}_i +\\
&\quad \mathsf{emp}(\mathsf{Without}_i,\mathsf{Interrupt}_i,\mathsf{trivToBe})\cdot \mathsf{WorkerCSInterruptTrivToBe}\\
\mathsf{WorkerCSInterruptTrivToBe}_i &= \textstyle\sum_{t\in\{\mathsf{finish},\mathsf{begin},\mathsf{continue}\}} \mathsf{ok!}(t_i)\cdot \mathsf{WorkerCSInterruptTrivToBe}_i +\\
&\quad \textstyle\sum_{s\in\{\mathsf{Free},\mathsf{NonCrit},\mathsf{Post}\}} \mathsf{at?}(s_i)\cdot \mathsf{WorkerCSInterruptNotYetToBe}_i +\\
&\quad \mathsf{at?}(\mathsf{PreToBe}_i)\cdot \mathsf{WorkerCSInterruptRequestToBe}_i\\
\mathsf{WorkerCSInterruptNotYetToBe}_i &= \textstyle\sum_{t\in\{\mathsf{finish},\mathsf{begin},\mathsf{continue}\}} \mathsf{ok!}(t_i)\cdot \mathsf{WorkerCSInterruptNotYetToBe}_i +\\
&\quad \mathsf{emp}(\mathsf{Interrupt}_i,\mathsf{Without}_i,\mathsf{notYetToBe})\cdot \mathsf{WorkerCSWithoutTrivToBe}_i\\
\mathsf{WorkerCSInterruptRequestToBe}_i &= \mathsf{emp}(\mathsf{Interrupt}_i,\mathsf{With}_i,\mathsf{requestToBe})\cdot \mathsf{WorkerCSWithTrivToBe}_i\\
\mathsf{WorkerCSWithTrivToBe}_i &= \textstyle\sum_{t\in\{\mathsf{finish},\mathsf{begin},\mathsf{continue},\mathsf{pickUp},\mathsf{layDown}\}} \mathsf{ok!}(t_i)\cdot \mathsf{WorkerCSWithTrivToBe}_i +\\
&\quad \textstyle\sum_{s\in\{\mathsf{Free},\mathsf{NonCrit},\mathsf{Post}\}} \mathsf{at?}(s_i)\cdot \mathsf{WorkerCSWithDoneToBe}_i\\
\mathsf{WorkerCSWithDoneToBe}_i &= \textstyle\sum_{t\in\{\mathsf{finish},\mathsf{begin},\mathsf{continue}\}} \mathsf{ok!}(t_i)\cdot \mathsf{WorkerCSWithDoneToBe}_i +\\
&\quad \mathsf{emp}(\mathsf{With}_i,\mathsf{Without}_i,\mathsf{doneToBe})\cdot \mathsf{WorkerCSWithoutTrivToBe}_i
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{WorkerEvolToBe}_i &= \mathsf{WorkerEvolPhase2TrivToBe}_i\\
\mathsf{WorkerEvolPhase2TrivToBe}_i &= \textstyle\sum_{t\in\mathsf{LAct}(\mathsf{WorkerToBe_i})} \mathsf{ok!}(t_i)\cdot \mathsf{WorkerEvolPhase2TrivToBe}_i
\end{aligned}
$$

Scheduler and role SchedulerEvol in the to-be model:

$$
\begin{aligned}
\mathsf{SchedulerToBe} &= \mathsf{Checking_1ToBe}\\
\mathsf{HelpingToBe}_i &= \mathsf{man}(\mathsf{proceedToBe}_i)\cdot \mathsf{CheckingToBe}_{i+1}\\
\mathsf{CheckingToBe}_i &= \mathsf{man}(\mathsf{grantToBe}_i)\cdot \mathsf{HelpingToBe}_i + \mathsf{man}(\mathsf{passToBe}_i)\cdot \mathsf{CheckingToBe}_{i+1}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{SchedulerEvolToBe} &= \mathsf{SchEvolPhase2TrivToBe}\\
\mathsf{SchEvolPhase2TrivToBe} &= \sum_{t\in\mathsf{LAct}(\mathsf{SchToBe})} \mathsf{ok!}(t)\cdot \mathsf{SchEvolPhase2TrivToBe}
\end{aligned}
$$

To express the collaboration of Scheduler and Workers in the to-be model in isolation, the relevant specifications from above are composed in parallel:

$$
\mathsf{SWSysToBe} = \partial_{\mathsf{H}}\ (\|_i\ (\,\mathsf{WorkerToBe}_i \parallel \mathsf{WorkerCSToBe}_i\,) \parallel \mathsf{SchedulerToBe}\,)
$$

with the synchronization defined as

$$
\begin{aligned}
\mathsf{grantToBe}_i &= \mathsf{man}(\mathsf{grantToBe}_i) \mid \mathsf{ok!}(\mathsf{grantToBe}_i) \mid \mathsf{emp}(\mathsf{Interrupt}_i,\mathsf{With}_i,\mathsf{requestToBe})\\
\mathsf{proceedToBe}_i &= \mathsf{man}(\mathsf{proceedToBe}_i) \mid \mathsf{ok!}(\mathsf{proceedToBe}_i) \mid\\
&\quad \mathsf{emp}(\mathsf{With}_i,\mathsf{Without}_i,\mathsf{doneToBe}) \mid \mathsf{emp}(\mathsf{Without}_{i+1},\mathsf{Interrupt}_{i+1},\mathsf{trivToBe})\\
\mathsf{passToBe}_i &= \mathsf{man}(\mathsf{passToBe}_i) \mid \mathsf{ok!}(\mathsf{passToBe}_i) \mid\\
&\quad \mathsf{emp}(\mathsf{Interrupt}_i,\mathsf{Without}_i,\mathsf{notYetToBe}) \mid \mathsf{emp}(\mathsf{Without}_{i+1},\mathsf{Interrupt}_{i+1},\mathsf{trivToBe})
\end{aligned}
$$

Again note, similar to the as-is model, McPal does not communicate with other components, which can be concluded from the definition of synchronization given above. Indeed, once the migration is done and the to-be behaviour is reached, running as it should, McPal returns to Hibernating and remains under this constraint till the next migration. Thus, $\mathsf{McPalToBe}$ is defined by

$$\begin{aligned}
\mathsf{McPalToBe} &= \mathsf{McPalContent} \\
\mathsf{McPalObserving} &= \mathsf{ok?(wantChange)} \cdot \mathsf{McPalJITting} \\
\mathsf{McPalJITting} &= \mathsf{ok?(giveOut)} \cdot \mathsf{McPalStartMigr} \\
\mathsf{McPalStartMigr} &= \delta \\
\mathsf{McPalContent} &= \mathsf{ok?(cleanUp)} \cdot \mathsf{McPalObserving}
\end{aligned}$$

and incorporated as a component of the to-be behaviour of process $\mathsf{SysToBe}$ specified as

$$\mathsf{SysToBe} \;=\; \partial_{\mathsf{H}}\ (\,\mathsf{McPalToBe} \parallel \mathsf{McPalEvolToBe} \parallel \mathsf{SysToBe}'\,)$$

$$\mathsf{SysToBe}' \;=\; \|_i\ (\,\mathsf{WorkerToBe}_i \parallel \mathsf{WorkerCSToBe}_i \parallel \mathsf{WorkerEvolToBe}_i\,) \parallel\ \mathsf{SchedulerToBe} \parallel \mathsf{SchedulerEvolToBe}$$

but it does not add any relevant behaviour. As usual $\partial_{\mathsf{H}}$ blocks all not synchronized communicating actions. The specification of $\mathsf{McPalEvolToBe}$ is the same as the specification of $\mathsf{McPalEvolAsIs}$. Given the two specifications of $\mathsf{McPalToBe}$ and $\mathsf{McPalEvolToBe}$, we can show that after renaming all $\mathsf{McPal}$ actions, $\mathsf{Act}(\mathsf{McPalToBe})$, into $\tau$, $\mathsf{SysToBe}$ becomes branching bisimilar to $\mathsf{SWSysToBe}$: $\tau_{\mathsf{Act(McPalToBe)}}(\mathsf{SysToBe})$ is branching bisimilar to $\mathsf{SWSysToBe}$ (Theorem 3.2).