Dynamic Adaptation with Distributed Control in Paradigm

S. Andova[a], L.P.J. Groenewegen[b], E.P. de Vink[c,d,*]

[a] *ICT–Technology, Fontys University of Applied Sciences, Eindhoven, The Netherlands*
[b] *Leiden Institute of Advanced Computer Science, Leiden University, The Netherlands*
[c] *Department of Mathematics and Computer Science, Eindhoven University of Technology, The Netherlands*
[d] *Centrum Wiskunde & Informatica, Amsterdam, The Netherlands*

Abstract

Adaptation of a component-based system can be achieved in the coordination modeling language Paradigm through the special component McPal. McPal regulates the propagation of new behaviour and guides the changes in the components and in their coordination. Here we show how McPal may delegate part of its control to local adaptation managers, created on-the-fly, allowing for distribution of the adaptation indeed. We illustrate the approach for the well-known example of the dining philosophers problem, by modeling migration from a deadlock-prone solution to a deadlock-free and starvation-free solution without any system quiescence. The system migration goes through various stages, exhibiting a shift of control among McPal and its helpers, and changing degrees of orchestrated and choreographic collaboration. The distributed system adaptation is formally verified using the `mCRL2` model checker.

Key words: component-based systems, dynamic system adaptation, distributed control, formal verification

1. Introduction

Many systems today are affected by changes in their operational environment when running, while they cannot be shutdown to be updated and restarted again. Instead, *dynamic adaptive systems* must be able to change their behaviour on-the-fly and to self-manage adaptation steps accommodating a new policy.

Dynamic adaptive systems consist of interacting components, usually distributed, and possibly hierarchically organized. In such a system, components may start adaptation in response to various triggers, such as changes in the underlying execution environment (e.g. failures or network congestion) or changes of requirements (e.g. imposed by the user). Adaptation of one component in the system may inadvertently influence the behavior of the components it is interacting with, possibly bringing about a cascade of dynamic changes in other parts of the system. Therefore, the adaptation of a distributed system is a combination of local changes per component and global adaptation across components and hosts in the system. As such, adaptation has to be performed in a consistent and coordinated manner so that the functionality of each separate component and of the system as a whole are preserved while the adaptation is in progress. Due to the complexity of the distributed dynamics of a system adapting on-the-fly, it may be rather difficult to understand whether a realization of a change plan indeed allows the system to perform as it is supposed to, and does not violate any of its requirements, during and after system adaptation.

One way to circumvent this is to formally model and analyze the system behaviour and the adaptation changes to be followed. In [28, 5, 8] we advocated how orchestrated adaptation can conveniently be captured in the coordination modeling language Paradigm (backed up by a translation into the specification languages of the model checkers `mCRL2` and `Prism`). In Paradigm, a system architecture is organized along specific collaboration dimensions, called partitions. A partition is a well-chosen set of sub-behaviours of the local behaviour of a component, specifying the phases the component goes through when taking part in the collaboration. At a higher layer in the architecture, the component participates via its role, an abstract representation of the phases. A protocol determines the phase transfers for the components involved. In fact, in Paradigm, dynamic adaptation is modeled as just another collaboration, coordinated

*Corresponding author, email `e.p.d.vink@tue.nl`.

by a special component *McPal* [28, 5, 8]. As progress within a phase is completely local to the component, the use of phase transfer instead of state transfer, is the key concept of Paradigm. This makes it possible to model, at the same time and separated from one another, both behavioural local changes per component, and global changes across architectural layers.

Within Paradigm, the possibility of splitting up coordination and cooperation between components within a system architecture into collaborations, each following its own protocol, has useful consequences for Paradigm's applicability and scalability. In a context without adaptation, Paradigm models have been developed and successfully applied for quite large coordination situations [43, 39, 45, 46]. In [43] the latest UNIX version publicly available at that time, is remodeled and studied for a multi-processor platform. In [39] the CRIS-case from Information Systems is modeled and studied as well as some elevator systems. In [45, 46] different architectures of hospital information systems are modeled and investigated. The modeling invariably is in Paradigm, the investigations are through arguing and simulation, as at that time a connection with formal verification was not established yet. These and other examples underpin both applicability and scalability of Paradigm modeling, based on clearly separated collaborations. All this is without formal verification as well as without adaptation. Main lessons learned are, Paradigm models for large real world systems can be developed; such large systems may range from purely technical ICT (software) to purely human (business).

From 2006 on, we have published about *McPal* [28, 5, 8]: how a Paradigm model, through a special component called *McPal*, can adapt itself into a new Paradigm model, on-the-fly of the model's ongoing dynamicity. In [28] a rather rigid *McPal* is used for two consecutive adaptations: from a not so efficient non-deterministic solution for three components competing for a critical section, via a more efficient non-deterministic solution (with a "smaller" critical section), finally to a deterministic round robin solution (for the smaller critical section). In [5] the *McPal* component is rather more flexible as it can adapt itself too. Moreover, the migration trajectories are formally verified. The migration example is similar to the [28] example, but the migration is more direct: from the not so efficient non-deterministic solution in one migration to the more efficient deterministic round robin solution. Model checking is done with `mCRL2`. Some variants of *McPal* are discussed too. In [8] the same *McPal* is used to migrate from a deterministic round robin solution for a 4-party critical section problem to a probabilistic solution. Here *McPal* is achieving this via gradually adding more probabilistic control into the solution at the cost of the original round robin character. Model checking is done via Prism instead of via `mCRL2`. From 2008 on, papers [7, 9, 5, 8, 10] explain how to translate a Paradigm model into the `mCRL2` language. Main lessons learned are, for Paradigm models self-adaptation is a lazy or just-in-time form of coordination in Paradigm style. Moreover, as Paradigm models can be translated to process algebra, model checking of Paradigm models can be carried out. In particular, migration trajectories arising in self-adapting Paradigm models can be formally analyzed through model checking.

Both with respect to migration and with respect to model checking, applicability and scalability are substantially lagging behind the applicability and scalability of modeling in Paradigm without adaptation and without additional model checking. For Paradigm modeling with model checking we have made some progress in scalability in [10], as roles and the clear separation of collaborations at a higher level do facilitate reduction in the state space to be model checked. For the adaptation we have only a first beginning of understanding the model suites (sequences of consecutive Paradigm models appearing and disappearing one after another) during the migration trajectories that can be realized by the adaptation.

The split-up in collaborations adds to the scalability of the approach: protocols, capturing component interaction, that are orthogonal to each other can be addressed more focusedly, namely with respect to the specific roles of the components involved. By focusing on one specific collaboration at a time only part of the detailed behaviour of the component is needed explicitly, which should lead to smaller state spaces to be built. Further work is needed to assess the potential benefits, in particular when components are intertwined in many different ways. As in the remaining sections of this paper delegation of migration coordination is structured according to clearly separated collaborations, we have the impression this is useful for scalability and so for applicability of the *McPal* approach, as it is in line with the reasons underpinning Paradigm's applicability and scalability for normal coordination, see [43, 39, 45, 46] and the above remarks concerning these papers.

The suitability of Paradigm to model distributed adaptation strategies, extending our earlier centralized adaptation studies, is discussed in this paper on the dining philosophers example (see also [4]). A deadlock-prone solution of the dining philosophers problem is taken as a source system, to be migrated to a target solution, both deadlock-free and starvation-free. Both systems are modeled in Paradigm, as is the migrating from source to target. As typical for

dynamic adaptation in Paradigm, *McPal* regulates the propagation of new behaviour and guides the structural changes in the components and in their coordination. Seamlessly, parallel to ongoing execution, new behaviour is woven into the ongoing behaviour, such that smooth migration is established from *as-is* behaviour (of the source system) via intermediate behaviour to *to-be* behaviour (of the target system). Note, both intermediate and *to-be* behaviours were originally unknown. All this is characteristic for *McPal*. But here, although adding to the complexity of the solution, *McPal* delegates part of its control to local adaptation managers $McPhil_i$, $i = 1..5$, one for each philosopher, while *McPal* keeps controlling them globally. Thus, we argue, the component-based character of the Paradigm language allows for modeling distributed adaptation: separate modeling of strategies for changes of local behaviour, coordinated by *McPal* as system adaptation manager guiding the architectural changes. This reveals the distributed potential of system adaptation within Paradigm. So far, the paper repeats the results from [4]. The extension in this journal version particularly lies in showing how this form of distributed migration can be verified.

More in particular, the main new contribution of this paper is twofold. Firstly, since a Paradigm model describes the collaboration architecture of a system, we show how this feature can be used to extract an abstract view on the migration of the coordination. Such an architectural view, accurately detailed in terms of subsequent architectural snapshots, eases the understanding of the different and various trajectories the whole system may follow during migration, and it also gives an additional insight how the migration of a distributed system (such as the dining philosophers) can be formally represented and structured in a static language as `mCRL2`. Secondly, in this paper we provide a proof of the correctness of the migration, where the conference contribution [4] lacks formal verification of the migration altogether. Here we elaborate on an explicit correspondence between the subsequent architectural snapshots of the system's collaboration structure mentioned above and certain relevant subsequent stages occurring during the distributed migration coordination as we model it. It is based on this correspondence we are able to organize the formal verification of the migration through model checking, thus avoiding state space explosion effectively.

The organization of the formal verification as done here, actually extends the process algebra translations from our earlier work. As before, we specify the Paradigm model in the process modeling language `mCRL2` (cf. [30, 29]) and we exploit the `mCRL2` toolset to analyze the migration. Basically we follow the translation of Paradigm into `mCRL2` as introduced in [7, 9], but benefiting from the architectural view on the migration, we are able to give more structure to this translation. This happens to be more intuitive and it clearly reflects Paradigm's concepts and modeling potential. The main challenge regarding the formal `mCRL2` representation here is to encode the dynamics of the migration, nicely captured in Paradigm, in the static `mCRL2` language. We do this via so-called rule processes, that specify which consistency rules can be executed at a specific moment. A state change of a rule process leads to a change of the eligibility of consistency rules and hence leads to a change in the dynamics of the system.

We check that the starting *as-is* situation of the dining philosophers model admits deadlock, while the final *to-be* situation, to be reached after migration has been completed, is indeed both deadlock-free and starvation-free. However, more interestingly, we formally verify that always the *to-be* configuration will be eventually reached, independent of the specific distribution of the migration coordination among *McPal* and the $McPhil_i$. Also we argue that the migration is seamless, the behaviour of the philosophers themselves isn't changed at all during the migration and requests to eat can be granted also at a later stage of the system evolution. Thus, the overall adaptation is transparent to the philosopher components.

Related work. In recent years a number of approaches has been proposed addressing several issues of dynamic system adaptation. Some of them e.g. [32, 15, 38, 23], focus on adaptive software architectures, where functionalities, considered as black boxes, are connected via ports. The actual adaptation is mainly achieved by reorganizing the architecture. In [26] dynamic structural, possibly hierarchical, adaptation of freely interactive software agents is addressed. The structural adaptation is defined by coordination patterns, represented by graph transformations. In contrast to our work, the local functionality of components remains intact during adaptation. In [40] alternative implementations of software modules are allowed to support dynamic reconfiguration when a fault is detected. In the framework proposed there fault tolerance policies are defined as a part of the architecture of distributed embedded system, and in that sense the changes are architectural only. A system manager is responsible for collecting status information, and computing and distributing instructions regarding the reconfiguration to the software modules. Although these approaches are concerned with the dynamic adaptation of distributed system, they consider only the structural changes. Thus, they are closely related to the architectural view at the Paradigm adaptation model (cf. Section 6), which is only one aspect of dynamic adaptation that Paradigm can describe. The adaptation mechanism of [40] is not distributed in the way

as we treat it in this paper. In our case, the local adaptation managers $McPhil_i$ do not only execute the instructions from the central manager *McPal* but they are also responsible for local behavioural adaptation which is not influenced by *McPal*. Graph transformations for reconfiguration of component-based systems is addressed in [31, 34] in the context of the Reo coordination language. There, connectors that coordinate the interaction of a number of components alternate establishing flow and possible adaptation. So, although a form of quiescence is necessary for the connector to evolve, meanwhile activity of the components is allowed.

Formal modeling of dynamic adaptation has been addressed in e.g. [36, 3, 20, 26, 48, 1]. However, none of these approaches deal with distribution explicitly. In [22, 24] dynamic adaptation is formally modeled by means of graph transformation. Although graph transformation techniques are well suited for distributed systems, there is no explicit focus on modeling distributed control for adaptation in the papers mentioned. A framework for formal modeling and verification of dynamic adaptation of distributed system, based on a transitional-invariant lattice technique, is proposed in [17]. The approach uses theorem proving techniques to show that during and after adaptation, the system always satisfies the transitional-invariants. This adaptation framework, however, does not support distribution in the style discussed in this paper: distributing adaptation tasks among local adaptation conductors by delegation.

Some aspects of dynamic adaptation of distributed systems, tailored for the domains considered, have been treated in [2, 37]. In the domain of Web Services, [2] proposes a method to generate distributed adapters from given service descriptions. [37] focuses on modeling and deployment of distributed resources for adaptive services in a mobile environment. Tuosto et al. [18, 14, 19] advocate the use of contracts in distributed setting, in particular addressing service-oriented computing involving multiple parties and multiple sessions. In this set-up contracts-as-formulas enable to split up the interaction in stages distinguishing between global assertions and their local projections.

The Conductor framework [47] for distributed adaptation allows for dynamic deployment of multiple adaptation conductors at various points in a network, an approach which is more suitable for complex and heterogeneous collaborations. It includes a distributed planning algorithm which determines for a triggered adaptation the most appropriate combination of conductors, distributed across the network. In [41] a distributed adaptation model for component-based applications is proposed. The model consists of two types of functionalities: mandatory that manage basic adaptation operations and optional that can be used to distribute adaptation activities. This way the adaptation mechanism of the whole system can be hierarchically organized, resembling as such our hierarchical structures of *McPal* conductors. However, both in [41] and [47], the main focus is on designing the adaptation itself, while the formal modeling and analysis of the adaptation remain uncovered, positioning them complementary to our treatment of distributed adaptation.

Structure of the paper. Section 2 is an overview of Paradigm through the example of the deadlock-prone solution as source system. The target system, deadlock and starvation free, is described in Section 3. An analysis of the separate source and target system using `mCRL2` is given in this section too. Section 4 gives the distributed migration set-up from source to target system, with Section 5 discussing coordination technicalities separately. Section 6 explains from an architectural point of view the distribution of the coordination across the various stages of the migration. Formal verification of the migration itself is addressed in Section 7. Section 8 wraps up and provides conclusions. Compared to [4], sections 6 and 7, and parts of 1, 2 and 3 are new.

2. Dining philosophers *as-is*: deadlock-prone

This section presents a first solution to the dining philosophers problem: $Phil_i$ components share $Fork_i$ components, where $i = 1..5$. We shall refer to this solution as the *as-is* system or just *as-is*. As an extra requirement, the system has the ability to migrate from its ongoing *as-is* behaviour to *to-be* behaviour. Thus, *as-is* behaviour is the source behaviour in the adaptation of the dining philosophers. But the migration in the *as-is* system, as presented in this section, is not triggered yet. However, in view of the migration to come (cf. Section 4), the special component *McPal* in the Paradigm model of the *as-is* system is hibernatingly present to guide, once triggered, upcoming system migration, but at first not influencing the *as-is* behaviour at all.

The *as-is* solution itself is the well-known and failing deadlock-prone solution: Any $Phil_i$, while thinking and getting hungry, first waits until the left $Fork_i$ can be got, then gets it, waits until the right $Fork_{i+1}$ can be got, gets it and once having both forks starts eating. After eating, having satisfied her hunger, $Phil_i$ lays down both forks and

returns to thinking again. Apparently, steps taken by the *Phil*s and step-like status changes of the *Fork*s are to be consistently aligned to model precisely the particular *as-is* solution. This means, behaviour of the five *Phil*s and *Fork*s has to be coordinated such that the *as-is* system is realized, failing as it may. In this section we show how this is achieved in Paradigm by means of five collaborations, one collaboration per philosopher and the two forks assigned to her. Through the example of the *as-is* system for the five *Phil* and *Fork* components with a hibernating *McPal* in place, we shall illustrate Paradigm. But first we briefly introduce Paradigm and the problem situations addressed by it.

The coordination modeling language Paradigm addresses coordination of interacting components. Paradigm does so in terms of exogenous coordination: to be specified outside and between the components involved. UML 2.0's structural diagram coming closest to exogenous coordination is the collaboration diagram, a special composite structure diagram with its own representation. Please note, a UML 2.0 collaboration diagram is different from a UML 1.4 collaboration diagram (the latter being a diagram of UML 2.0 too, renamed as communication diagram), as it does not specify any interaction behaviour. So the UML 2.0 collaboration diagram has no sends and receives. In subfigure 1a we use such a collaboration diagram to visualize coordination issues of interacting components. At the bottom one recognizes the components, here called A, . . . , K. Such components can also be objects or even non-software elements like business departments or human actors. In view of interacting with each other, the components participate in a particular collaboration (dashed oval), to which they send their contribution and from which they receive influence. For its sending and receiving a component has a port: the small square on its border. The port itself is connected to a place-holder within the collaboration where the contribution of the component plays its role in the ongoing interaction and from where influence is received. Links between the place-holder roles have been omitted as their precise form is less relevant here.

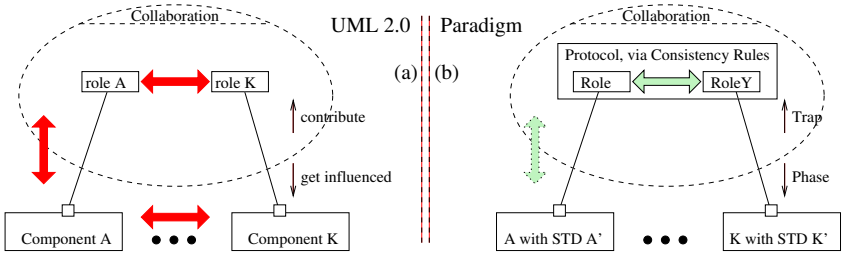

Figure 1: Collaborations: (a) dynamic consistency problems, (b) Paradigm's approach.

A serious issue with such collaboration diagrams (and for that matter with all UML 2.0's behavioural diagrams, like communication diagrams, sequence diagrams, interaction overview diagrams) is behavioural consistency, both horizontal and vertical, see [35, 25]. Horizontal is between components or between ports or between roles; vertical is between a component and its port, and between a port or component and the place-holder role. Moreover, as UML does not define the place-holder roles at all, this underlines its lacking in consistency even further. In the drawing the consistency problems are indicated with double-headed thick dark arrows; the thin black arrows suggest the sending direction of what is being contributed to a collaboration and of the influence being received from it.

By concentrating on behavioural features and based on its operational semantics, Paradigm solves the coordination of such collaborative interactions through substantially reducing the behavioural consistency issues as follows. Interaction is always between purely sequentially behaving, i.e. ongoing or running, things or threads. This is captured by a state transition diagram (STD). Built on the notion of an STD, Paradigm has four other basic notions: phase, (connecting) trap, role and consistency rule. During the collaborative interaction an ongoing STD gets influenced through a temporary constraint imposed on it: a phase, which is a subSTD of the ongoing STD, the part of it to which the ongoing behaviour is restricted. Similarly, during the collaborative interaction an ongoing STD contributes information about progress within the phase currently imposed: a trap, being a subset of the states of a phase which, as a subset, cannot be left as long the phase remains imposed. Thus a trap entered can be seen as a further constraint, where the STD commits to stay within the current phase. If a trap of a phase is moreover connecting to a next phase, meaning that all states of the trap are states of that next phase too (but not necessarily a trap of it), that next phase is a possible candidate for being imposed right after the currently imposed phase: a phase transfer from the currently

imposed phase to that next phase to be imposed will be smooth enough. A role, yet another ongoing STD, specifies the dynamics of phases imposed and connecting traps actually used for a phase transfer. Thus, a role STD of an (ongoing) STD has phases of that ongoing STD as states and it has connecting traps between these phases as transitions and is itself ongoing, while keeping track of the phase currently imposed as well as of the trap currently committed to and possibly used for the transfer to the next phase. In this way the Paradigm language syntactically guarantees vertical dynamic consistency between an ongoing STD and its likewise ongoing role, see [9]. Finally, a consistency rule synchronizes role steps from one or more roles in the collaboration. Such a rule can be seen as a protocol step belonging to the collaboration. Data can be incorporated into a consistency rule via a change clause, see [27]. All such protocol steps together then constitute the full protocol for the collaboration.

The above is graphically summarized in subfigure 1b, concentrating on alleviating the dynamic consistency problems of subfigure 1a. The double-headed dotted arrow with lightly gray interior reflects the vertical consistency being syntactically guaranteed based on Paradigm's operational semantics. The double-headed clear-cut arrow with a similar lightly gray interior reflects the now simplified character of the original coordination problem, as at the level of the roles -if well chosen, well modeled- the essence of the behaviour is kept. Also, Paradigm's five basic notions: STD, phase, trap, role and consistency rule are indicated in subfigure 1b, positioned in the context of the collaboration diagram from UML 2.0.

The following formal definitions of the Paradigm notions underpin the above intuition.

- An *STD* Z (*state-transition diagram*) is a triple $Z = \langle \mathtt{ST}, \mathtt{AC}, \mathtt{TR} \rangle$ with $\mathtt{ST}$ the set of states, $\mathtt{AC}$ the set of actions and $\mathtt{TR} \subseteq \mathtt{ST} \times \mathtt{AC} \times \mathtt{ST}$ the set of transitions of Z, notation $x \xrightarrow{a} x'$.

- A *phase* S of an STD $Z = \langle \mathtt{ST}, \mathtt{AC}, \mathtt{TR} \rangle$ is an STD $S = \langle \mathtt{st}, \mathtt{ac}, \mathtt{tr} \rangle$ such that $\mathtt{st} \subseteq \mathtt{ST}$, $\mathtt{ac} \subseteq \mathtt{AC}$ and $\mathtt{tr} \subseteq \{ (x, a, x') \in \mathtt{TR} \mid x, x' \in \mathtt{st},\ a \in \mathtt{ac} \}$.

- A *trap* t of a phase $S = \langle \mathtt{st}, \mathtt{ac}, \mathtt{tr} \rangle$ of STD Z is a non-empty set of states $t \subseteq \mathtt{st}$ such that $x \in t$ and $x \xrightarrow{a} x' \in \mathtt{tr}$ imply $x' \in t$. If $t = \mathtt{st}$, the trap is called *trivial*. A trap t of phase S of STD Z *connects* phase S to a phase $S' = \langle \mathtt{st}', \mathtt{ac}', \mathtt{tr}' \rangle$ of Z if $t \subseteq \mathtt{st}'$. Such trap-based connectivity between two phases of Z is called a *phase transfer* and is denoted as $S \xrightarrow{t} S'$.

- A *partition* $\pi = \{ (S_i, T_i) \mid i \in I \}$ of an STD $Z = \langle \mathtt{ST}, \mathtt{AC}, \mathtt{TR} \rangle$, I a non-empty index set, is a set of pairs (S_i, T_i) consisting of a phase $S_i = \langle \mathtt{st}_i, \mathtt{ac}_i, \mathtt{tr}_i \rangle$ of Z and of a set T_i of traps of S_i.

- A *role* $Z(\pi)$ at the level of a partition $\pi = \{ (S_i, T_i) \mid i \in I \}$ of an STD $Z = \langle \mathtt{ST}, \mathtt{AC}, \mathtt{TR} \rangle$ is an STD $Z(\pi) = \langle \widehat{\mathtt{ST}}, \widehat{\mathtt{AC}}, \widehat{\mathtt{TR}} \rangle$ with $\widehat{\mathtt{ST}} \subseteq \{ S_i \mid i \in I \}$, $\widehat{\mathtt{AC}} \subseteq \bigcup_{i \in I} T_i$ and $\widehat{\mathtt{TR}} \subseteq \{ S_i \xrightarrow{t} S_j \mid i, j \in I,\ t \in \widehat{\mathtt{AC}} \}$ a set of phase transfers. Z is called the *detailed* STD underlying *global* STD $Z(\pi)$, being role $Z(\pi)$.

- A *consistency rule* ρ for an ensemble of roles $Z_1(\pi_1), \ldots, Z_k(\pi_k)$ is a mechanism for synchronizing the transitions mentioned in ρ, mainly from roles in the ensemble. As such a consistency rule ρ is denoted as a string starting with an '$*$' followed by a nonempty comma-separated list of phase transfers taken from different roles from the ensemble. The string may be preceded by one transition from a non-role STD Z. In the presence of a transition from a non-role STD Z a so-called *change clause* Z:[$y := expr$] can be part of the list, overwriting the variable y accessible for Z by the value of *expr* of appropriate type. If a change clause is inserted, the list of phase transfers may be empty. An STD Z_k occurring in the list of phase transfers, is called a *participant* of ρ; if a transition of a non-role STD Z occurs in ρ, Z is called a *conductor* of ρ. A consistency rule with a conductor is also called an *orchestration step*; a consistency rule without a conductor is also called a *choreography step*.

- A Paradigm *model* is an ensemble of STDs, roles thereof and consistency rules.

- A subset P of the consistency rules from a Paradigm model, is called *protocol* P if for any role $Z_i(\pi_i)$ occurring in a rule from P, role $Z_i(\pi_i)$ does not occur in whatever consistency rule outside P. Any consistency rule ρ belonging to a protocol P is called a *protocol step* of P. A protocol P is called a *choreography*, if all consistency rules in P are choreography steps. A protocol not being a choreography is called an *orchestration*. The conductor of an orchestration step in orchestration P is called a conductor of P too.

For more elaborate introductions to Paradigm see [9], among other things covering Paradigm's operational semantics, or see [6], its Section 2 containing a not too large and illustrative example presented in a more intuitive manner.

There are a few more things we would like to stress more explicitly, however. Paradigm models consist of quite a lot of STDs, which are all executing concurrently: detailed STDs; role STDs located on the border of a component, each one at its own port; role STDs located in a collaboration protocol, each one being placeholder of the role STD at one port and hence of the underlying participating component. Via different roles, a component can participate in different protocols.

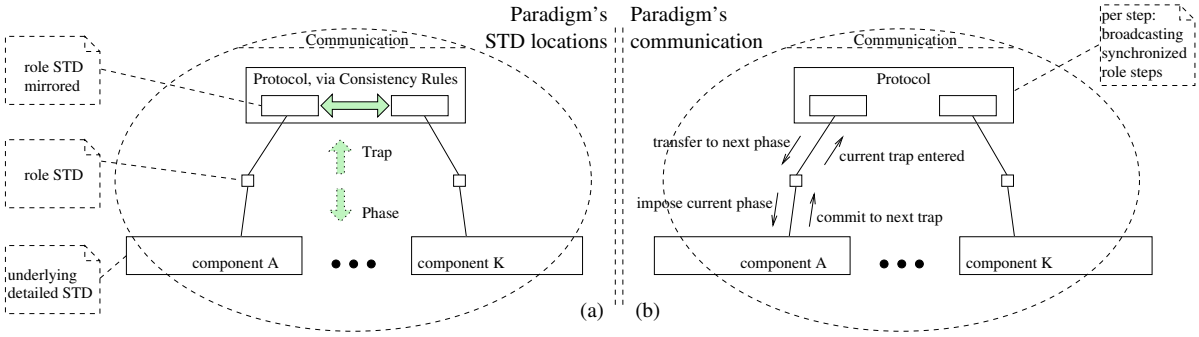

Figure 2: Communications: (a) between which STDs, (b) type of communication sent/received.

Communication or interaction between all these ongoing STDs occurs on the basis of Paradigm's phases and traps only. Both parts of Figure 2 use a UML 2.0 communication diagram, be it in a non-standard manner, to visualize the communication sent and received between the various STDs in a Paradigm model.

There are six such deviations from the regular communication diagram: (1) Our communication diagrams not only cover the protocol and thereby the roles contributed to it, but they also span relevant ports and participating components from where those roles are being contributed. (2) A port is no longer at the border of the component it is a port of, as communication between a port and its component should be made explicit. (3) There are no links between the roles within a protocol, as synchronizing role steps is done by the protocol as a whole on the basis of what the protocol received from the various ports. (4) In Figure 2a Paradigm's variated STD locations are indicated via UML's comment-like notes: a detailed STD is located at a component, every role STD thereof is located at its own specific port of the component, the same but 'mirrored' role STD is located within a protocol; moreover, Paradigm's consequences for the various dynamic consistency issues mentioned earlier are taken from Figure 1b. Please note, 'mirrored' refers to asynchronous send-receive mirroring: what is sent by a port role is received later by the mirrored protocol role and what is sent by the mirrored protocol role is received later by the port role; but the sequence of actual phase transfers taken is the same for both roles. (5) Send-receive directions are indicated as usual via an arrow along a link between sender and receiver, but in Figure 2b each such arrow is adorned with a type-like indication of the kind of signaling sent and received in that direction, instead of a sequence of concrete signals sent and received. Thus Figure 2b gives four types of communication: (*i*) From a mirrored protocol role to the corresponding port role it is the transfer from the current phase to the next current phase. (*ii*) From a port role to its component (underlying detailed) STD: the current phase is being imposed; this means, it indeed restricts the detailed STD dynamics to those actions selectable according to the current phase. (*iii*) From a component (underlying detailed) STD to a port role of it: each action corresponding to entering a trap of the current phase imposed; this means, the entering is indeed a commit to staying within the trap as long as the current phase does not change. (*iv*) From a port role to the mirrored protocol role: each trap entered subsequently within the current phase. (6) A protocol step synchronizes the type-(*i*)-sends (and not their later receives) by broadcasting from the protocol precisely the phase transfers of all protocol roles involved in the protocol step together. All other sends and receives are asynchronous.

The remainder of this section illustrates the above through the Paradigm model for the *as-is* solution of the dining philosophers. Component behaviour of the five *Phil*s and *Fork*s is specified by STDs. Subfigure 3a gives the STD for each $Phil_i$ in UML style. It says, $Phil_i$ starts in state *Thinking* and has forever cycling behaviour, passing through her three states by repeatedly taking her three actions *getHungry*, *take*, *layDown* in that order. When $Phil_i$ gets hungry in *Thinking*, she takes action *getHungry*, thus arriving in *AskingForks*. In accordance to the *as-is* solution, when

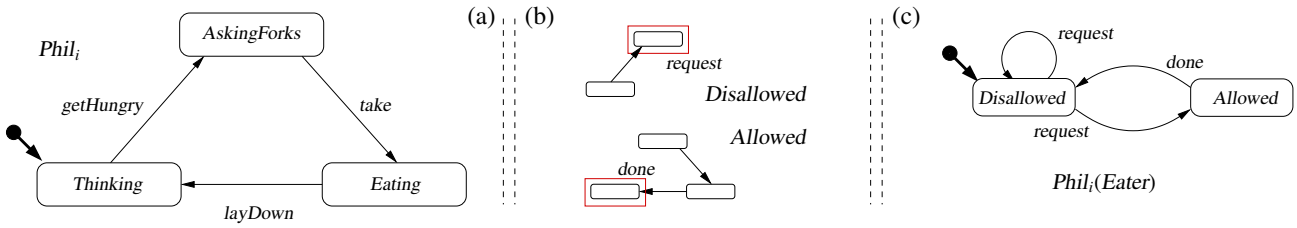

Figure 3: The five $Phil_i$: (a) STD, (b) phase / trap constraints, (c) role $Phil_i(Eater)$.

an arbitrary $Phil_i$ is sojourning in state *AskingForks*, the following is supposed to happen subsequently: (i) $Fork_i$ is claimed by her; (ii) $Fork_i$ is assigned to her; (iii) $Fork_{i+1}$ is claimed by her; (iv) $Fork_{i+1}$ is assigned to her. Thereupon $Phil_i$ performs action *take* for taking up the two *Fork*s, assigned to her by now, thus arriving in state *Eating*. Later when no longer hungry, $Phil_i$ goes from *Eating* to *Thinking* by taking action *layDown* for returning both *Fork*s to the table. We see, claiming and assigning of *Fork*s is not reflected in the STD steps of $Phil_i$.

In Paradigm, the claiming and assigning are modeled through temporary constraints on STD behaviour, the phases of the STD. $Phil_i$'s STD behaviour is influenced, i.e. temporarily constrained, by the combined behaviours of $Fork_i$ and $Fork_{i+1}$, as $Phil_i$ can proceed to state *Eating* only if both *Fork*s have been assigned to her and remain so. In addition, as long as the forks remain assigned to her, $Phil_i$ can return to *Thinking* but she should not be able to proceed to *AskingForks* while holding them. (Similarly, the behaviour of $Fork_i$ and $Fork_{i+1}$ are influenced by $Phil_i$, as we shall see below.) This intermingled influence between the components needs to be coordinated well, so that the *as-is* solution is properly modeled. This coordination aspect is explained below.

In line with the combined fork influence on $Phil_i$ behaviour indicated above, subfigure 3b visualizes two phases $Phil_i$: *Disallowed* and *Allowed*. The subfigure graphically couples the two phases with the full STD in part a, by representing each phase as a sub-STD, a scaled-down part of $Phil_i$. Because of readability such scaling down normally does not allow to incorporate state or action names. As one can see, phase *Disallowed* (on top) prohibits $Phil_i$ to be in *Eating*, but she may get as far as *AskingForks*. Contrarily, phase *Allowed* (at bottom) permits $Phil_i$ to enter and to leave *Eating* once, but returning to *AskingForks* is not allowed. Subfigure 3c specifies STD role $Phil_i(Eater)$ through which $Phil_i$ participates in a collaboration called $Phil2Forks_i$ (cf. Figure 6). Note, the phases of $Phil_i$ return as states of role STD $Phil_i(Eater)$.

Phase drawings as in subfigure 3b, are additionally decorated with one or more polygons, each polygon grouping states of that phase. In the simple case here polygons are rectangles comprising a single state. Polygons visualize traps. It is easy to verify, each polygon / trap once entered cannot be left as long as the phase remains the constraint imposed. Note, the traps label transitions in role STD $Phil_i(Eater)$, see subfigure 3c. Being a constraint committed to, a trap of a phase serves as a guard for a phase transfer from that phase.

Thus, role $Phil_i(Eater)$ behaviour expresses the ongoing alternation between *Disallowed* and *Allowed*: phase transfer from *Disallowed* to *Allowed* only happens after connecting trap *request* has been entered, intuitively meaning that she holds both forks in her hands. Similarly, phase transfer from *Allowed* to *Disallowed* only happens after connecting trap *done* has been entered, indicating that $Phil_i$'s hunger has been satisfied. Moreover, an explicitly prolonged sojourn in *Disallowed* can happen after the (connecting) trap *request* has been entered, expressing the waiting of $Phil_i$ for both forks. Though the choice of the *request* step appears as non-deterministic, it is in fact deterministic and is resolved by the combined behaviour of $Fork_i$ and $Fork_{i+1}$.

The STD $Fork_i$ is visualized in subfigure 4a. State *AtRH* means, $Fork_i$ is assigned to (the right hand of) $Phil_{i-1}$. Similarly, state *AtLH* means, $Fork_i$ is assigned to (the left hand of) $Phil_i$. To express not being assigned to any of the two philosophers $Phil_{i-1}$ or $Phil_i$, STD $Fork_i$ is on the table, but in two different states *LH?* and *RH?*, reflecting a different bias: in *LH?* the bias is to $Phil_i$ and in *RH?* the bias is to $Phil_{i-1}$. In addition, upon returning to the table from having been assigned to a philosopher's hand, the bias is first to the other philosopher. This means, each *Fork* follows a round robin approach for honoring requests from $Phil_{i-1}$ and $Phil_i$, rather than a non-deterministic one. Figure 4 also presents two roles of $Fork_i$: role *ForLH* for collaborating with $Phil_i$ (left hand) in subfigure 4c, and role *ForRH* for collaborating with $Phil_{i-1}$ (right hand) in subfigure 4e. Role *ForLH* is based on phases *Freed* and *Claimed* and their connecting traps *gone* and *got* as given in subfigure 4b. As in Figure 3 the graphical representation of the phases of

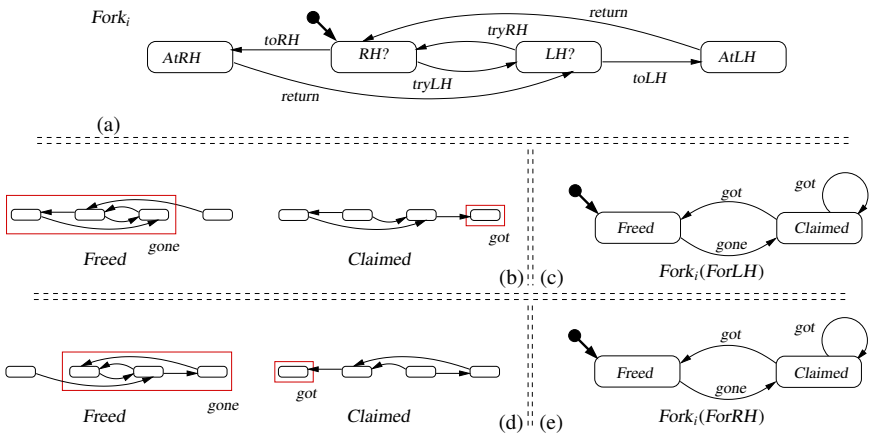

Figure 4: Five $Fork_i$: (a) STD, (b,d) phase/trap constraints per (c,e) role $Fork_i(ForLH)$, $Fork_i(ForRH)$.

$Fork_i$ keeps the form of the original STD, be it scaled down and again without original names of states and transitions. Likewise, role *ForRH* is based on phases carrying the same name but having a different specification, as here the fork role is meant for the right hand of philosopher $Phil_{i-1}$. Role *ForLH* is relevant for the same fork being picked up or laid down, but by the left hand of philosopher $Phil_i$. Note for instance, how in *Freed* of role *ForLH* the particular $Fork_i$ is being steered towards giving up staying assigned to $Phil_i$'s left hand, thus returning to the table with the possibility to get assigned to $Phil_{i-1}$'s right hand but, for the moment, not to $Phil_i$'s left hand.

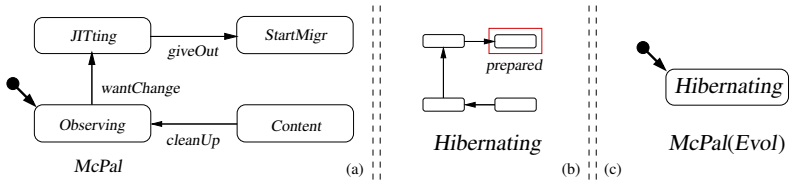

Figure 5: *McPal*: (a) STD, (b) phase/trap constraint (c) role *McPal*(*Evol*).

The five *Phil*s and *Fork*s are all component ingredients needed for the *as-is* system. In view of still unknown later adaptation, an extra STD *McPal* is in place, see Figure 5a. Initially, *McPal* is in its hibernating form, not interfering at all with the *as-is* system, but with the ability to interfere with itself first, thus adapting itself and then later, as a consequence of its gained dynamics, to interfere with the *as-is* system. Subfigures 5b and 5c underline this idea: (i) via phase *Hibernating* being the full *McPal* behaviour as long as *McPal* has not adapted itself yet; (ii) via role *Evol* which is restricted to sojourning in phase *Hibernating* as long as *McPal* remains unchanged. Thus we see, *McPal* starts in *Observing* and via *JITting* can go as far as *StartMigr*, which coincides with entering trap *prepared* of *Hibernating*. What cannot be seen from the figure but only from the consistency rules given below, through step *giveOut* leading into trap *prepared*, the hibernating *McPal* will extend the Paradigm *as-is* model specification with a specification of a *to-be* model as well as with a well-fitting model fragment for possible migration trajectories from *as-is* to *to-be*. To this aim, *McPal* embodies the reflectivity of a Paradigm model, by owning a local variable *Crs* where it stores the current model specification: consistency rules with all STDs, phases, traps and roles involved. Thus, by taking step *giveOut*, *Crs* is being extended, with at least one step series leaving *StartMigr*, such that *McPal* is no longer hibernating and able to coordinate the various migration trajectories. Having returned to phase *Hibernating*, step *cleanUp* from *Content* to *Observing* then refreshes *Crs* by removing all model fragments obsolete by then, keeping the *to-be* model only. Note, so far *McPal* is the same as in [5, 8].

In terms of the STDs, phases, traps and roles, Paradigm defines the 'coordination glue' between them through its consistency rules, being a synchronization of single role steps from different roles. The set of consistency rules for

the coordination of the *as-is* system, with *McPal* in place, is as follows.

$$* \ \mathit{Phil}_i(\mathit{Eater}) \colon \mathit{Disallowed} \xrightarrow{\mathit{request}} \mathit{Disallowed}, \ \mathit{Fork}_i(\mathit{ForLH}) \colon \mathit{Freed} \xrightarrow{\mathit{gone}} \mathit{Claimed} \tag{1}$$

$$* \ \mathit{Fork}_i(\mathit{ForLH}) \colon \mathit{Claimed} \xrightarrow{\mathit{got}} \mathit{Claimed}, \ \mathit{Fork}_{i+1}(\mathit{ForRH}) \colon \mathit{Freed} \xrightarrow{\mathit{gone}} \mathit{Claimed} \tag{2}$$

$$* \ \mathit{Fork}_{i+1}(\mathit{ForRH}) \colon \mathit{Claimed} \xrightarrow{\mathit{got}} \mathit{Claimed}, \ \mathit{Phil}_i(\mathit{Eater}) \colon \mathit{Disallowed} \xrightarrow{\mathit{request}} \mathit{Allowed} \tag{3}$$

$$* \ \mathit{Phil}_i(\mathit{Eater}) \colon \mathit{Allowed} \xrightarrow{\mathit{done}} \mathit{Disallowed}, \tag{4}$$

$$\mathit{Fork}_i(\mathit{ForLH}) \colon \mathit{Claimed} \xrightarrow{\mathit{got}} \mathit{Freed}, \ \mathit{Fork}_{i+1}(\mathit{ForRH}) \colon \mathit{Claimed} \xrightarrow{\mathit{got}} \mathit{Freed}$$

$$\mathit{McPal} \colon \mathit{JITting} \xrightarrow{\mathit{giveOut}} \mathit{StartMigr} \quad * \ \mathit{McPal} \colon [\ \mathit{Crs} := \mathit{Crs} + \mathit{Crs}_{\mathit{migr}} + \mathit{Crs}_{\mathit{toBe}}\ ] \tag{5}$$

$$\mathit{McPal} \colon \mathit{Content} \xrightarrow{\mathit{cleanUp}} \mathit{Observing} \ * \ \mathit{McPal} \colon [\ \mathit{Crs} := \mathit{Crs}_{\mathit{toBe}}\ ] \tag{6}$$

Rules (1)–(4) are choreography steps. Moreover, for each fixed i, they together cover all transitions from the three roles $\mathit{Phil}_i(\mathit{Eater})$, $\mathit{Fork}_i(\mathit{ForLH})$, $\mathit{Fork}_{i+1}(\mathit{ForRH})$. So they together are a protocol, a choreography in particular, which we call $\mathit{Phil2Forks}_i$. The corresponding UML-like collaboration diagram, having the same name, is given in Figure 6. Rules (5), (6) are orchestration steps with conductor *McPal* and with only a change clause behind the $*$. Hence, conductor *McPal* is not coordinating ongoing collaborative *as-is* behaviour, but through applying rule 5 it can change the specification of the Paradigm model in an as yet unknown and hopefully well-defined manner.

It is through consistency rules (1)–(4) the deadlock-prone solution is achieved. The choreography can be read like this (numbers referring to rules): (1) if Phil_i wants to eat and her left Fork_i hasn't been claimed yet, it is claimed; (2) if Phil_i has got her left Fork_i assigned and her right Fork_{i+1} hasn't been claimed yet, it is claimed; (3) if Phil_i has got her right Fork_{i+1} assigned, she is allowed to eat and can start doing so; (4) if Phil_i stops eating, her Fork_i and Fork_{i+1} are being freed and she is prohibited to eat any longer. In addition, rules (5)–(6) are orchestration steps with *McPal* as conductor, not influencing ongoing collaborative *as-is* behaviour, but (5) extending the *as-is* model specification just before the migration should start and (6) reducing the model specification to the *to-be* specification aimed at immediately after the migration has been done. The migration itself is not specified here, as neither the *to-be* situation nor intermediate migration are known at present. Please note, *Crs* is a variable of *McPal*, being of type 'Paradigm model fragment'. The model fragment actually comprises a set of consistency rules together with the relevant instances of Paradigm notions referred to in the rules: detailed STDs, phases, traps and role STDs. The initial value of *Crs* is the Paradigm model for the *as-is* situation: rules (1)–(6) together with the detailed STDs, phases, traps and role STDs from Figures 3, 4 and 5. Later, after the migration, the same variable will contain the Paradigm model for a still unknown *to-be* situation. Similarly, both $\mathit{Crs}_{\mathit{migr}}$ and $\mathit{Crs}_{\mathit{toBe}}$ are variables of *McPal*, being of type Paradigm model fragment too. Although concrete migrations as well as *to-be* situations are supposed to be known when *McPal* leaves state *JITting*, the two model fragments $\mathit{Crs}_{\mathit{migr}}$ and $\mathit{Crs}_{\mathit{toBe}}$ are arranged as variables instead of as constants. This is done in view of allowing the actual values of these variables to depend on a particular migration trajectory traversed. Thus, the model fragments can vary on-the-fly of the migration as will be explained in more detail in Sections 5 and 6.

3. Dining philosophers *to-be*: no deadlock, no starvation

Before addressing migration in Sections 4 and 5, this section presents the goal to be reached by the migration, referred to as the *to-be* system or *to-be* solution. The problem situation is the same as the one of the *as-is* situation, the five Phil_i and five Fork_i. But, the solution is better now: no deadlock and also no starvation. This is achieved in the following well-known way: for at least one Phil_i, but not for all five, the order of claiming her forks Fork_i and Fork_{i+1} is being reversed.

For the Paradigm model of the *to-be* solution this means, all STDs, phases, traps and roles remain the same, but the consistency rules are different. For their formulation we need some extra notation. Let the index sets L, R be a non-empty disjoint partitioning of $\{1..5\}$, L referring to those Phil_is for which the left Fork_i is claimed first, and R referring to those Phil_is for which the right Fork_{i+1} is claimed first. In the consistency rules below we let i range

over L and j over R (with the usual cyclic values of $j-1=5$ for $j=1$ and $j+1=1$ for $j=5$ as we similarly have for i already).

$$* \ \mathit{Phil}_i(\mathit{Eater}){:}\ \mathit{Disallowed} \xrightarrow{\mathit{request}} \mathit{Disallowed},\ \mathit{Fork}_i(\mathit{ForLH}){:}\ \mathit{Freed} \xrightarrow{\mathit{gone}} \mathit{Claimed} \tag{7}$$

$$* \ \mathit{Fork}_i(\mathit{ForLH}){:}\ \mathit{Claimed} \xrightarrow{\mathit{got}} \mathit{Claimed},\ \mathit{Fork}_{i+1}(\mathit{ForRH}){:}\ \mathit{Freed} \xrightarrow{\mathit{gone}} \mathit{Claimed} \tag{8}$$

$$* \ \mathit{Fork}_{i+1}(\mathit{ForRH}){:}\ \mathit{Claimed} \xrightarrow{\mathit{got}} \mathit{Claimed},\ \mathit{Phil}_i(\mathit{Eater}){:}\ \mathit{Disallowed} \xrightarrow{\mathit{request}} \mathit{Allowed} \tag{9}$$

$$* \ \mathit{Phil}_i(\mathit{Eater}){:}\ \mathit{Allowed} \xrightarrow{\mathit{done}} \mathit{Disallowed}, \quad \mathit{Fork}_i(\mathit{ForLH}){:}\ \mathit{Claimed} \xrightarrow{\mathit{got}} \mathit{Freed},\ \mathit{Fork}_{i+1}(\mathit{ForRH}){:}\ \mathit{Claimed} \xrightarrow{\mathit{got}} \mathit{Freed} \tag{10}$$

$$* \ \mathit{Phil}_j(\mathit{Eater}){:}\ \mathit{Disallowed} \xrightarrow{\mathit{request}} \mathit{Disallowed},\ \mathit{Fork}_{j+1}(\mathit{ForRH}){:}\ \mathit{Freed} \xrightarrow{\mathit{gone}} \mathit{Claimed} \tag{11}$$

$$* \ \mathit{Fork}_{j+1}(\mathit{ForRH}){:}\ \mathit{Claimed} \xrightarrow{\mathit{got}} \mathit{Claimed},\ \mathit{Fork}_j(\mathit{ForLH}){:}\ \mathit{Freed} \xrightarrow{\mathit{gone}} \mathit{Claimed} \tag{12}$$

$$* \ \mathit{Fork}_j(\mathit{ForLH}){:}\ \mathit{Claimed} \xrightarrow{\mathit{got}} \mathit{Claimed},\ \mathit{Phil}_j(\mathit{Eater}){:}\ \mathit{Disallowed} \xrightarrow{\mathit{request}} \mathit{Allowed} \tag{13}$$

$$* \ \mathit{Phil}_j(\mathit{Eater}){:}\ \mathit{Allowed} \xrightarrow{\mathit{done}} \mathit{Disallowed}, \quad \mathit{Fork}_j(\mathit{ForLH}){:}\ \mathit{Claimed} \xrightarrow{\mathit{got}} \mathit{Freed},\ \mathit{Fork}_{j+1}(\mathit{ForRH}){:}\ \mathit{Claimed} \xrightarrow{\mathit{got}} \mathit{Freed} \tag{14}$$

$$\mathit{McPal}{:}\ \mathit{JITting} \xrightarrow{\mathit{giveOut}} \mathit{StartMigr} \ * \ \mathit{McPal}{:}\ [\ \mathit{Crs} := \mathit{Crs} + \mathit{Crs}_{\mathit{migr}} + \mathit{Crs}_{\mathit{toBe}}\ ] \tag{15}$$

$$\mathit{McPal}{:}\ \mathit{Content} \xrightarrow{\mathit{cleanUp}} \mathit{Observing} \ * \ \mathit{McPal}{:}\ [\ \mathit{Crs} := \mathit{Crs}_{\mathit{toBe}}\ ] \tag{16}$$

Rules (7)–(10) together with (15)–(16) are exactly the rules (1)–(6) from Section 2. It is not difficult to observe, rules (11)–(14) mirror (7)–(10) by reversing the order of claiming indeed. Furthermore, note that only the consistency rules have been adapted, so the change remains restricted to the 'coordination glue' between the components, particularly the choreography steps only. *McPal* is in hibernation, as usual with no migration going.

3.1. Dining philosophers as-is *and* to-be*: mCRL2 specification*

By now we have given two different solutions for the dining philosophers problem. The two solutions have been represented in the Paradigm language, marking the starting point and end point of the migration from the *as-is* to the *to-be* solution. We have also indicated that the *as-is* solution is deadlock-prone and the *to-be* solution is deadlock-free and starvation-free. In this subsection we justify this claim by a rigorous analysis of the two models, and formally show that this is indeed the case. We do so by translating the Paradigm models in the process modeling language mCRL2, which comes equipped with a toolset for formal verification and validation.[1]

As the two models, *as-is* and *to-be*, are rather simple, their mCRL2 specifications can also be simple. Nevertheless, we present them here for two reasons. Firstly, to informally introduce the mCRL2 specification language. Secondly, to highlight the main lines of the translation of Paradigm models into mCRL2 specifications, and, thus, prepare the ground for the analysis of the migration model in Section 7, which is much more complicated. In fact, the mCRL2 representations of the *as-is* and *to-be* Paradigm models are sub-specifications of the migration model. This is to be expected as these two Paradigm models are sub-models of the migration model given in Section 4.

mCRL2 is a process modeling language [30, 29] used for description of concurrent processes based on the process algebra ACP [16, 13]. It includes facilities for multi-party synchronization and user-defined abstract datatypes. The language is supported by an extensive toolset which allows for model checking properties expressed in the modal μ-calculus for a given mCRL2 specification. In addition, the toolset provides support for manipulation of the specification, such as, state space generation and visualization, state space reduction, abstraction, simulation, etc., which makes the toolset rather suited for our modeling and analysis.

Processes in mCRL2, just like in any other process algebraic language, are described by algebraic expressions. Non-terminating processes can be expressed by recursive specifications. In our translation of the *as-is* situation, and

[1] Available from `http://www.mcrl2.org`.

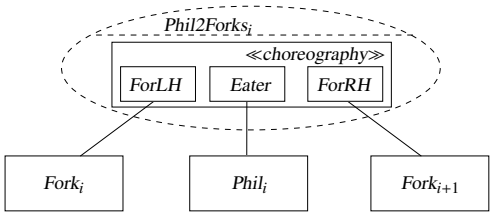

Figure 6: Collaboration of *Phil*$_i$, *Fork*$_i$ and *Fork*$_{i+1}$.

similar for the *to-be*, we first identify five protocols $\texttt{Prot}_i$, $i = 1, \ldots, 5$ that run in parallel. They correspond to the five collaborations *Phil2Forks*$_i$ depicted in Figure 6. We also define philosopher identities $\texttt{PID} = \{ \texttt{P}_i \mid i = 1, \ldots, 5 \}$ and fork identities $\texttt{FID} = \{ \texttt{F}_i \mid i = 1, \ldots, 5 \}$, and bind them to their corresponding protocol $\texttt{Prot}_i$. For instance, $\texttt{Prot}_1$ consists of three communicating processes, viz. processes for $\texttt{Phil}_1$, $\texttt{Fork}_1$ (as her left fork) and $\texttt{Fork}_2$ (as her right fork). The binding is defined by means of the mappings `Phil2Prot`, `LFork2Prot` and `RFork2Prot`.

```
1.  proc Phil(i:PID, s:PState, p:PPhase) =
2.       ( ( s == Thinking ) && ( p == Disallowed ) ) →
                 getHungry(i) .  Phil(i,AskingForks,p) +
3.       ( ( s == AskingForks ) && ( p == Allowed ) ) → take(i) .  Phil(i,Eating,p) +
4.       ( ( s == Eating ) && ( p == Allowed ) ) → layDown(i) .  Phil(i,Thinking,p) +
5.  %% left-right order as-is
6.       ( ( p == Disallowed ) && ( s == AskingForks ) ) →
                 rule01DisallowedRequest(Phil2Prot(i)).  Phil(i,s,Disallowed) +
7.       ( ( p == Disallowed ) && ( s == AskingForks ) ) →
                 rule03DisallowedRequest(Phil2Prot(i)) .  Phil(i,s,Allowed) +
8.       ( ( p == Allowed ) && ( s == Thinking ) ) →
                 rule04AllowedDone(Phil2Prot(i)) .  Phil(i,s,Disallowed);
```

Table 1: mCRL2 specification of *Phil*$_i$ in *as-is*.

The mCRL2 specifications of *Phil*$_i$ are shown in Table 1 and Table 2 for *as-is* and *to-be*, respectively. A first thing to note is that the detailed STD *Phil*$_i$ and the role STD *Phil*$_i$(*Eater*) are combined here in one single mCRL2 process, namely process $\texttt{Phil}_i$. Therefore, besides the philosopher's identifier `i`, two other parameters in the definition of the process `Phil` are used to store: (i) `s` – the current state of the *Phil*$_i$ STD, and (ii) `p` – the current phase of role *Phil*$_i$(*Eater*). In Table 1, lines 2–4 specify the local actions *Phil*$_i$ can execute, each of them enabled only if the role *Eater* is residing in the phase that allows the transition. Lines 6–8 specify the actions for the phase transfers of role *Phil*$_i$(*Eater*), allowed only if the local behaviour has progressed enough, i.e. the connecting trap has been entered. The actions here indicate the collaboration of *Phil*$_i$ in the protocol *Phil2Forks*$_i$ via its role *Eater*, just as defined in the consistency rules (1), (3) and (4). In fact, the action names clearly point at the consistency rule they belong to. The parameter of these actions stores the associated protocol $\texttt{Prot}_i$. The execution of a consistency rule in mCRL2 is specified as the synchronization of several actions. The groups of actions that synchronize are defined by the communication function. Furthermore, the exact actions that are executed within the same protocol, and which result in a synchronized action, are distinguished by the protocol identifier $\texttt{Prot}_i$, obtained via the mappings `Phil2Prot`, `LFork2Prot` and `RFork2Prot`, the last two called in the *Fork*$_i$ specification.

The mCRL2 specification for *Fork*$_i$ and its roles *ForLH* and *ForRH* is defined in the same style as for the philosopher.[2] Here, we just mention that mCRL2 process $\texttt{Fork}_i$ carries four parameters: the *Fork* identifier, the current local state of the *Fork* STD, the current phase of role *ForLH* and the current phase of role *ForRH*.

[2] See `http://www.win.tue.nl/~evink/research/paradigm.html` for the full mCRL2 specification.

```
 1.  proc Phil(i:PID, s:PState, p:PPhase) =
 2.       ( ( s == Thinking ) && ( p == Disallowed ) ) →
                 getHungry(i) .  Phil(i,AskingForks,p) +
 3.       ( ( s == AskingForks ) && ( p == Allowed ) ) → take(i) .  Phil(i,Eating,p) +
 4.       ( ( s == Eating ) && ( p == Allowed ) ) → layDown(i) .  Phil(i,Thinking,p) +
 5.  %% left-right order to-be
 6.       ( ( p == Disallowed ) && ( s == AskingForks ) ) →
                 rule07DisallowedRequest(Phil2Prot(i)) .  Phil(i,s,Disallowed) +
 7.       ( ( p == Disallowed ) && ( s == AskingForks ) ) →
                 rule09DisallowedRequest(Phil2Prot(i)) .  Phil(i,s,Allowed) +
 8.       ( ( p == Allowed ) && ( s == Thinking ) ) →
                 rule10AllowedDone(Phil2Prot(i)) .  Phil(i,s,Disallowed) +
 9.  %% right-left order to-be
10.       ( ( p == Disallowed ) && ( s == AskingForks ) ) →
                 rule11DisallowedRequest(Phil2Prot(i)) .  Phil(i,s,Disallowed) +
11.       ( ( p == Disallowed ) && ( s == AskingForks ) ) →
                 rule13DisallowedRequest(Phil2Prot(i)) .  Phil(i,s,Allowed) +
12.       ( ( p == Allowed ) && ( s == Thinking ) ) →
                 rule14AllowedDone(Phil2Prot(i)) .  Phil(i,s,Disallowed);
```

Table 2: mCRL2 specification of $Phil_i$ in *to-be*.

As already mentioned, a philosopher can, regarding the order of picking up her forks, be either left-oriented, in which case she participates in rules (1)–(4) or (7)–(10), or right-oriented – behaviour allowed only in the *to-be* solution (and during migration) – and then she participates in rules (11)–(14). Accordingly, the mCRL2 specifications for components $Phil_i$ and $Fork_i$ only allow for left-orientation in the case of *as-is*, and allow for both orientations in the *to-be* situation. As is to be expected, the two mCRL2 specifications slightly differ, however their structure is similar.

```
15.  proc IProtRules(protid:ProtID, irs:IRuleSet) =
16.       ( irs == LO) → (
               rule07ok(protid) .  IProtRules() + rule08ok(protid) .  IProtRules() +
               rule09ok(protid) .  IProtRules() + rule10ok(protid) .  IProtRules()  ) +
17.       ( irs == RO) → (
               rule11ok(protid) .  IProtRules() + rule12ok(protid) .  IProtRules() +
               rule13ok(protid) .  IProtRules() + rule14ok(protid) .  IProtRules()  );
```

Table 3: mCRL2 specification of IProtRules process.

The mCRL2 process $\texttt{Phil}_i$ of the *to-be* model is given in Table 2. It describes both possibilities: that a philosopher is left-oriented – lines 6–8, and that a philosopher is right-oriented – lines 10–12. Although the modeling can be done differently, it can be seen in Table 2 that we have opted for an mCRL2 specification where the choice of orientation is not made in the $\texttt{Phil}_i$ process. Instead, this choice is specified by a separate process, IProtRules, one for each protocol $\texttt{Prot}_i$, for $i = 1, \ldots, 5$ (see Table 3). This better reflects the distributed nature of the components. Thus, in $\texttt{IProtRules}_i$ the orientation of $\texttt{Prot}_i$ is stored in the parameter irs for individual (consistency) rules, with $\texttt{irs} \in \{\texttt{LO}, \texttt{RO}\}$, i.e. left-order or right-order. Accordingly, the $\texttt{IProtRules}_i$ process exactly provides synchronization within $\texttt{Prot}_i$ as specified in the consistency rules for the given orientation in irs; namely, if $\texttt{irs} = \texttt{LO}$ then $\texttt{IProtRules}_i$ acts left-order according to rules (7)–(10) (code lines 6–8 in Figure 2), otherwise $\texttt{IProtRules}_i$ acts right-order and follows the consistency rules (11)–(14) (code lines 10–12 in Figure 2). To accommodate this modeling decision, the

communication function in the mCRL2 model of the *to-be* situation is extended to the processes $\texttt{IProtRules}_i$. For instance, consistency rule (7) is captured by the communication:

$$\texttt{rule07DisallowedRequest} \mid \texttt{rule07FreedGone} \mid \texttt{rule07ok} \rightarrow \texttt{rule07},$$

where rule07DisallowedRequest is executed by $\texttt{Phil}_i$, rule07FreedGone is executed by $\texttt{Fork}_i$, rule07ok is executed by $\texttt{IProtRules}_i$ and rule07 is the result of this multi-party synchronization involving three processes. Note that process IProtRules does not influence the behaviour of other components in any other way; the construct ruleXXok(protid).IProtRules() indicates that after action ruleXXok(protid) is executed, no value of any of the parameters of IProtRules will be changed.

Now that we have outlined the mCRL2 specification of the *as-is* as well as the *to-be* protocol, one may wonder why the processes IProtRules have to be added at all. It is clear to see that the protocol orientation can be easily solved, for instance, by adding irs as a parameter to process Phil. The idea behind the process IProtRules is that it decides which set of consistency rules is to be followed based on the information this process has about the current situation of the overall system or parts of the system. While in the above example of the *to-be* protocol such a decision can be easily embedded and made by another component of the model, this is not the case in general. As we will discuss in Section 4, in the migration model, during the execution the dynamic changes may happen throughout and across all components involved. Hence, having as solution a monitoring process which observes the execution and guides the system dynamics accordingly shows very suitable. In fact, these monitoring processes should be seen as explicit specifications in mCRL2 of the clause changes in Paradigm models. Now the system dynamics can be changed (enriched or reduced) during the execution too. A more extensive example of IProtRules will be discussed in the migration Section 7.

3.2. Dining philosophers as-is *and* to-be*: property verification*

To confirm that the Paradigm model of the *as-is* solution is deadlock prone, and that the Paradigm model of the *to-be* solution is deadlock and starvation free, we formally analyze the two mCRL2 specifications using the mCRL2 model checker. The properties we want to check for the models are expressed in a variant of the modal μ-calculus [21], the property language for mCRL2. For both models we check the deadlock-free property, expressed as

```
[ true* ] < true > true
```

It reads as: after every finite sequence of actions there is always an action that can be executed. As expected, the model checker returns false when this property is checked for the mCRL2 *as-is* model. It returns true when the property is checked for the mCRL2 *to-be* specification.

If a philosopher, who has requested a fork (meaning requested to eat) is never allowed to eat by her neighboring philosophers, because they are always 'faster' in claiming and picking up the forks, then it is said that the philosopher would starve. In order to avoid this undesirable property and to allow each process to get access to the resource once requested, in our case to allow each philosopher always to eventually eat, it is a standard solution to use an external process for fork scheduling. Each philosopher has to ask and get permission to pick up the forks. However, in the Paradigm models defined above, no such explicit scheduling component has been specified. Instead, we benefit from the possibility to impose constraints directly on the *Fork* detailed STD, by means of roles' phases and traps. In this case, the combination of the two roles *ForLH* and *ForRH*, the phases defined there, *Freed* in, e.g. *ForLH*, and *Claimed* for *ForRH*, together with their traps enforce a round-robin usage of the *Fork*. As a result, if the *Fork* has been requested by both philosophers, none of them will be allowed to use it more than once in a row. This informal analysis has been supported by the results of the model checking analysis performed on the mCRL2 specification. The following property returns true when checked for the *to-be* model:

```
forall i:ProtID . [ true*.rule07(i) ]
    mu Y . ( [ !rule09(i) ]Y || ( nu X . < tau > X && < true*.rule09(i) > true ))
```

It should be noted that the mCRL2 specification has been manipulated in the following way: all actions that are local to the processes Phil and Fork have been turned into internal τ actions (using the hiding operator). Thus, only the Prot actions rule07, ..., rule14 have been kept observable. Due to possible internal τ-divergence in the specification, we

assume strong fairness so that we can interpret the subformula `nu X . < tau > X && < true*.rule09(i) > true` to mean that any τ-loop that avoids the application of rule (9) is eventually left. The μ-calculus formula stated above, expresses in total that in a left-oriented `Prot`$_i$ involving rules (7) up to (10), once action `rule07(i)` has been executed, meaning that *Phil*$_i$ has requested her left fork, under the fairness assumption as explained above, action `rule09(i)` will always be executed eventually, providing *Phil*$_i$ with her right fork and allowing her to eat. Of course, the case for the right-orientation is entirely symmetric, with properly selected action names, therefore non-starvation holds for all protocols. As argued, for a situation as the *to-be* solution, where both left-oriented and right-oriented protocols are in place simultaneously, both the property displayed above and its right-oriented version hold.

4. Migration coordination set-up among helpers

As Section 3 announced, the envisioned migration is from the *as-is* situation to the *to-be* situation, i.e. starting from the deadlock-prone solution of the dining philosophers problem to the well-known, far better deadlock-free and starvation-free solution, where at least one *Phil*$_i$ gets her *Fork*$_i$ and *Fork*$_{i+1}$ assigned in a different order. So, there is ample room for different *to-be* solutions meeting the requirements. Also, for each *to-be* solution different migration trajectories reaching it can be developed.

In view of this observation, we restrict the range of our *to-be* solutions as follows: regarding the sets L, R introduced above — claiming left fork first for *Phil*$_i$ with $i \in L$ versus claiming right fork first for *Phil*$_j$ with $j \in R$ — we require L and R to have either 2 or 3 elements. Moreover, we demand if $L = \{i, i'\}$ then *Phil*$_i$ and *Phil*$_{i'}$ are not adjacent, i.e. $i = i' + 2$ or $i = i' + 3$, and similarly, if $R = \{j, j'\}$ then *Phil*$_j$ and *Phil*$_{j'}$ are not adjacent. Thus, for the *to-be* solution, of the two groups of *Phil*s one group consists of two *Phil*s and the other group consists of three *Phil*s. In addition, the *Phil*s from the group of two are not neighbors. This reduction in admitted *to-be* solutions will illustrate the interplay of central change management and local change management more clearly[1]. (By the way, a much easier migration and *to-be* solution could have been, letting *Phil*$_1$ be the only philosopher who swaps to taking the right fork first and that only when she gets hungry anew or if deadlock has occurred already. But then, distribution of migration coordination would have made no sense, being not very helpful in view of our goal to investigate the distributed potential of *McPal*.)

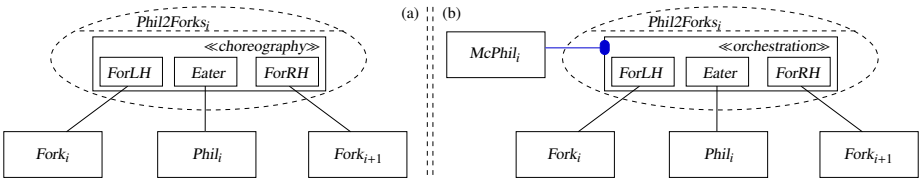

Figure 7: Two collaboration snapshots (a) during hibernation, (b) during migration.

Before addressing the actual migration through coordination not yet specified, we want to clarify an important structural difference in the collaboration of *Phil*$_i$ and her two forks *Fork*$_i$ and *Fork*$_{i+1}$ during *McPal*'s hibernation and during migration. Figure 7a gives, in UML-style, the collaboration diagram *Phil2Forks*$_i$ during hibernation. It says, the three components *Phil*$_i$, *Fork*$_i$, *Fork*$_{i+1}$ are involved in it and, in line with the Paradigm model, they contribute to it via their respective roles *Eater*, *ForLH*, *ForRH*. This makes the collaboration into a choreography. Note, this architectural snapshot is valid for the *as-is* as well as for the *to-be* solution, the difference being in the behaviour only.

During the migration the collaboration has a slightly different structure, however. See Figure 7b. For each *Phil*$_i$ an extra component *McPhil*$_i$ is involved, meant as delegated helper of *McPal* for the *Phil2Forks*$_i$ collaboration only, to enlarge *McPal*'s influence. As we shall see below, *McPhil*$_i$ joins the collaboration as a new local driver of the ongoing choreography, thereby turning *Phil2Forks*$_i$ into an orchestration with *McPhil*$_i$ as its conductor, with essentially the same collaborative behaviour for a short while. Then, as conductor in place, it migrates the orchestration and informs

[1] For an animated migration trajectory, see the extended version of the FACS 2010 presentation at `http://www.win.tue.nl/~evink/research/paradigm.html`.

McPal about the result achieved so far, whereupon *McPal* decides about keeping or altering an intermediate result. Then $McPhil_i$ does so and steps back as conductor, turning collaboration $Phil2Forks_i$ into a choreography again. More detail on architectural changes during migration are given in Section 6.

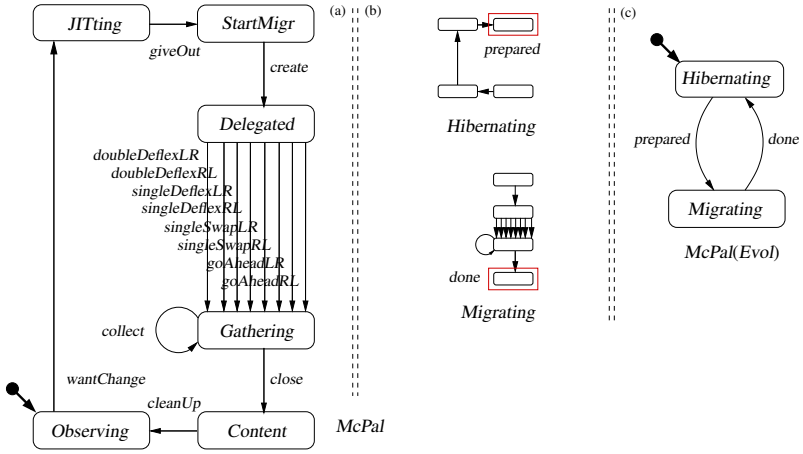

Figure 8: *McPal* during migration: (a) STD, (b) phase / trap constraints, (c) role *McPal*(*Evol*).

McPal's actual migration activity is outlined in Figure 8: *McPal*, upon awakening from phase *Hibernating*, becomes active within phase *Migrating* as main conductor of the migration orchestration. Here it immediately delegates the actual migration coordination to its five helpers $McPhil_i$, by taking step *create*. In doing so, *McPal* provides each $McPhil_i$ with the orchestration rules for the local migration, while keeping the *as-is* choreography rules. Arrived in state *Delegated*, *McPal* then waits for the local results from the $McPhil_i$, being preliminary only. The preliminary results can be of two kinds: either $Phil_i$ (still) belongs to L or $Phil_i$ (now) belongs to R. Depending on the combined results of the five $McPhil_i$, conductor *McPal* takes one out of eight possible steps to state *Gathering*: by possibly letting zero, one or two specific $McPhil_i$ adjust their preliminary result when finalizing and by letting the other $McPhil_i$ make their preliminary results permanent. In state *Gathering*, *McPal* starts collecting the five sets $Crs_{i,toBe}$ of consistency rules the various $McPhil_i$ have to deliver when finalizing: the *to-be* choreography local to $Phil2Forks_i$, constituting $McPhil_i$'s final result. To this aim *McPal* takes step *collect* five times, one per $McPhil_i$. After having collected the five sets $Crs_{i,toBe}$ and after the five helper $McPhil_i$ have stopped their activities, *McPal* takes step *close* to state *Content*, thus entering trap *done* marking the final stage of the migration phase.

The overall migration conducting of *McPal* sets the stage for the local migration exerted by $McPhil_i$ on the ongoing collaboration $Phil2Forks_i$. The behaviour of each $McPhil_i$ is drawn in Figure 9a. From starting state *NonExisting* to the state *Awake* it takes step *stir*, to get ready for whatever it has to do. From *Awake* it takes step *takeOver* to state *JoiningIn*, thereby removing the *as-is* choreography rules from the (local) model specification Crs_i, thus keeping the orchestration rules only, that were already added earlier by *McPal* when delegating the local migration to $McPhil_i$.

By taking step *conductToL* from state *JoiningIn* to state *ToL* and by iterating step *conductToL* in *ToL*, helper $McPhil_i$ sticks to the *as-is* orchestration, for the *L*-order that is. Similarly, by taking step *conductToR* from state *JoiningIn* to state *ToR* and by iterating step *conductToR* in *ToR*, helper $McPhil_i$ swaps the orchestration of the *as-is* choreography to the orchestration for the *R*-order. From state *ToL* helper $McPhil_i$ can, apart from iterating, either take step *choreofyToL* to state *ToBeL*, in which case $McPhil_i$ sticks to the *L*-order but turns the orchestration back into the equivalent choreography, or $McPhil_i$ can take step *choreofyToR* to state *ToBeR*, in which case $McPhil_i$ swaps to the *R*-order (on second thought, instigated by *McPal*) and moreover turns the orchestration into the equivalent choreography for the *R*-order. Analogously, from state *ToR* helper $McPhil_i$ can, apart from iterating, either take step *choreofyToR* to state *ToBeR*, in which case $McPhil_i$ sticks to the *R*-order but turns the orchestration into the equivalent choreography, or $McPhil_i$ can take step *choreofyToL* to state *ToBeL*, in which case $McPhil_i$ swaps back to the *L*-order and moreover turns the orchestration into the equivalent choreography for the *L*-order. From then on, in two consecutive steps, viz. *yawn* and *immobilize*, helper $McPhil_i$ returns to state *NonExisting*.

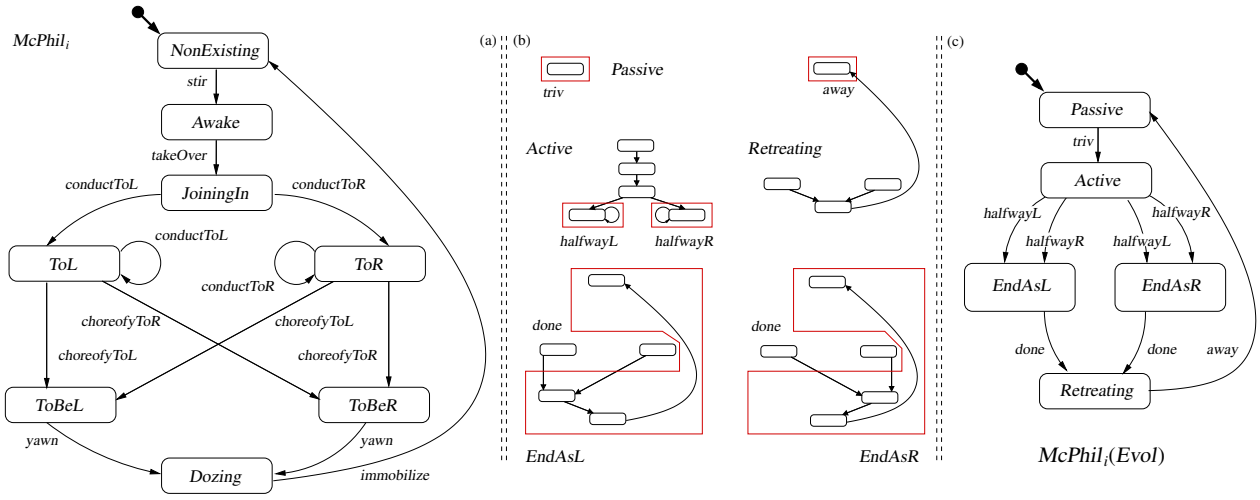

Figure 9: $McPhil_i$ during migration: (a) STD, (b) phase / trap constraints, (c) role $McPhil_i(Evol)$.

Figure 9b presents the phase and trap constraints on $McPhil_i$. Based on these constraints, role $McPhil_i(Evol)$ is given in part 9c. In phase *Passive* helper $McPhil_i$ can't do anything. In phase *Active* it can go as far as providing to *McPal* its preliminary result, being of two possible kinds, one per trap *halfwayL* or *halfwayR*. Phases *EndAsL* and *EndAsR* correspond to the two final results possible, the original *L*-order or the new *R*-order, respectively, available once trap *done* has been entered. Finally, in phase *Retreating* helper $McPhil_i$ enters trap *away*. After that it returns to *Passive* where it can't do anything.

It is stressed all this is to happen dynamically, on-the-fly, without any system halting. Consistency rules specifying this turn out to be quite technical. Therefore we discuss them separately in Section 5.

5. Migration coordination distributed among helpers

The consistency rules below specify the coordination according to Section 4's migration set-up. The technicalities of the rules mainly arise where change clauses, in view of influencing the migration, manipulate sets of rules and of the various model fragments the rules refer to. The manipulation of rules and corresponding model fragments timely adapts the coordination strategy, gracefully enforcing the system's change. The following sets of consistency rules and corresponding model fragments occur. As the relevant model fragments follow from the rules, we mention rules only when paraphrasing the sets.

$Crs_{i,asIs}$ ::= choreography of $Phil2Forks_i$, *L*-order only (as in Section 2)
Crs_{hib} ::= orchestration conducted by *McPal* during phase *Hibernating*
Crs_{noHb} ::= orchestration conducted by *McPal* during phase *Migrating*
$Crs_{i,orch}$ ::= orchestration conducted by $McPhil_i$
$Crs_{i,toBeL}$ ::= choreography of $Phil2Forks_i$ in *L*-order
$Crs_{i,toBeR}$ ::= choreography of $Phil2Forks_i$ in *R*-order
Crs_{migr} ::= Crs_{noHb} (Crs_{migr} from rule (5) here coincides with Crs_{noHb})

The above sets remain fixed, so they are constants. The sets below are variables, varying during the migration. Basically, all such variables are global, but in the beginning it is as if they all belong to *McPal*, being the one component responsible for migration coordination on the basis of a new specification coming available on leaving state *JITting*. As *McPal* is to delegate some of its migration coordination, it will also delegate ownership of some of these variables. We shall refer to such varying ownership as governance and governing.

$$
\begin{array}{lll}
\mathit{Crs} & ::= & \text{varying orchestration/choreography governed by } \mathit{McPal} \\
\mathit{Crs}_i & ::= & \text{varying } \mathit{McPhil}_i\text{-governed protocol } \mathit{Phil2Forks}_i \\
\mathit{Crs}_{i,toBe} & ::= & \text{either } \mathit{Crs}_{i,toBeL} \text{ or } \mathit{Crs}_{i,toBeR} \\
\mathit{Crs}_{toBe} & ::= & \text{growing from } \mathit{Crs}_{hib} \text{ to } \mathit{Crs}_{hib} + \mathit{Crs}_{1,toBe} + \cdots + \mathit{Crs}_{5,toBe}
\end{array}
$$

In the context of the above governance we do not want to introduce a notion of scope for the $\mathit{Crs}, \mathit{Crs}_i, \ldots$ variables. Therefore we keep these variables global. But we shall carefully arrange their usage by *McPal* and by the five McPhil_i such that it mimics scope effect, as we shall see below.

The fixed sets of the consistency rules are specified first. Note, the variable sets of consistency rules are updated through detailed change clauses involving the fixed sets.[2] Consistency rules (17)–(20) making up $\mathit{Crs}_{i,asIs}$ are exactly rules (1)–(4) from Section 2.

$$
* \ \mathit{Phil}_i(\mathit{Eater}) \colon \mathit{Disallowed} \xrightarrow{request} \mathit{Disallowed}, \ \mathit{Fork}_i(\mathit{ForLH}) \colon \mathit{Freed} \xrightarrow{gone} \mathit{Claimed} \tag{17}
$$

$$
* \ \mathit{Fork}_i(\mathit{ForLH}) \colon \mathit{Claimed} \xrightarrow{got} \mathit{Claimed}, \ \mathit{Fork}_{i+1}(\mathit{ForRH}) \colon \mathit{Freed} \xrightarrow{gone} \mathit{Claimed} \tag{18}
$$

$$
* \ \mathit{Fork}_{i+1}(\mathit{ForRH}) \colon \mathit{Claimed} \xrightarrow{got} \mathit{Claimed}, \ \mathit{Phil}_i(\mathit{Eater}) \colon \mathit{Disallowed} \xrightarrow{request} \mathit{Allowed} \tag{19}
$$

$$
\begin{aligned}
* \ \mathit{Phil}_i(\mathit{Eater}) \colon \mathit{Allowed} \xrightarrow{done} \mathit{Disallowed}, \qquad & \\
\mathit{Fork}_i(\mathit{ForLH}) \colon \mathit{Claimed} \xrightarrow{got} \mathit{Freed}, \ \mathit{Fork}_{i+1}(\mathit{ForRH}) \colon \mathit{Claimed} \xrightarrow{got} \mathit{Freed} & \qquad (20)
\end{aligned}
$$

Likewise, rules (21)–(22) making up Crs_{hib}, are exactly rules (5)–(6) and also rules (15)–(16) from Section 2 and 3, respectively.

$$
\mathit{McPal} \colon \mathit{JITting} \xrightarrow{giveOut} \mathit{StartMigr} \ * \ \mathit{McPal} \colon [\, \mathit{Crs} := \mathit{Crs} + \mathit{Crs}_{migr} + \mathit{Crs}_{toBe} \,] \tag{21}
$$

$$
\mathit{McPal} \colon \mathit{Content} \xrightarrow{cleanUp} \mathit{Observing} \ * \ \mathit{McPal} \colon [\, \mathit{Crs} := \mathit{Crs}_{toBe} \,] \tag{22}
$$

Note the two assignments to *Crs*. In rule (21), on the verge of migration, *Crs* is extended with the rules in Crs_{migr} as well as in Crs_{toBe}, the latter set at this moment containing Crs_{hib} only, already present in *Crs*. In rule (22), right after the migration, *Crs* is replaced by Crs_{toBe}, by then containing all choreography rules computed by the five *McPhil* plus the two rules in Crs_{hib} already present.

Next we present the consistency rule set Crs_{noHb}, rules (23) to (36), covering the interaction of *McPal* and its five helper McPhil_i.

$$
* \ \mathit{McPal}(\mathit{Evol}) \colon \mathit{Hibernating} \xrightarrow{prepared} \mathit{Migrating} \tag{23}
$$

$$
\begin{aligned}
\mathit{McPal} \colon \mathit{StartMigr} \xrightarrow{create} \mathit{Delegated} \ * \qquad & \\
\mathit{McPal} \colon [\, \mathit{Crs} := \mathit{Crs}_{noHb} + \mathit{Crs}_{hib} \,], & \\
\mathit{McPhil}_1(\mathit{Evol}) \colon \mathit{Passive} \xrightarrow{triv} \mathit{Active}, \ldots, \mathit{McPhil}_5(\mathit{Evol}) \colon \mathit{Passive} \xrightarrow{triv} \mathit{Active}, & \\
\mathit{McPhil}_1 \colon [\, \mathit{Crs}_1 := \mathit{Crs}_{1,asIs} + \mathit{Crs}_{1,orch} \,], \ \ldots, \ \mathit{McPhil}_5 \colon [\, \mathit{Crs}_5 := \mathit{Crs}_{5,asIs} + \mathit{Crs}_{5,orch} \,] & \qquad (24)
\end{aligned}
$$

The set Crs_{noHb} contains the choreography step (23) for *McPal*'s own phase transfer from *Hibernating* to initiate the migration. From then on one finds various orchestration steps conductor *McPal* may take within phase *Migrating*. Rule (24) gets the five helper McPhil_i going, providing each with the local *as-is* choreography as well as with the local orchestration it has to conduct, while *McPal* keeps the rules from Crs_{hib} and Crs_{noHb} only.

The STD of *McPal* in Figure 8 provides eight transitions from state *Delegated* to state *Gathering*. *McPal* takes a transition from its state *Delegated* to the state *Gathering* once all five McPhil_i have reached a 'halfway' trap, either trap *halfwayL* or trap *halfwayR*, in their phase *Active*. Therefore, the figure shows eight different actions, dependent on the various combinations. By coupling a local transition of *McPal* to a global step in the *Evol* role of the McPhil_i,

[2] One may call this behaviour computation, programming in terms of behavioural constraints.

the proper transition is taken by *McPal* and the right continuation for the $McPhil_i$ is prescribed. Below, in the set of consistency rules Crs_{noHb}, we only provide the rules for the actions *doubleDeflexLR* and *singleDeflexRL*, rules (25) and (26), leaving the details of the remaining six rules, rules (27) to (32), to the reader.

$$\begin{aligned} &McPal\colon Delegated \xrightarrow{doubleDeflexLR} Gathering \ * \\ &\quad McPhil_1(Evol)\colon Active \xrightarrow{halfwayR} EndAsL,\ McPhil_2(Evol)\colon Active \xrightarrow{halfwayR} EndAsR, \\ &\quad McPhil_3(Evol)\colon Active \xrightarrow{halfwayR} EndAsL,\ McPhil_4(Evol)\colon Active \xrightarrow{halfwayR} EndAsR, \\ &\quad McPhil_5(Evol)\colon Active \xrightarrow{halfwayR} EndAsR \end{aligned} \tag{25}$$

$$\begin{aligned} &McPal\colon Delegated \xrightarrow{singleDeflexRL} Gathering \ * \\ &\quad McPhil_i(Evol)\colon Active \xrightarrow{halfwayR} EndAsR,\ McPhil_{i+1}(Evol)\colon Active \xrightarrow{halfwayL} EndAsL, \\ &\quad McPhil_{i+2}(Evol)\colon Active \xrightarrow{halfwayL} EndAsR,\ McPhil_{i+3}(Evol)\colon Active \xrightarrow{halfwayL} EndAsL, \\ &\quad McPhil_{i+4}(Evol)\colon Active \xrightarrow{halfwayL} EndAsL \end{aligned} \tag{26}$$

six more rules similar to (25) and (26) (27)–(32)

Rules (25)–(32) deal with the eight scenarios for handling the combined preliminary results from the five helpers $McPhil_i$. In particular, rule (25) for action *doubleDeflexLR* covers the case where all five $McPhil_i$ follow R-order, so two non-neighboring ones of them have to be swapped (back) to L-order, here we choose the first and the third. Similarly, rule (26) covers the case where exactly one $McPhil_i$ follows R-order, so another non-neighboring one has to be swapped to R-order (as yet), here we choose $McPhil_{i+2}$.

$$\begin{aligned} &McPal\colon Gathering \xrightarrow{collect} Gathering \ * \\ &\quad McPhil_i(Evol)\colon EndAsR \xrightarrow{done} Retreating, \\ &\quad McPal\colon [\ Crs := Crs + Crs_{i,toBe}, Crs_i := \emptyset, Crs_{toBe} := Crs_{toBe} + Crs_{i,toBe}\ ] \end{aligned} \tag{33}$$

$$\begin{aligned} &McPal\colon Gathering \xrightarrow{collect} Gathering \ * \\ &\quad McPhil_i(Evol)\colon EndAsL \xrightarrow{done} Retreating, \\ &\quad McPal\colon [\ Crs := Crs + Crs_{i,toBe}, Crs_i := \emptyset, Crs_{toBe} := Crs_{toBe} + Crs_{i,toBe}\ ] \end{aligned} \tag{34}$$

$$\begin{aligned} &McPal\colon Gathering \xrightarrow{close} Content \ * \\ &\quad McPhil_1(Evol)\colon Retreating \xrightarrow{away} Passive,\ \ldots,\ McPhil_5(Evol)\colon Retreating \xrightarrow{away} Passive \end{aligned} \tag{35}$$

$$*\ McPal(Evol)\colon Migrating \xrightarrow{done} Hibernating \tag{36}$$

Rules (33) and (34) incorporate the final local R-order or the final local L-order, respectively, as final choreography part into the two variable sets *Crs* and Crs_{toBe}. Rule (35) passivates the five helpers $McPhil_i$. Rule (36) allows *McPal* to return into hibernation, mirroring rule (23).

Before continuing with the sets of rules and model fragments governed by $McPhil_i$, we come back to the point of mimicking scope effect for the actually global variables $Crs, Crs_i, \ldots$. Our usage of governance instead of ownership is an informal explanation of this mimicking. In particular in rule (24), when *McPal* starts up the five $McPhil_i$ with "their local" variables Crs_i, McPal also gives the five Crs_i their initial values, simultaneously removing the same value parts from its "own local" variable *Crs*. From this point on each $McPhil_i$ governance starts. And indeed, from then on *McPal* leaves the sets Crs_i untouched until, rules (33) and (34), *McPal* is absolutely sure for each $McPhil_i$ separately, $McPhil_i$ will not use its local Crs_i any longer as Crs_i has become a choreography. *McPal* does this by turning the value of Crs_i into the empty set, simultaneously extending its own local *Crs* with $Crs_{i,toBe}$, which contains the "final" value of Crs_i before it became the empty set. The emptiness of the set underlines the ending of scope, as from here on $McPhil_i$ has nothing to govern any longer, its final local result having been delivered.

We continue specifying the sets of rules and model fragments. The set $Crs_{i,orch}$ with the actual adaptation orchestration by $McPhil_i$, comprising the rules (37) to (68) below, follows the STD of Figure 9.

$$McPhil_i\colon Awake \xrightarrow{takeOver} JoiningIn \;*\; McPhil_i\colon [\, Crs_i := Crs_i - Crs_{i,asIs} \,] \tag{37}$$

$$McPhil_i\colon JoiningIn \xrightarrow{conductToL} ToL \;* \tag{38}$$
$$Phil_i(Eater)\colon Disallowed \xrightarrow{request} Disallowed,\; Fork_i(ForLH)\colon Freed \xrightarrow{gone} Claimed$$

$$McPhil_i\colon JoiningIn \xrightarrow{conductToL} ToL \;* \tag{39}$$
$$Fork_i(ForLH)\colon Claimed \xrightarrow{got} Claimed,\; Fork_{i+1}(ForRH)\colon Freed \xrightarrow{gone} Claimed$$

$$McPhil_i\colon JoiningIn \xrightarrow{conductToR} ToR \;* \tag{40}$$
$$Phil_i(Eater)\colon Disallowed \xrightarrow{request} Allowed,\; Fork_{i+1}(ForRH)\colon Claimed \xrightarrow{got} Claimed$$

$$McPhil_i\colon JoiningIn \xrightarrow{conductToR} ToR \;*\; Phil_i(Eater)\colon Allowed \xrightarrow{done} Disallowed, \tag{41}$$
$$Fork_i(ForLH)\colon Claimed \xrightarrow{got} Freed,\; Fork_{i+1}(ForRH)\colon Claimed \xrightarrow{got} Freed$$

$$McPhil_i\colon JoiningIn \xrightarrow{conductToR} ToR \;*\; Phil_i(Eater)\colon Disallowed \xrightarrow{request} Disallowed, \tag{42}$$
$$Fork_i(ForLH)\colon Claimed \xrightarrow{got} Freed,\; Fork_{i+1}(ForRH)\colon Claimed \xrightarrow{triv} Claimed$$

Rule (37) removes the *as-is* choreography. Here, (38)–(41), with $McPhil_i$ in $JoiningIn$, cover the four previous choreography steps of the *as-is* protocol, cf. rules (17)–(20), but now orchestrated, with $McPhil_i$ as conductor. Rules (38) and (39) lead the conductor to state ToL to continue conducting the original L-order; rules (40) and (41) lead the conductor to state ToR to continue according to the new R-order. In these two steps the swap from L-order to R-order is easy as it happens to coincide with stopping to eat or with getting hungry anew. Rule (42) is needed to escape deadlock, a subtlety not further elaborated here.

$$McPhil_i\colon ToR \xrightarrow{conductToR} ToR \;* \tag{43}$$
$$Phil_i(Eater)\colon Disallowed \xrightarrow{request} Disallowed,\; Fork_{i+1}(ForRH)\colon Freed \xrightarrow{gone} Claimed$$

$$McPhil_i\colon ToR \xrightarrow{conductToR} ToR \;* \tag{44}$$
$$Fork_{i+1}(ForRH)\colon Claimed \xrightarrow{got} Claimed,\; Fork_i(ForLH)\colon Freed \xrightarrow{gone} Claimed$$

$$McPhil_i\colon ToR \xrightarrow{conductToR} ToR \;* \tag{45}$$
$$Fork_i(ForLH)\colon Claimed \xrightarrow{got} Claimed,\; Phil_i(Eater)\colon Disallowed \xrightarrow{request} Allowed$$

$$McPhil_i\colon ToR \xrightarrow{conductToR} ToR \;*\; Phil_i(Eater)\colon Allowed \xrightarrow{done} Disallowed, \tag{46}$$
$$Fork_i(ForLH)\colon Claimed \xrightarrow{got} Freed,\; Fork_{i+1}(ForRH)\colon Claimed \xrightarrow{got} Freed$$

four similar rules for cycling in ToL (47)–(50)

Rules (43)–(46), with $McPhil_i$ in ToR, cover the new R-order, basically implementing the *to-be* rules (11)–(14), but conducted by $McPhil_i$ while sojourning in state ToR, waiting for $McPal$'s consent. The symmetric rules (47)–(50), with $McPhil_i$ staying in ToL are suppressed.

$$McPhil_i\colon ToL \xrightarrow{choreofyToL} ToBeL \;* \tag{51}$$
$$Phil_i(Eater)\colon Disallowed \xrightarrow{request} Disallowed,\; Fork_i(ForLH)\colon Freed \xrightarrow{gone} Claimed,$$
$$McPhil_i\colon [\, Crs_i := Crs_i - Crs_{i,orch} + Crs_{i,toBeL},\; Crs_{i,toBe} := Crs_{i,toBeL} \,]$$

$$\mathit{McPhil}_i\colon \mathit{ToL} \xrightarrow{\mathit{choreofyToL}} \mathit{ToBeL} \;*\; \tag{52}$$
$$\mathit{Fork}_i(\mathit{ForLH})\colon \mathit{Claimed} \xrightarrow{\mathit{got}} \mathit{Claimed},\ \mathit{Fork}_{i+1}(\mathit{ForRH})\colon \mathit{Freed} \xrightarrow{\mathit{gone}} \mathit{Claimed},$$
$$\mathit{McPhil}_i\colon [\, \mathit{Crs}_i := \mathit{Crs}_i - \mathit{Crs}_{i,\mathit{orch}} + \mathit{Crs}_{i,\mathit{toBeL}},\ \mathit{Crs}_{i,\mathit{toBe}} := \mathit{Crs}_{i,\mathit{toBeL}} \,]$$
$$\mathit{McPhil}_i\colon \mathit{ToL} \xrightarrow{\mathit{choreofyToL}} \mathit{ToBeL} \;*\; \tag{53}$$
$$\mathit{Fork}_{i+1}(\mathit{ForRH})\colon \mathit{Claimed} \xrightarrow{\mathit{got}} \mathit{Claimed},\ \mathit{Phil}_i(\mathit{Eater})\colon \mathit{Disallowed} \xrightarrow{\mathit{request}} \mathit{Allowed},$$
$$\mathit{McPhil}_i\colon [\, \mathit{Crs}_i := \mathit{Crs}_i - \mathit{Crs}_{i,\mathit{orch}} + \mathit{Crs}_{i,\mathit{toBeL}},\ \mathit{Crs}_{i,\mathit{toBe}} := \mathit{Crs}_{i,\mathit{toBeL}} \,]$$
$$\mathit{McPhil}_i\colon \mathit{ToL} \xrightarrow{\mathit{choreofyToL}} \mathit{ToBeL} \;*\; \mathit{Phil}_i(\mathit{Eater})\colon \mathit{Allowed} \xrightarrow{\mathit{done}} \mathit{Disallowed}, \tag{54}$$
$$\mathit{Fork}_i(\mathit{ForLH})\colon \mathit{Claimed} \xrightarrow{\mathit{got}} \mathit{Freed},\ \mathit{Fork}_{i+1}(\mathit{ForRH})\colon \mathit{Claimed} \xrightarrow{\mathit{got}} \mathit{Freed},$$
$$\mathit{McPhil}_i\colon [\, \mathit{Crs}_i := \mathit{Crs}_i - \mathit{Crs}_{i,\mathit{orch}} + \mathit{Crs}_{i,\mathit{toBeL}},\ \mathit{Crs}_{i,\mathit{toBe}} := \mathit{Crs}_{i,\mathit{toBeL}} \,]$$

Rules (51)–(54), with McPhil_i moving from *ToL* to *ToBeL*, cover the continuation of *L*-order as the *to-be* protocol, starting as orchestration. Additionally, with McPhil_i as conductor, the *L*-order orchestration in Crs_i is immediately replaced by the *L*-order choreography.

$$\mathit{McPhil}_i\colon \mathit{ToL} \xrightarrow{\mathit{choreofyToR}} \mathit{ToBeR} \;*\; \mathit{Phil}_i(\mathit{Eater})\colon \mathit{Disallowed} \xrightarrow{\mathit{request}} \mathit{Disallowed}, \tag{55}$$
$$\mathit{Fork}_i(\mathit{ForLH})\colon \mathit{Freed} \xrightarrow{\mathit{triv}} \mathit{Freed},\ \mathit{Fork}_{i+1}(\mathit{ForRH})\colon \mathit{Freed} \xrightarrow{\mathit{gone}} \mathit{Claimed},$$
$$\mathit{McPhil}_i\colon [\, \mathit{Crs}_i := \mathit{Crs}_i - \mathit{Crs}_{i,\mathit{orch}} + \mathit{Crs}_{i,\mathit{toBeR}}, \mathit{Crs}_{i,\mathit{toBe}} := \mathit{Crs}_{i,\mathit{toBeR}} \,]$$
$$\mathit{McPhil}_i\colon \mathit{ToL} \xrightarrow{\mathit{choreofyToR}} \mathit{ToBeR} \;*\; \tag{56}$$
$$\mathit{Fork}_i(\mathit{ForLH})\colon \mathit{Claimed} \xrightarrow{\mathit{triv}} \mathit{Freed},\ \mathit{Fork}_{i+1}(\mathit{ForRH})\colon \mathit{Freed} \xrightarrow{\mathit{gone}} \mathit{Claimed},$$
$$\mathit{McPhil}_i\colon [\, \mathit{Crs}_i := \mathit{Crs}_i - \mathit{Crs}_{i,\mathit{orch}} + \mathit{Crs}_{i,\mathit{toBeR}},\ \mathit{Crs}_{i,\mathit{toBe}} := \mathit{Crs}_{i,\mathit{toBeR}} \,]$$
$$\mathit{McPhil}_i\colon \mathit{ToL} \xrightarrow{\mathit{choreofyToR}} \mathit{ToBeR} \;*\quad \mathit{Phil}_i(\mathit{Eater})\colon \mathit{Disallowed} \xrightarrow{\mathit{triv}} \mathit{Disallowed}, \tag{57}$$
$$\mathit{Fork}_i(\mathit{ForLH})\colon \mathit{Claimed} \xrightarrow{\mathit{triv}} \mathit{Freed},\ \mathit{Fork}_{i+1}(\mathit{ForRH})\colon \mathit{Claimed} \xrightarrow{\mathit{triv}} \mathit{Claimed},$$
$$\mathit{McPhil}_i\colon [\mathit{Crs}_i := \mathit{Crs}_i - \mathit{Crs}_{i,\mathit{orch}} + \mathit{Crs}_{i,\mathit{toBeR}},\ \mathit{Crs}_{i,\mathit{toBe}} := \mathit{Crs}_{i,\mathit{toBeR}} \,]$$
$$\mathit{McPhil}_i\colon \mathit{ToL} \xrightarrow{\mathit{choreofyToR}} \mathit{ToBeR} \;*\; \tag{58}$$
$$\mathit{Phil}_i(\mathit{Eater})\colon \mathit{Disallowed} \xrightarrow{\mathit{request}} \mathit{Allowed},\ \mathit{Fork}_{i+1}(\mathit{ForRH})\colon \mathit{Claimed} \xrightarrow{\mathit{got}} \mathit{Claimed},$$
$$\mathit{McPhil}_i\colon [\, \mathit{Crs}_i := \mathit{Crs}_i - \mathit{Crs}_{i,\mathit{orch}} + \mathit{Crs}_{i,\mathit{toBeR}},\ \mathit{Crs}_{i,\mathit{toBe}} := \mathit{Crs}_{i,\mathit{toBeR}} \,]$$
$$\mathit{McPhil}_i\colon \mathit{ToL} \xrightarrow{\mathit{choreofyToR}} \mathit{ToBeR} \;*\; \mathit{Phil}_i(\mathit{Eater})\colon \mathit{Allowed} \xrightarrow{\mathit{done}} \mathit{Disallowed}, \tag{59}$$
$$\mathit{Fork}_i(\mathit{ForLH})\colon \mathit{Claimed} \xrightarrow{\mathit{got}} \mathit{Freed},\ \mathit{Fork}_{i+1}(\mathit{ForRH})\colon \mathit{Claimed} \xrightarrow{\mathit{got}} \mathit{Freed},$$
$$\mathit{McPhil}_i\colon [\, \mathit{Crs}_i := \mathit{Crs}_i - \mathit{Crs}_{i,\mathit{orch}} + \mathit{Crs}_{i,\mathit{toBeR}},\ \mathit{Crs}_{i,\mathit{toBe}} := \mathit{Crs}_{i,\mathit{toBeR}} \,]$$
nine rules for leaving *ToR*, similar to rules (51)–(59) (60)–(68)

Rules (55)–(59), with McPhil_i heading for *ToBeR*, cover the orchestrated swapping from *L*-order to *R*-order, thereby replacing it by *R*-order choreography. In particular, (55) covers claiming the first (right) *Fork* without having to undo an earlier claim of the left *Fork*. Contrarily, (56) covers claiming the first (right) *Fork* together with necessary undoing of an earlier claim of the left *Fork*. Notably, rule (57) covers continuing to claim the first (right) *Fork* together with necessary undoing of an earlier claim of the left *Fork*. Thus, (57) provides an escape from deadlock, similar to rule (42). Rules (58) and (59) cover starting and stopping to eat, for the last time as resulting from *L*-order. The symmetric rules (60)–(68) for leaving *ToR* are omitted.

Finally, the rule sets $\mathit{Crs}_{i,\mathit{toBeL}}$ and $\mathit{Crs}_{i,\mathit{toBeR}}$ contain the choreography rules for the *to-be*-situation in *L*-order and in *R*-order, rules (69)–(72) and (73)–(76) respectively.

$$* \;\mathit{Phil}_i(\mathit{Eater})\colon \mathit{Disallowed} \xrightarrow{\mathit{request}} \mathit{Disallowed},\ \mathit{Fork}_i(\mathit{ForLH})\colon \mathit{Freed} \xrightarrow{\mathit{gone}} \mathit{Claimed} \tag{69}$$

$$* \ \mathit{Fork}_i(\mathit{ForLH}) \colon \mathit{Claimed} \xrightarrow{got} \mathit{Claimed}, \ \mathit{Fork}_{i+1}(\mathit{ForRH}) \colon \mathit{Freed} \xrightarrow{gone} \mathit{Claimed} \tag{70}$$

$$* \ \mathit{Fork}_{i+1}(\mathit{ForRH}) \colon \mathit{Claimed} \xrightarrow{got} \mathit{Claimed}, \ \mathit{Phil}_i(\mathit{Eater}) \colon \mathit{Disallowed} \xrightarrow{request} \mathit{Allowed} \tag{71}$$

$$* \ \mathit{Phil}_i(\mathit{Eater}) \colon \mathit{Allowed} \xrightarrow{done} \mathit{Disallowed}, \tag{72}$$
$$\mathit{Fork}_i(\mathit{ForLH}) \colon \mathit{Claimed} \xrightarrow{got} \mathit{Freed}, \ \mathit{Fork}_{i+1}(\mathit{ForRH}) \colon \mathit{Claimed} \xrightarrow{got} \mathit{Freed}$$

$$* \ \mathit{Phil}_i(\mathit{Eater}) \colon \mathit{Disallowed} \xrightarrow{request} \mathit{Disallowed}, \ \mathit{Fork}_{i+1}(\mathit{ForRH}) \colon \mathit{Freed} \xrightarrow{gone} \mathit{Claimed} \tag{73}$$

$$* \ \mathit{Fork}_{i+1}(\mathit{ForRH}) \colon \mathit{Claimed} \xrightarrow{got} \mathit{Claimed}, \ \mathit{Fork}_i(\mathit{ForLH}) \colon \mathit{Freed} \xrightarrow{gone} \mathit{Claimed} \tag{74}$$

$$* \ \mathit{Fork}_i(\mathit{ForLH}) \colon \mathit{Claimed} \xrightarrow{got} \mathit{Claimed}, \ \mathit{Phil}_i(\mathit{Eater}) \colon \mathit{Disallowed} \xrightarrow{request} \mathit{Allowed} \tag{75}$$

$$* \ \mathit{Phil}_i(\mathit{Eater}) \colon \mathit{Allowed} \xrightarrow{done} \mathit{Disallowed}, \tag{76}$$
$$\mathit{Fork}_i(\mathit{ForLH}) \colon \mathit{Claimed} \xrightarrow{got} \mathit{Freed}, \ \mathit{Fork}_{i+1}(\mathit{ForRH}) \colon \mathit{Claimed} \xrightarrow{got} \mathit{Freed}$$

Note, rules (17)–(76) cover all migration trajectories. The rather large number of sixty rules is the consequence of our aim to distribute the migration, thus revealing the distributed potential of the Paradigm-*McPal* tandem for system adaptation by giving freedom to *McPhil*s as delegates. As final remark we note, neither the STDs of *Phil*s and *Fork*s nor their roles *Eater*, *ForLH*, and *ForRH* had to be changed: for this example the migration is fully situated within the coordination of the five ongoing collaborations and, again, *Phil* and *Fork* components remain running while the system migrates, dynamically indeed. From other examples we know in addition, *McPal* can also handle migration in detailed STD dynamics and in role dynamics, see e.g. [8].

6. An architectural view on distributed migration coordination

The set-up of the migration coordination as discussed in Section 4, also sets the stage for the understanding of Section 5. Understanding the technical and cumulative complexities of the collected consistency rules given is not an easy matter, however. The more so, as during an actual migration trajectory, the rules themselves do change depending on the trajectory; we called this *behaviour computation* already. Such a change of rules, via a concrete change clause, determines a change in the Paradigm model at the moment the change clause is executed. This means, in the course of a migration trajectory, a particular sequence of Paradigm models is being computed where the models succeed each other in being *the one Paradigm model currently valid*: from the *as-is* model via several intermediate models to one of the different *to-be* models. Such sequences will be referred to as *model suites*. By representing each model suite through a series of collaboration-like architectural snapshots –one snapshot per representative model from the suite– we aim to clarify the precise structure of the behaviour computation at hand. In addition, such model suites and their corresponding series of collaborations are most helpful in getting model checking of migrations organized.

As the change clauses that are relevant for the model changes in the model suites are formulated in terms of sets of consistency rules as defined at the start of Section 5, we repeat those sets here, both the constant sets and the variable ones.

$\mathit{Crs}_{i,asIs}$ ::= *as-is* *L*-order choreography of $\mathit{Phil2Forks}_i$ (as in Section 2)

$\mathit{Crs}_{i,orch}$ ::= orchestration conducted by McPhil_i

$\mathit{Crs}_{i,toBeL}$ ::= possible *to-be* choreography of $\mathit{Phil2Forks}_i$ in *L*-order

$\mathit{Crs}_{i,toBeR}$ ::= possible *to-be* choreography of $\mathit{Phil2Forks}_i$ in *R*-order

Crs_{hib} ::= orchestration conducted by *McPal* during phase *Hibernating*

Crs_{noHb} ::= orchestration conducted by *McPal* during phase *Migrating*

Crs_{migr} ::= Crs_{noHb} (Crs_{migr} from rule (5) here coincides with Crs_{noHb})

Crs_{asIs} ::= $\mathit{Crs}_{hib} + \mathit{Crs}_{1,asIs} + \cdots + \mathit{Crs}_{5,asIs}$

Actually, the above table contains an extra constant, viz. Crs_{asIs}, not occurring in the consistency rules at all but useful for the explanation. The above sets are constants, the sets below vary during the migration.

Crs ::= varying orchestration/choreography, not governed by $McPhil_i$ but by *McPal*

Crs_i ::= varying $McPhil_i$-governed rule set for $Phil2Forks_i$, both starting and ending as $\emptyset$

$Crs_{i,toBe}$ ::= either $Crs_{i,toBeL}$ or $Crs_{i,toBeR}$, being the *to-be* choreography of $Phil2Forks_i$ (as in Section 3)

Crs_{toBe} ::= growing from Crs_{hib} to $Crs_{hib} + Crs_{1,toBe} + \cdots + Crs_{5,toBe}$

Amid these numerous sets, a reader should concentrate most on the variable sets Crs and Crs_i, as they are the sets of rules, which actually define parts of the model and of the model dynamics a particular component is responsible for. More specifically, the set Crs always refers to the part of the currently valid model which is being governed by *McPal* and, at that moment, not by any $McPhil_i$. Contrarily, for each $i = 1, \ldots, 5$, the set Crs_i is the currently valid model part which is being governed by $McPhil_i$. Hence set Crs_{migr} contains the rules for the orchestration conducted by *McPal* when not inside phase *Hibernating*, but it does not contain the rules for the orchestrations conducted by the various $McPhil_i$. Another consequence is, each set Crs_i has the following value life cycle.

1. the empty set $\emptyset$ of rules, as long as $McPhil_i$ has not yet really started;
2. $Crs_{i,asIs} + Crs_{i,orch}$, as $McPhil_i$ has just started and taken over the governing of the local choreography $Crs_{i,asIs}$ from *McPal*, having its conducting of the local orchestration $Crs_{i,orch}$ installed, ready for instant use;
3. $Crs_{i,orch}$, no more local choreography, only keeping its own conducting of local orchestration;
4. $Crs_{i,toBe}$, no more local conducting, only keeping the local *to-be* choreography;
5. again the empty set $\emptyset$ of rules, as $McPhil_i$ has just quit by handing over the governing of the local choreography to *McPal*.

The life-cycle of the local Crs_i values sketched above can also be observed from architectural snapshots concerning the various model suites. Below we introduce and discuss these snapshots in the order of the corresponding models in a suite.

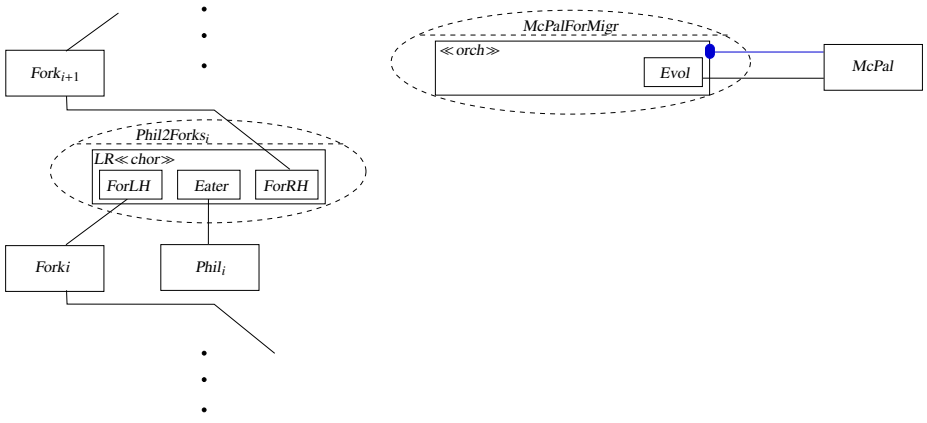

Figure 10: Collaboration before migration.

The first architectural snapshot gives the complete collaborations for the *as-is* situation, see Figure 10. The collaborations are in UML 2.0-style, with some Paradigm flavour, e.g. where UML stereotypes refer to Paradigm protocols: ≪*chor*≫ to choreography and ≪*orch*≫ to orchestration. As one can see, the snapshot comprises the five different collaborations $Phil2Forks_i$ known from Figure 7a, extended with collaboration *McPalForMigr*. It is worthwhile to observe, how two collaborations $Phil2Forks_i$ and $Phil2Forks_{i+1}$ are connected via component $Fork_{i+1}$ contributing a separate role to each of the two collaborations. Contrarily, collaboration *McPalForMigr* is as yet completely disconnected from the other five collaborations, reflecting that component *McPal* still is in hibernation, as migration has not yet started. Note, *McPal* contributes its *Evol* role to collaboration *McPalForMigr* and *McPal* moreover is conductor of the same collaboration *McPalForMigr*. Although collaboration *McPalForMigr* is completely disconnected from the five choreography collaborations $Phil2Forks_i$, it is set Crs where the rules for the five choreographies reside, governed by *McPal*.

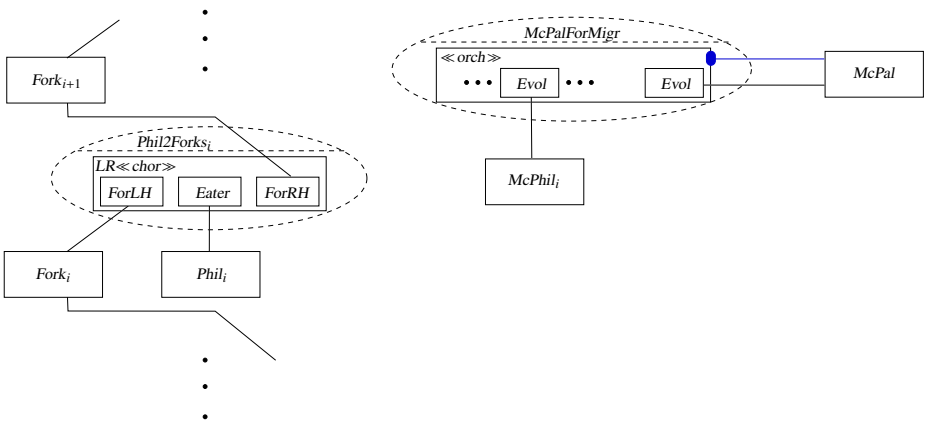

Figure 11: Collaboration at the start of migration, before delegation.

The second architectural snapshot in Figure 11 gives the complete collaborations relevant for the Paradigm model resulting from applying consistency rule (21), in particular from its change clause $Crs := Crs + Crs_{migr} + Crs_{toBe}$. Here, (i) the old value of *Crs* equals Crs_{asIs}, comprising the original choreographic rules (17)–(20) being $Crs_{1,asIs} + \ldots + Crs_{5,asIs}$ for the five $Phil2Forks_i$ collaborations as well as the rules (21)–(22) being Crs_{hib} for *McPal* while hibernating; (ii) Crs_{migr} is the set Crs_{noHb} of rules for *McPal* while migrating; (iii) Crs_{toBe} has the set Crs_{hib} as starting value. This last value might seem a bit meager, but one should keep in mind how at the end of the migration the set of rules Crs_{toBe} has to contain all rules for the *to-be* situation: those for *McPal* while hibernating plus the five local *to-be* choreographies of collaborations $Phil2Forks_i$. Thus, in Figure 11 we see the five components $McPhil_i$ appearing together with one *Evol* role each. But, as the sets Crs_i start with each being empty, the five $McPhil_i$ do not yet own their conducting responsibility. Also, the rules for five *as-is* choreographies still reside in *Crs* governed by *McPal*.

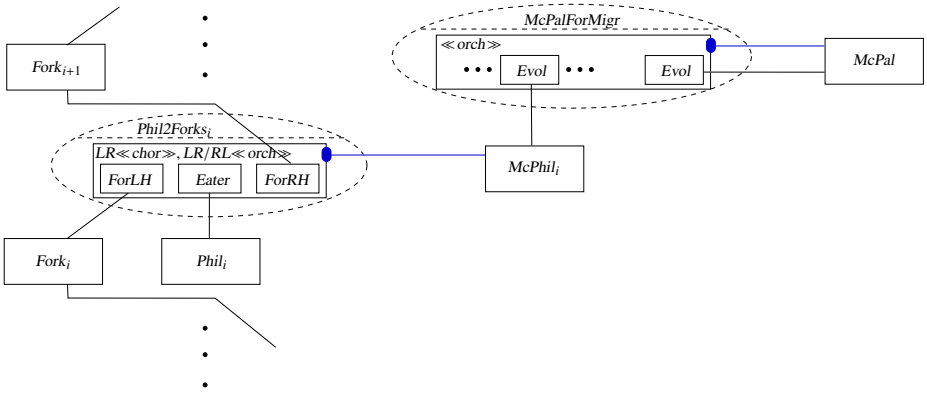

Figure 12: Collaboration in the beginning of migration, at the start of delegation.

The third architectural snapshot in Figure 12 results from the application of consistency rule (24) and more in particular from its six change clauses $Crs := Crs_{noHb} + Crs_{hib}$ and $Crs_i := Crs_{i,asIs} + Crs_{i,orch}$. As the six change clauses belong to the same consistency rule, they are being executed simultaneously. This then means, (i) *Crs* contains the rules where *McPal* is conducting or has an *Evol* role step; (ii) the five *as-is* choreographies $Crs_{i,asIs}$ are being removed from *Crs*; (iii) the same five *as-is* choreographies are being added each, simultaneously as well as distributedly, to the five sets Crs_i respectively; (iv) also the five orchestrations $Crs_{i,orch}$ are being added each to the five sets Crs_i respectively. Any of the five orchestrations added, covers both orders of taking forks, *LR*- as well as *RL*-order,

together with the relevant swappings from one to the other. Thus, in Figure 12 we see the five $McPhil_i$ connected each to collaboration $Phil2Forks_i$ as conductor. As each set Crs_i does not only contain the orchestration rules $Crs_{i,orch}$ but also the *as-is* choreography rules $Crs_{i,asIs}$, the protocol of $Phil2Forks_i$ covers the orchestration in both orders as well as the *as-is* choreography, explicitly indicated in the figure too.

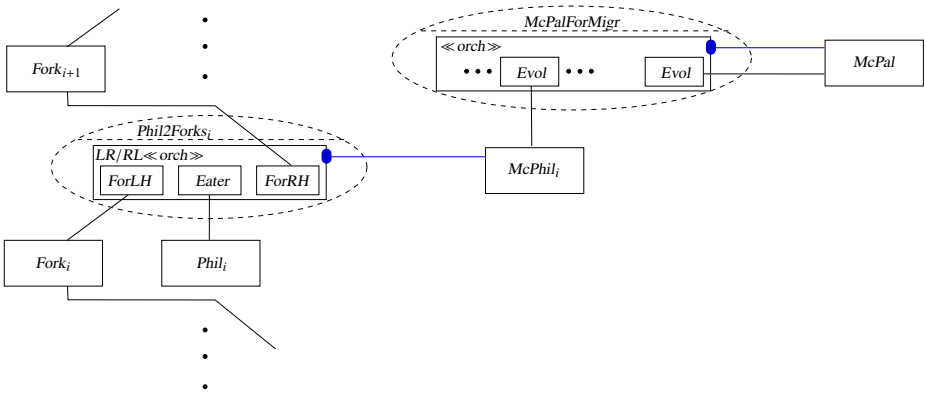

Figure 13: Collaboration when delegation is ongoing.

The fourth architectural snapshot in Figure 13 results from application of consistency rule (37) and more in particular from its change clause $Crs_i := Crs_i - Crs_{i,asIs}$. By taking the step from state *Awake* to *JoiningIn*, the newly created component $McPhil_i$ performs its first contribution in view of guiding the protocol for collaboration $Phil2Forks_i$, not by conducting a protocol step, but by governing the local model specification, changing it from a mixture of choreography and orchestration into pure orchestration for both orders of fork taking, *LR* and *RL*, and for the swapping between them. This then means, the local choreography $Crs_{i,asIs}$ is removed from Crs_i, hence Crs_i contains the local orchestration $Crs_{i,orch}$ only. Thus, in Figure 13 we see the new stereotype $LR/RL\ll orch\gg$ indicated for the protocol inside collaboration $Phil2Forks_i$ no longer containing $LR\ll chor\gg$. Please note, the snapshot suggests that each $McPhil_i$ has applied rule (37). As the $McPhil_i$ do this on their own, not necessarily simultaneously, the snapshot from Figure 13 actually results from 5 model changes after the snapshot from Figure 12, done in arbitrary order.

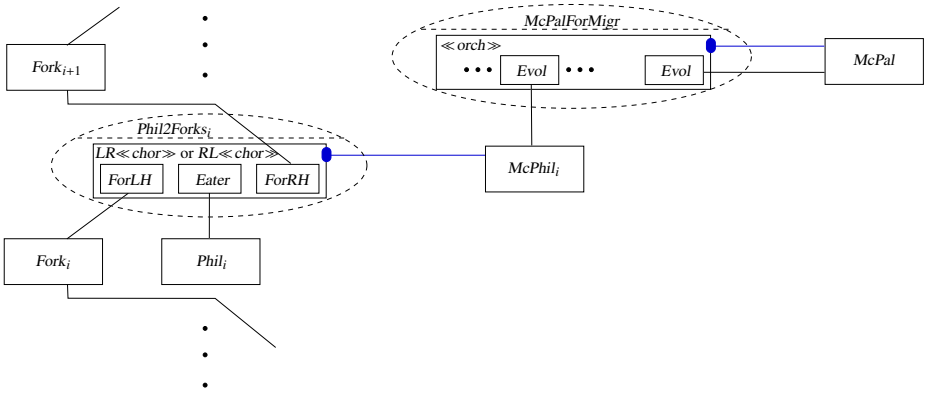

Figure 14: Collaboration when delegation is ready, before being stopped.

The fifth architectural snapshot in Figure 14 results from the application of one of the consistency rules (51)–(68). More in particular the fifth snapshot results either from the two change clauses $Crs_i := Crs_i - Crs_{i,orch} + Crs_{i,toBeR}$ and $Crs_{i,toBe} := Crs_{i,toBeR}$ or from the two change clauses $Crs_i := Crs_i - Crs_{i,orch} + Crs_{i,toBeL}$ and $Crs_{i,toBe} := Crs_{i,toBeL}$. Please note the difference between the indices *toBeR* and *toBeL*. By taking precisely one of the steps specified

through rules (51)–(68), $McPhil_i$ performs either action *choreofyToL* or action *choreofyToR*, thereby removing the local orchestration $Crs_{i,orch}$ from Crs_i as well as adding the final local choreography to it, being either $Crs_{i,toBeL}$ or $Crs_{i,toBeR}$. As *McPal* is not taking over the governance of the final local choreography yet, for the moment this is stored in Crs_i. But in preparation of later taking over of such governance by *McPal*, the same choreography is also stored into the auxiliary variable $Crs_{i,toBe}$. Thus, in Figure 14 we once more see a new stereotype, now $LR\ll chor\gg$ or $RL\ll chor\gg$, indicated for the protocol of collaboration $Phil2Forks_i$: pure choreography be it either in *LR*-order or in *RL*-order. Also here the snapshot from Figure 14 results from five model changes after the snapshot from Figure 13, done in arbitrary order.

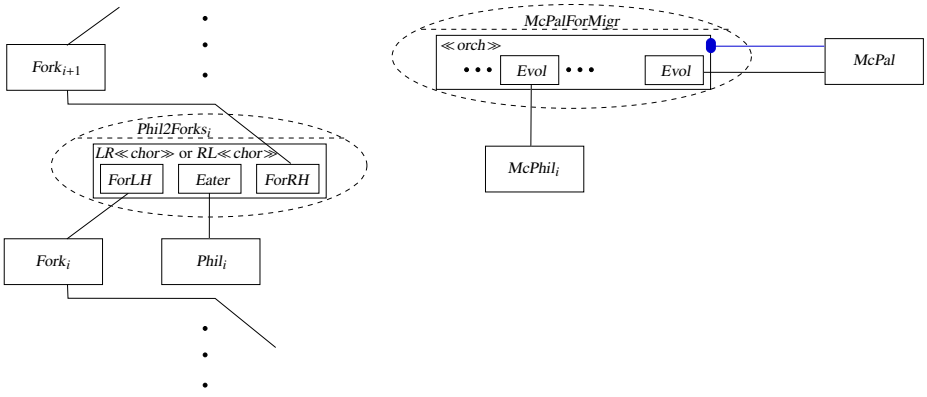

Figure 15: Collaboration at the end of migration after stopping delegation.

The sixth architectural snapshot in Figure 15 results from application of one of the consistency rules (33)–(34) and more in particular from its two change clauses $Crs := Crs + Crs_{i,toBe}$ and $Crs_i := \emptyset$. By actually taking, for each value of i separately, a step specified through either rule (33) or (34), *McPal* gradually moves the governance of each final local choreography from the specific $McPhil_i$ to itself, thereby simultaneously making set Crs_i empty and extending *Crs*; please note, the value of $Crs_{i,toBe}$ is exactly the last nonempty value of Crs_i. Moreover, through the third change clause $Crs_{toBe} := Crs_{toBe} + Crs_{i,toBe}$ in rules (33)–(34), *McPal* like-wise gradually builds the value of Crs_{toBe}. Thus, in Figure 15 we see each $McPhil_i$ returning as disconnected from collaboration $Phil2Forks_i$, highly similar to Figure 11, but now with a new *to-be* choreography in place, either in *LR*-order or in *RL*-order. Once more, the snapshot from Figure 15 results from five model changes after the snapshot from Figure14, done in arbitrary order.

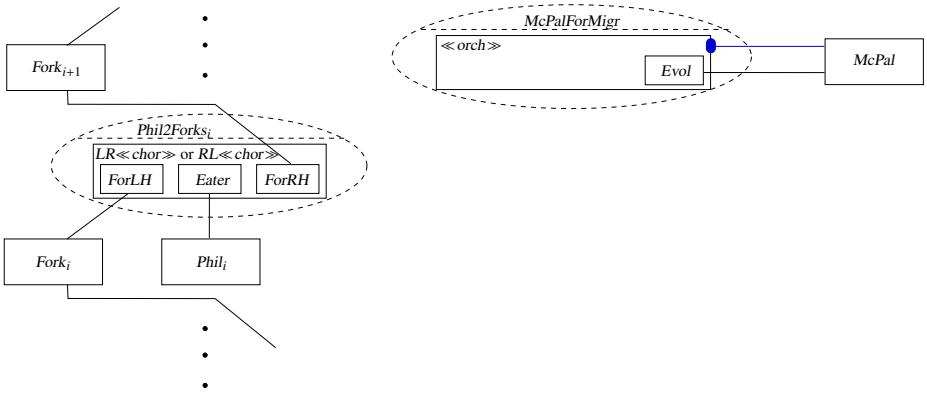

Figure 16: Collaboration after migration.

The seventh and last architectural snapshot in Figure 16 results from application of consistency rule (22) and more in particular from its change clause $Crs := Crs_{toBe}$. By actually taking the step specified through rule (22),

McPal replaces the model specification it governs, as kept in *Crs*, by Crs_{toBe}. Thus *McPal* removes every bit of model specification regarding migration, precisely keeping $Crs_{toBe} = Crs_{hib} + Crs_{1,toBe} + \cdots + Crs_{5,toBe}$ as *McPal* has built this value earlier through five times applying either one of the rules (33)–(34); remember Crs_{toBe}'s starting value to be Crs_{hib}, the hibernation rules for *McPal* only. Thus, in Figure 16 we see the five $McPhil_i$ and their *Evol* roles have disappeared, leaving *McPal* in complete isolation, highly similar to Figure 10, but with a new *to-be* choreography in place, either in *LR*-order or in *RL*-order.

The above seven architectural snapshots from Figures 10–16 actually hide an enormous amount of model suites. Restricting the counting to strict interleaving of subsequent model changes as well as to not discriminating between *LR*-order and *RL*-order, the step from the snapshot in Figure 12 to the snapshot in Figure 13 has $5! = 120$ orders of five subsequent model changes. Similarly, the step from the snapshot in Figure 13 to the snapshot in Figure 14 and also the next step to the snapshot in Figure 15 also have $5! = 120$ orders of five subsequent model changes. Already this makes $5! \times 5! \times 5! = 1,728,000$ model suites, each of length of $1 + 1 + 1 + 5 + 5 + 5 + 1 = 19$ subsequent models. Again, this result is apart from accidental simultaneity of 2 or 3 or 4 or even 5 model changes (among the first two cases of the 5! orders) as well as apart from 10 different final combinations of the five *to-be* choreographies being possible.

7. Model checking the migration model

Paradigm, as described in the previous sections, supports dynamic model adaptation via change clauses. The specification language `mCRL2` in use, on the other hand, does not allow modification of the specification during the execution. In that sense the specification is static, no new process behaviour can be created or added. Thus, the dynamics of the execution boils down to selection of the next action to be executed, out of a set of enabled action alternatives in a current state, and to the execution thereof. To be able to capture the dynamic model evolution, as present in the Paradigm model of migration, the static `mCRL2` specification, first of all, has to cover any possible behaviour of the system in any possible situation, or in terms of the model suite of Section 6, during the migration. As the system behaviour is defined by the behaviour of its components and the way they interact, this means that our `mCRL2` specification has to contain sub-specifications of each component for every possible model suite during the migration, as well as the sub-specifications of the communication between components at any point in the execution, which as mentioned, dynamically changes during the migration. Second, the specification should rely on a mechanism which mimics dynamic model changes, by enabling more, less or simply a different set of actions alternatives. For instance, in Subsection 3.2 process `IProtRules` has been used as a simple mechanism which selects the orientation of a *Phil* by enabling only a selected set of actions process `Phil` can execute.

The dynamic changes during the migration add another dimension of complexity too. As indicated already, the number of possible migration trajectories and model suites for the migration of the dinning philosophers example is relatively large. Therefore, any mechanism to be used has to be detached from a concrete migration trajectory and concrete set of rules and should operate at a higher level of abstraction. But still, it should neither exclude any of the possible model suites nor any of the possible migration trajectories.

The architectural view of the Paradigm migration model discussed in Section 6 turned out to be very beneficial and served as a stepping stone to a structured specification of the migration model in `mCRL2`. It indicates different turning points at which our mechanism for dynamic adaptation should trigger the changes, and which processes are to be influenced by these changes. It also reflects the idea that the changes are distributed: some changes are global (relevant for all components), while others are local (influencing a particular set of components only, such as a philosopher and both forks assigned to her).

The `mCRL2` specification of the Paradigm migration model includes a (recursive) process specification for each component in the Paradigm model. Thus, we have the following processes, sixteen in total: $\texttt{Phil}_i$, $\texttt{Fork}_i$, $\texttt{McPhil}_i$ and `McPal`. Their process specifications are derived from their Paradigm STDs and the consistency rules, in a way similar as explained in Subsection 3.2. Nevertheless, while migrating the components have a much richer behaviour. This is not to be concluded from the corresponding STDs, but is to be expected from the precision required for the description of the communication. The rather long list of consistency rules does not only capture the interaction between the components but also the contribution of each component per consistency rule. For instance, the specifi-

cation $\texttt{Phil}_i$ of Phil_i is structured according to the STDs in Figure 3, but the specification also describes every action through which Phil_i is participating in the consistency rules (17)–(76).

While the specification of the overall behaviour is obtained from the STDs and the consistency rules, the dynamic part of the behavioural changes is modeled by means of extra processes, called *protocol rule processes*, and is specified according to the Paradigm model architecture. Protocol rule processes (i) keep track of the currently active sub-specifications of the components, the current model suite, and (ii) enable or disable accordingly the activities and interaction, thus navigating the system to the next model from the suite. In the concrete migration model, inspired by the architectural view of the model, we distinguish six different protocol rule processes. As it is to be expected from the analysis and classification in Section 6, we have identified one general protocol rule process `GProtRules` which is the counterpart of the change clauses conducted by *McPal*, and five more individual protocol rule processes $\texttt{IProtRules}_i$, distributed over `Prots`. These five processes are counterparts to the change clauses conducted by the five McPhil_i.

In Table 4 the mCRL2 process specification of `GProtRules` is shown. The process has one parameter `grs` which assumes values from the set `GRuleSet = {Hib, Start, Halfway, End}`. The four values of `grs` correspond to the four different collaborations the system is smoothly going through during the migration, again, from the global point of view. They are properly aligned to the migration architecture (and the snapshots) of Section 6, for instance, `grs == Hib` corresponds to the collaboration in Figure 10, and `grs` keeps value `Halfway` as long as the collaborations of Figures 12, 13, 14 and 15 are in place. Thus, the parameter `grs` governs the consistency rules that can be invoked and couples their activity to a specific stage of the migration.

```
proc GProtRules(grs:GRuleSet)  =
    ( grs == Hib ) → (
        rule21ok .  GProtRules(Start) ) +
    ( grs == Start ) → (
        rule23ok .  GProtRules() + rule24ok .  GProtRules(Halfway) ) +
    ( grs == Halfway ) → (
        rule25ok .  GProtRules() + rule26ok .  GProtRules() +
        rule27ok .  GProtRules() + rule28ok .  GProtRules() +
        rule29ok .  GProtRules() + rule30ok .  GProtRules() +
        rule31ok .  GProtRules() + rule32ok .  GProtRules() +
        ( sum protid:ProtID . rule33ok(protid) .  GProtRules() ) +
        ( sum protid:ProtID . rule34ok(protid) .  GProtRules() ) +
        rule35ok .  GProtRules() + rule36ok .  GProtRules() +
        rule22ok .  GProtRules(End) ) +
    ( grs == End ) → ( delta );
```

Table 4: mCRL2 specification of `GProtRules` process.

The distributed character of the model yields distributed dynamic changes over individual protocols (a philosopher, the two assigned forks and possibly the corresponding *McPhil*) controlled by processes $\texttt{IProtRules}_i$. This process steers the protocol $\texttt{Prot}_i$ through the migration by changing the communication between $\texttt{Phil}_i$, $\texttt{McPhil}_i$, $\texttt{Fork}_i$ and $\texttt{Fork}_{i+1}$ accordingly and dynamically. The moments changes on $\texttt{Prot}_i$ are triggered are easily extracted from the architectural view, and as such integrated in the specification of process $\texttt{IProtRules}_i$. The changes themselves, also integrated in the specification, are induced from the migration strategy encoded in the consistency rules (also discussed in Section 6). Part of the specification of `IProtRules` is given in Table 5.[3]

[3] See `http://www.win.tue.nl/~evink/research/paradigm.html` for the full mCRL2 specification.

```
proc IProtRules(i:ProtID,irs:IRuleSet)  =
    ( irs == IAsIs ) → (
        rule17ok(i) .  IProtRules() + ··· + rule20ok(i) .  IProtRules() +
        rule37ok(i) .  IProtRules(i, McPhilHasTakenOver) ) +
    ( irs == McPhilHasTakenOver ) → (
        rule38ok(i) .  IProtRules(i,McPhilChoseL) +
        rule39ok(i) .  IProtRules(i,McPhilChoseL) +
        rule40ok(i) .  IProtRules(i,McPhilChoseR) +
        rule41ok(i) .  IProtRules(i,McPhilChoseR) +
        rule42ok(i) .  IProtRules(i,McPhilChoseR) ) +
    ( irs == McPhilChoseL ) → (
        rule47ok(i) .  IProtRules() + ··· + rule50ok(i) .  IProtRules() +
        rule51ok(i) .  IProtRules(i,ToBeLSet) + ··· +
                rule54ok(i) .  IProtRules(i,ToBeLSet) +
        rule55ok(i) .  IProtRules(i,ToBeRSet) + ··· +
                rule59ok(i) .  IProtRules(i,ToBeRSet) ) +
    ( irs == McPhilChoseR ) → (
        rule43ok(i) .  IProtRules() + ··· + rule46ok(i) .  IProtRules() +
        rule60ok(i) .  IProtRules(i,ToBeRSet) + ··· +
                rule63ok(i) .  IProtRules(i,ToBeRSet) +
        rule64ok(i) .  IProtRules(i,ToBeLSet) + ··· +
                rule68ok(i) .  IProtRules(i,ToBeLSet) ) +
    ( irs == ToBeLSet ) → (
        rule69ok(i) .  IProtRules() + ··· + rule72ok(i) .  IProtRules() +
    ( irs == ToBeRSet ) → (
        rule73ok(i) .  IProtRules() + ··· + rule76ok(i) .  IProtRules() );
```

Table 5: Part of the mCRL2 specification of IProtRules process.

The process specification has two parameters: the parameter i, which is the protocol identifier, and irs which takes values from the set IRuleSet={ IAsIs, McPhilHasTakenOver, McPhilChoseL, McPhilChoseR, ToBeLSet, ToBeRSet }. The six values of this parameter correspond to the six different collaborations the protocol $\texttt{Prot}_i$ is smoothly going through locally. The initial value irs = IAsIs corresponds to the first period of the migration changes when no local, but only global changes are made, captured in the snapshots in Figures 10, 11 and 12. As can be seen in Table 5, process IProtRules allows execution of action rule37 which can be interpreted as closing the *as-is* behaviour and opening the migration behaviour. The change clause $Crs_i := Crs_i - Crs_{i,asIs}$ in this rule, in the specification is represented by disabling any *as-is* action (from the choreography collaboration), and instead, by updating irs to McPhilHasTakenOver, enabling the first bundle of new actions containing all possible first steps of the first migration collaboration (Figure 14), now under $McPhil_i$'s conducting. In a similar way, with any next update of irs to McPhilChoseL or McPhilChoseR, and thereafter to ToBeLSet or ToBeRSet, always a new set of actions becomes allowed, while the obsolete ones become disallowed. As expected, no other changes should be made once the *to-be* behaviour is established for $\texttt{Prot}_i$. Thus the last value of irs is either ToBeLSet or ToBeRSet. It is worth noticing that the IProtRules specification clearly describes that in the end $\texttt{Prot}_i$ executes only the actions rule69 to rule72 if it has migrated to the *LR*-order protocol, or executes only rule73 to rule76 if it has migrated to the

RL-order protocol. However, this observation is informal, and in the remainder of this section we formally verify that this is indeed the case.

The challenge of the formal verification of the Paradigm migration model has been the tuning of the mCRL2 representation of the model and model changes, in particular to be able to conveniently express the properties of the migration. The previous subsection focused on our solution for achieving the first aspect. In this subsection we focus on the formulation and verification of the properties that together confirm the correctness of the migration. We restrict to a selection of the analysis results we obtained that are the most interesting and relevant for the discussion here.

Deadlock and starvation freedom. In Section 3.2 we discussed μ-calculus formula expressing deadlock freedom and starvation freedom. A difference for the isolated *as-is* and *to-be* models and the complete migration models is that a philosopher's wish to eat as expressed by its action *getHungry* is observed through different synchronized actions in different stages of the system evolution. We have the action rule17 at the starting stage; the trap *request* being reached represents at the global level the execution of *getHungry* at the local level. With the relevant $McPhil_i$ as conductor, it is rule36 or rule40 during the first orienting steps of $McPhil_i$ that catches the trap *request* being reached and, once $McPhil_i$ has chosen a direction of picking up the forks, it is rule43 or rule47 for *L*-orientation and *R*-orientation, respectively, that takes care of this. After *McPal* has assigned a final orientation and $McPhil_i$ is stepping out as conductor such is done by rule51, rule55, rule58 or rule60, rule64, rule67, dependent on a swap being needed or not and the current phases of the forks. In the final stage of the migration, the *to-be* situation, a request to eat can be observed through the action rule69 or rule73. Apart from this, deadlock freedom and starvation freedom for the migration model are implied by a guarantee that the *to-be* situation will always be reached, to be discussed in a minute, together with deadlock freedom and starvation freedom for the *to-be* situation, as established in Section 3.2. However, subtleties arise regarding the exact matching of requests to eat and permission to do so, as we will address below.

In view of the above we started out to check that the migration model is deadlock free by model checking the deadlock-free property (see Subsection 3.2). Related to this, we have shown the significance of the rule (25) and its *halfwayL* counterpart rule (32) (not explicitly given in the set of rules) for the migration, by showing that exclusion of any of these two rules leads to a deadlock during migration. Contrasting, exclusion of any of the rules (26)–(31) does not cause a deadlock in the migration model, but gives rise to non-termination of the migration. Intuitively, all five protocols $\texttt{Prot}_i$ remain active in the collaborations conducted by $McPhil_i$ (Figure 13), and no action can move them forward. We have confirmed as well that if rule (42) or rule (57) is excluded from the migration model deadlock does occur.

Migration termination analysis. Termination of the migration is definitely a mandatory property of the migration model. From the model and its process specification we see that the migration starts as soon as *McPal* makes the *giveOut* step, which corresponds to the execution of action rule21 in the mCRL2 process specification. The migration is closed by the step *cleanUp* of *McPal*, i.e., by the execution of rule22 action in the mCRL2 process. Thus, it is required that for a migration trajectory that has started, i.e. once rule21 has occurred, eventually action rule22 is also executed. Of course relying on the fairness assumption that every action which is infinitely often enabled, is eventually executed. The termination of migration is expressed by the following μ-calculus formula:

```
[ tau*.rule21.tau* ] < tau*.rule22 > true
```

In the specification for which this formula is verified all actions, except rule21 and rule22, are renamed into the internal tau action. Therefore, by fairness, the formula says that for every path on which rule21 occurs, after finitely many tau steps rule22 will occur. The pattern '`[ ... rule21.tau* ] < tau*.rule22 ... >`' guarantees that all possible fair paths are inspected.

Behaviour before and after the migration. The migration is initiated by *McPal* when rule21 action is executed. But before this action is triggered, the five protocols are supposed to follow *as-is* behaviour. This means that only protocol rules actions rule17–rule20 happen before rule21 action has taken place in all possible migration trajectories. The

following formula expresses the opposite of this, for brevity here given for the case of two philosophers:

```
< (!rule21)*.(
        !rule17(Prot1) && !rule18(Prot1) && !rule19(Prot1) && !rule20(Prot1) &&
        !rule17(Prot2) && !rule18(Prot2) && !rule19(Prot2) && !rule20(Prot2)
                ).(!rule21)*.rule21 > true
```

The formula states that there exists an execution of the specification in which an action different from `rule17` to `rule20` is executed before `rule21`. The model checker returns *false*, meaning that no such path is possible for our model.

Complementary, execution of `rule22` marks the closing of the migration, after which only protocol rules of the *to-be* situation are allowed, thus only actions `rule69`–`rule76` may happen. The formula

```
[ true*.rule22.true* ] [ tau ||
        rule69(Prot1) || rule70(Prot1) || rule71(Prot1) || rule72(Prot1) ||
        rule73(Prot1) || rule74(Prot1) || rule75(Prot1) || rule76(Prot1) ||
        rule69(Prot2) || rule70(Prot2) || rule71(Prot2) || rule72(Prot2) ||
        rule73(Prot2) || rule74(Prot2) || rule75(Prot2) || rule76(Prot2)    ] true
```

is confirmed by the model checker to be correct for our model. It states that, here for the case of the protocol with two philosophers, in any execution in which `rule22` has occurred, at any point after its appearance, only the actions `rule69` to `rule76` or the action `tau` are executed. The latter action `tau` represent the local steps, that were hidden in the specification that was model checked.

Seamless migration. The Paradigm model for the adaptation describes a smooth evolution of the system from the source *as-is* solution to the target *to-be* solution. The transient behaviour of the system, along any possible migration trajectory, is thus realized by the model. In that sense, the Paradigm model does not only allow us to inspect the source and the target behaviour, but also allows to inspect the adaptation process as a whole.

In the concrete case of the dining philosophers, one want to have that each philosopher, who requested so, is eventually allowed to eat. Moreover, one would like to preserve this property during the migration, independently of how far the migration has progressed and irrespective of the migration trajectory taken. In that sense, a request to eat in, e.g., the *as-is* phase – execution of action `rule17`, can be met in the *as-is* phase (action `rule20`), or at a later moment during migration, or even in the *to-be* phase. Seamless migration is exactly reflected in the properties like this. Thus, focusing on the first philosopher, we want to establish

```
[ true*.rule17(Prot1) ]
        < true*.( rule20(Prot1) || rule41(Prot1) || rule46(Prot1) || rule50(Prot1) ||
                  rule54(Prot1) || rule59(Prot1) || rule63(Prot1) || rule68(Prot1) ||
                  rule72(Prot1) || rule76(Prot1) ) > true
```

From the model we extracted all the consistency rules which denote that eating has finishing for the first philosopher. These are rules (20), (41), (46), (50), (54), (59), (63), (68), (72) and (76). Any of the corresponding actions in the `mCRL2` specification can match the request made by $Phil_1$ in the *as-is* phase when she executes action `rule17`, the actual match being dependent on the migration trajectory taken. Modulo fairness, the property above states that the request in the *as-is* phase will always eventually be met.

The above property is indeed confirmed by the model checker. However, we can go a bit further than this. Rather than restricting to one philosopher at the time, we quantify over all interaction protocols. Using a shorthand for the conjunction

```
!rule20(Proti) && !rule41(Proti) && !rule46(Proti) && !rule50(Proti) && !rule54(Proti) &&
!rule59(Proti) && !rule63(Proti) && !rule68(Proti) && !rule72(Proti) && !rule76(Proti)
```

we refine the formula to

```
forall Proti : ProtID . [ true*.rule17(Proti).( !rule20(Proti) && ... && !rule76(Proti) )* ]
        < true*.(rule20(Proti) || rule41(Proti) || rule46(Proti) || rule50(Proti) ||
                 rule54(Proti) || rule59(Proti) || rule63(Proti) || rule68(Proti) ||
                 rule72(Proti) || rule76(Proti) ) > true
```

This property states that whenever a request to eat by the philosopher in `Proti`, observed as the action `rule17(Proti)`, is pending, at some stage during the migration the request will be granted, which is observed as one of the actions `rule20(Proti)` to `rule76(Proti)`. Again, the model checker confirms the validity of the formula. Although action `rule20(Proti)` can be left out from the disjunction, none of the actions `rule41(Proti)` to `rule76(Proti)` can be omitted from the disjunction

```
rule20(Proti) || ... || rule76(Proti)
```

since then the formula would no longer hold. Thus, (i) progress of the *Phil*s depends on progress of the *McPhil*s, and (ii) matching of the request in the *as-is* situation to eat represented by the action `rule17` is not necessarily done in the *as-is* situation, i.e. by the action `rule20`, but can be upholded and settled during migration, by way of one of the actions `rule41`, . . ., `rule68`, or even postponed until the *to-be* situation has been reached, done by the action `rule72` or `rule76`.

8. Discussion and concluding remarks

In the setting of component-based system development, we have addressed dynamic system adaptation avoiding any form of quiescence. We actually address adaptation as coordination. As coordination is seamless, adaptation is too. By using the coordination modeling language Paradigm, in combination with the special component *McPal*, we particularly underlined the suitability of the approach for dynamic adaptation in a distributed manner. The distributed potential of the Paradigm-*McPal* tandem is our main result, actually revealed through delegation among helpers. Concrete form to the distributive aspect is given via the dining philosophers example: letting the system adapt itself from a rather bad solution (deadlock) to a substantially better one having neither deadlock nor starvation. Additionally, we have formally verified for the particular dining philosophers case the correctness of the migration. In the context of the example, the distributed character of the adaption produces another three new results as spin-off, all three showing a wider reach of the approach: (i) creation/deletion of STDs, (ii) adaptation with self-healing, (iii) behaviour computation. We elaborate on the three of them first.

In line with the coordination features offered by Paradigm, distribution of adaptation is achieved through delegation. Moreover, as adaptation is towards an originally unforeseen *to-be* solution, delegation thereof is brought into action by *McPal*. This results in concrete delegation to originally unforeseen components $McPhil_i$, one per collaboration $Phil2Forks_i$. As the *McPhil* components exist neither at the time the *as-is* solution is ongoing with *McPal* in hibernation nor at the time the *to-be* solution is ongoing with *McPal* in hibernation, in this case we model both STD creation and STD deletion in Paradigm, at the start and at the end of *McPal*'s non-hibernating phase *Migrating*, respectively. Modeling creation and deletion is achieved by simulating it via the phases of the various $McPhil_i(Evol)$ roles: creation of $McPhil_i$ when bringing it to life by leaving phase *Passive*; deletion of $McPhil_i$ when taking its life by returning to phase *Passive*. This way, STDs for components and for their roles can easily be created and deleted in a dynamically consistent manner, as all this comes down to suitable coordination.

As explained at the start of Section 4, coordinating adaptation, referred to as migration, is being modeled in state *JITting* such that different *to-be* situations can be reached, possibly through different migration trajectories. Accordingly, the migration model distributes the migration coordination among five helper $McPhil_i$, with the initial aim of locally achieving a reasonable result. Then *McPal*, by centrally collecting the partial results and comparing them in state *Delegated*, redistributes additional, specific alignment directives among the same five helper $McPhil_i$. After execution of the directives, final results are gathered and compiled into the particular *to-be* solution arising from the distributed migration coordination effort. The self-healing aspect, explicitly present in this example, lies in the activities occurring in state *Delegated* in view of selecting one out of eight outgoing action-transitions to state *Gathering*: rules (25)–(32) specify which particular alignment has to be done. The selection decision is the self-healing: it is solely based on trap information, certain combinations of five *halfwayL* vs. *halfwayR* traps having been entered. This means, it is solely based on intermediate migration results. Only in case of the two actions *goAheadLR* or *goAheadRL* the self-healing is empty; in the six other cases there is at least one adjustment from *L*-order to *R*-order or vice versa, and often two. Please note, such adjustments indeed arise on-the-fly of the still ongoing migration. Also interesting to note is, the self-healing directives are given at the level of *McPal*, the self-healing directives are performed at the (lower) delegation level of the five helper $McPhil_i$, very much in line with the architectural ideas in [33].

The above form of self-healing is finalized in *McPal*'s state *Gathering*. There the final *to-be* model is compiled into Crs_{toBe}, through composition of smaller model fragments composed to that aim by each helper $McPhil_i$. Fragments are about behaviour, so their composition certainly is behaviour computation, at the level of *McPal* as well as at the level of each $McPhil_i$. Thus, our behaviour computation is a distributed computation.

Another interesting feature of the example is the seamless zipping of a conductor into a choreography, turning it into an 'equivalent' orchestration. Conversely, the seamless zipping of a conductor out of an orchestration, turns it into the 'equivalent' choreography. In this perspective, the temporary conductor $McPhil_i$ is reminiscent to the notion of a 'scaffold' in [42]. In our example, through the additional *Evol* role of a conductor $McPhil_i$, the scaffold has additional flexibility, changing phase-wise, while the model remains ongoing during alterations as usual.

We exploited the translation of Paradigm into `mCRL2` for the formal verification of the migration. In particular, we proposed dedicated processes that manage the consistency rules that can be executed within a protocol at a specific moment during migration. We have shown that the proposed migration always leads from the deadlock-prone to the deadlock-free and starvation-free situation, while the philosophers continue their thinking and eating. By maintaining a one-one correspondence between the distribution of coordination and the process rule processes, the `mCRL2` specification faithfully represents the dynamic system adaptation as modeled in Paradigm.

As one might have observed, quite some redundancy appears in the above. (i) Paradigm has it in the role concept, repeating essence of component dynamics in view of *exogenous coordination* via consistency rules. (ii) Two roles per *Fork* introduce even more redundancy in view of architectural separation of five collaborative concerns. Behavioural redundancy is present too, organized in line with the five collaborations $Phil2Forks_i$: (iii) After any helper $McPhil_i$ has communicated its *partial result*, it possibly has to undo the partial result. Or (iv) $McPhil_i$ possibly does essentially nothing, as partial result and local *as-is* as well as local *to-be* collaboration remain unchanged (left fork first, as always). This means, within the *environment* of the other four ongoing collaborations, a single $McPhil_i$'s behaviour computation *robustly* meanders towards its final result instead of going there straightforwardly.

During the final panel session at FACS 2010, where a preliminary version of this work was presented, the above four italicized characteristics –robust instead of correct, environment as first class citizen, exogenous coordination, partial results– have been positioned [11] as crucial for service-orientation in comparison to component technology. See also [12]. They reflect the additional flexibility service-orientation has to offer, when taking the next step from component technology. In Paradigm, these characteristics arise from redundancy designed on purpose: in language, in model structure and in model dynamics.

With respect to the overall approach taken we can say the following. One starts from an *as-is* situation, which is pure coordination. Then, later, a *to-be* situation is modeled as pure coordination too. Note that the architecture of the *to-be* situation can be quite different from the *as-is* architecture. Given the starting and end point of possible migrations the modeler has to decide via which intermediate stages the *as-is* situation gradually should transform into the *to-be* situation. In principle, *McPal* is coordinating this, possibly supported by helpers like $McPhil_i$. When in hibernation *McPal* is ready for coordinating all kinds of migrations of Paradigm models depending on the intermediate models developed in *JITting*. Therefore, Paradigm's approach to dynamic adaptation by means of *McPal* is generic indeed.

Due to Paradigm's way of grouping consistency rules into protocols, coordination is treated modularly. Therefore the approach scales quite well [43, 45, 39, 46]. However, this can be propagated only limitedly: `mCRL2` translations of our Paradigm models do suffer from state space explosion. Paradigm's architecture of the migration is of some help here as it suggests a factorization of the system during migration. By focusing on the stages of the migration and abstracting the `mCRL2` specification accordingly, smaller state spaces are being dealt with. More work is needed to study this more systematically. In particular, we plan to investigate the correspondence between change clauses triggering model changes and relevant slices of state spaces.

Recently, the Paradigm-*McPal* tandem has been deployed within Edafmis [44]. The ITEA-project Edafmis aims at innovative integration of ICT-support from different advanced imaging systems into non-standard medical intervention practice, such that all flexibility needed during such interventions can be sustained smoothly and quickly, adequately and pleasantly. Particularly, the possibility for distributed migrations, as presented here, is of great value.

Acknowledgment. We thank the reviewers for their constructive feedback and helpful suggestions which assisted in improving the readability and positioning of the paper.

References

[1] R. Adler, I. Schaefer, M. Trapp, and A. Poetzsch-Heffter. Component-based modeling and verification of dynamic adaptation in safety-critical embedded systems. *ACM Transaction on Embedded Computing Systems*, 10(2), 2011. 20pp.

[2] M. Alia, S. Hallsteinsen, N. Paspallis, and F. Eliassen. Managing distributed adaptation of mobile applications. In J. Indulska and K. Raymond, editors, *Proc. DAIS 2007*, pages 104–118. LNCS 4531, 2007.

[3] R. Allen, R. Douence, and D. Garlan. Specifying and analyzing dynamic software architectures. In E. Astesiano, editor, *Proc. FASE 1998*, pages 21–37. LNCS 1382, 1998.

[4] S. Andova, L.P.J Groenewegen, and E.P. de Vink. Distributed adaption of dining philosophers. In L.S. Barbosa and M. Lumpe, editors, *Proc. FACS 2010*, pages 125–144. LNCS 6921, 2012.

[5] S. Andova, L.P.J. Groenewegen, J. Stafleu, and E.P. de Vink. Formalizing adaptation on-the-fly. In G. Salaün and M. Sirjani, editors, *Proc. FOCLASA 2009*, pages 23–44. ENTCS 255, 2009.

[6] S. Andova, L.P.J. Groenewegen, J.H.S. Verschuren, and E.P. de Vink. Architecting security with Paradigm. In R. de Lemos, J.-C. Fabre, C. Gacek, F. Gadducci, and M.H. ter Beek, editors, *Architecting Dependable Systems VI*, pages 255–283. LNCS 5835, 2009.

[7] 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 2008*, pages 3–20. ENTCS 229(2), 2009.

[8] S. Andova, L.P.J. Groenewegen, and E.P. de Vink. Towards dynamic adaptation of probabilistic systems. In T. Margaria and B. Steffen, editors, *Proc. ISOLA 2010*, pages 143–159. LNCS 6416, 2010.

[9] S. Andova, L.P.J. Groenewegen, and E.P. de Vink. Dynamic consistency in process algebra: From Paradigm to ACP. *Science of Computer Programming*, 76(8):711–735, 2011.

[10] S. Andova, L.P.J. Groenewegen, and E.P. de Vink. Towards reduction of paradigm coordination models. In L. Aceto and M.R. Mousavi, editors, *Proc. PACO 2011, Reykjavik*, pages 1–18. EPTCS 60, 2011.

[11] F. Arbab, 2010. Personal communication.

[12] F. Arbab. Will the real service oriented computing please stand up? In L.S. Barbosa and M. Lumpe, editors, *Proc. FACS 2010*, pages 277–285. LNCS 6921, 2012.

[13] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Number 18 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

[14] M. Bartoletti, E. Tuosto, and R. Zunino. Contracts in distributed systems. In A. Silva, S. Bliudze, R. Bruni, and M. Carbone, editors, *Proc. ICE*, pages 130–147. EPTCS 59, 2011.

[15] N. Bencomo, P. Sawyer, G.S. Blair, and P. Grace. Dynamically adaptive systems are product lines too: Using model-driven techniques to capture dynamic variability of adaptive systems. In *Proc. DSPL 2008*, pages 23–32. Limerick, 2008.

[16] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1/3):109–137, 1984.

[17] K.N. Biyani and S.S. Kulkarni. Assurance of dynamic adaptation in distributed systems. *Journal of Parallel Distributed Computing*, 68:1097–1112, 2008.

[18] L. Bocchi, K. Honda, E. Tuosto, and N. Yoshida. A theory of design-by-contract for distributed multiparty interactions. In P. Gastin and F. Laroussinie, editors, *Proc. CONCUR*, pages 162–176. LNCS 6269, 2010.

[19] L. Bocchi, J. Lange, and E. Tuosto. Amending contracts for choreographies. In A. Silva, S. Bliudze, R. Bruni, and M. Carbone, editors, *Proc. ICE*, pages 111–129. EPTCS 59, 2011.

[20] J.S. Bradbury, J.R. Cordy, J. Dingel, and M. Wermelinger. A survey of self-management in dynamic software architecture specifications. In D. Garlan, J. Kramer, and A.L. Wolf, editors, *Proc. WOSS 2004*, pages 28–33. ACM, 2004.

[21] J. Bradfield and C. Stirling. Modal mu-calculi. In *Handbook of Modal Logic*, volume 3 of *Studies in Logic and Practical Reasoning*, pages 721–756. Elsevier, 2007.

[22] A. Bucchiarone, P. Pelliccione, C. Vattani, and O. Runge. Self-repairing systems modeling and verification using AGG. In *Proc. WICSA/ECSA 2009, Cambridge*, pages 181–190. IEEE, 2009.

[23] C. Cetina, J. Fons, and V. Pelechano. Applying software product lines to build autonomic pervasive systems. In *Proc. SPLC 2008, Limerick*, pages 117–126. IEEE, 2008.

[24] H. Ehrig, C. Ermel, O. Runge, A. Bucchiarone, and P. Pelliccione. Formal analysis and verification of self-healing systems. In D. Rosenblum and G. Taentzer, editors, *Proc. FASE 2010*, pages 139–155. LNCS 6013, 2010.

[25] G. Engels, J.M. Küster, R. Heckel, and L. Groenewegen. A methodology for specifying and analyzing consistency of object-oriented behavioral models. In *Proc. FSE'01*, pages 186–195. ACM, 2001.

[26] H. Giese. Modeling and verification of cooperative self-adaptive mechatronic systems. In F. Kordon and J.Sztipanovits, editors, *Proc. of Reliable Systems on Unreliable Networked Platforms*, pages 258–280. LNCS 4322, 2007.

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

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

[29] J.F. Groote, A.H.J. Mathijssen, M.A. Reniers, Y.S. Usenko, and M.J. van Weerdenburg. The formal specification language mCRL2. In E. Brinksma, D. Harel, A. Mader, P. Stevens, and R. Wieringa, editors, *Methods for Modelling Software Systems*. IBFI, Schloss Dagstuhl, 2007. 34pp.

[30] J.F. Groote and M.A. Reniers. Algebraic process verification. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, pages 1151–1208. Elsevier, 2001.

[31] C. Koehler, F. Arbab, and E.P. de Vink. Reconfiguring distributed Reo connectors. In A. Corradini and U. Montanari, editors, *Proc. WADT 2008, Revised Selected Papers*, pages 221–235. LNCS 5486, 2009.

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

[33] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In L.C. Briand and A.L. Wolf, editors, *Proc. FOSE 2007*, pages 259–268. IEEE, 2007.
[34] C. Krause. *Reconfigurable Component Connectors*. PhD thesis, Leiden University, 2011.
[35] J. Küster. *Consistency Management of Object-Oriented Behavioral Models*. PhD thesis, University of Paderborn, 2004.
[36] J. Magee and J. Kramer. Dynamic structure in software architectures. *SIGSOFT Software Engineering Notes*, 21:3–14, 1996.
[37] T. Melliti, P. Poizat, and S.B. Mokhtar. Distributed behavioural adaptation for the automatic composition of semantic services. In J.L. Fiadeiro and P. Inverardi, editors, *Proc. FASE 2008*, pages 146–162. LNCS 4961, 2008.
[38] B. Morin, F. Fleurey, N. Bencomo, J.-M. Jézéquel, A. Solberg, V. Dehlen, and G. Blair. An aspect-oriented and model-driven approach for managing dynamic variability. In K. Czarnecki, I. Ober, J.-M. Bruel, A. Uhl, and M. Völter, editors, *Proc. MoDELS 2008*, pages 782–796. LNCS 5301, 2008.
[39] P.J.A. Morssink. *Behaviour Modeling in Information Systems Design: Application of the Paradigm Language*. PhD thesis, Leiden University, 1988.
[40] O. Rawashdeh and J.E. Lumpp Jr. Run-time behavior of Ardea: A dynamically reconfiguring distributed embedded control architecture. In *Proc. Aerospace Conference 2006, Big Sky, Montana*. IEEE, 2006.
[41] M.-T. Segarra and F. André. A distributed dynamic adaptation model for component-based applications. In I. Awan, M. Younas, T. Hara, and A. Durresi, editors, *Proc. AINA 2009*, pages 525–529. IEEE, 2009.
[42] A.W. Stam. *Interaction Protocols in PARADIGM*. PhD thesis, LIACS, Leiden University, 2009.
[43] M. van Steen. *Modeling Dynamic Systems by Parallel Decision Processes*. PhD thesis, Leiden University, 1988.
[44] C.J. Stettina, L.P.J. Groenewegen, and B.R. Katzy. Towards flexibility and dynamic coordination in computer-interpretable enterprise process models. In R. Poler, G. Doumeingts, B. Katzy, and R. Chalmeta, editors, *Enterprise Interoperability V, Part 2*, pages 105–115. Springer, 2012.
[45] W. Stut. *Constructing Large Conceptual Models with Movie*. PhD thesis, Leiden University, 1992.
[46] P.J. Toussaint. *Integration of Information Systems: a Study in Requirements Engineering*. PhD thesis, Leiden University, 1998.
[47] M. Yarvis, P. Reiher, and G.J. Popek. Conductor: A framework for distributed adaptation. In *Proc. HOTOS 1999, Rio Rico*, pages 44–51. IEEE, 1999.
[48] J. Zhang, H.J. Goldsby, and B.H.C. Cheng. Modular verification of dynamically adaptive systems. In K.J. Sullivan, A. Moreira, C. Schwanninger, and J. Gray, editors, *Proc. AOSD 2009*, pages 161–172. ACM, 2009.