Dynamic Evolution by Constraint Orchestration (position paper)

Luuk Groenewegen[1,⋆] and Erik de Vink[2]

[1] FaST Group, LIACS, Leiden University
[2] Formal Methods Group, Eindhoven University of Technology

Abstract. Consistency between collaborating components is notoriously difficult, even more so when a collaboration or the components are going to evolve dynamically, without any form of quiescence. For foreseen, non-migrating collaborations, the coordination modelling language Paradigm tackles the dynamic consistency problems by means of constraint orchestration. In case of migration, originally unforeseen dynamics occur, easily leading to (more) inconsistency. Within the setting of migrating Paradigm models, the special component McPal takes care of the consistency for dynamically evolving collaborations too, by explicitly coordinating originally unforeseen migration. The result applies both to software and its architecture and to business processes.

1 Problem Situation

Many software systems are large and complex. Moreover, software systems have a strong tendency to grow over time, in size and complexity. In order to deal with size and complexity, software architectures are used. A software architecture provides a global description of an actually far more detailed software system by giving an overview of the system in terms of *components* and *links*. Components are a system's main parts relevant for the overview, links are the relevant connections between them. Figure 1 presents two architectural visualizations

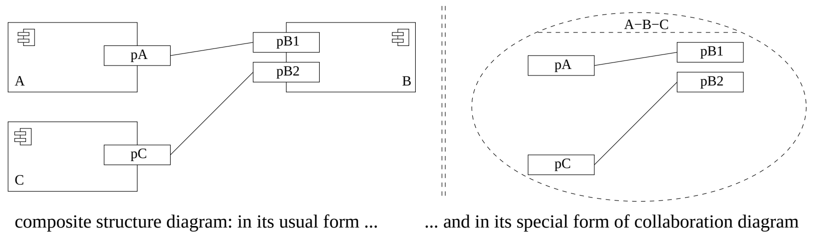
A
pA
pB1
pB2
B
C
pC
A–B–C
pA
pB1
pB2
pC
composite structure diagram: in its usual form ...
... and in its special form of collaboration diagram

Fig. 1. Two different (UML) composite structure diagrams.

in UML-style. The more usual, fully structural style is at the left. The larger, iconized rectangles are *components*. The smaller rectangles, positioned across a components border, are *ports*, each one representing an interface offered by that

⋆ corresponding author: `luuk@liacs.nl`

component. A port serves as the scene of action for a role the component is responsible for in the architectural constellation. Lines connecting ports are *links*, via which components communicate by executing their roles. At the right of Figure 1, a less usual, more dynamically oriented presentation is given. It visualizes a *collaboration*: a grouping of roles constituting a cooperative unit, a protocol. The roles are represented via their respective ports; components remain hidden.

An architecture serves as a basis for the global understanding of the system in terms of coherence between its constituents: components, ports/roles, links and collaborations/protocols. The coherence covers the structural fitting of the four constituent types, but, more importantly, also their dynamic fitting. In particular, each constituent type has its own type of dynamics: A) For a component it is local, internal component behaviour, (usually) hidden. B) For a port it is local, external role behaviour, i.e. visible. C) For a link it consists of the sending and receiving sequences in either direction, role interaction. D) For a collaboration it is the coordination of the role behaviours and their interactions into an overall protocol. We refer to these four types of dynamics as type A, B, C and D, respectively. In view of mutual dynamic fitting of behavioural specifications of the different types, coherence between such specifications is of utmost relevance. In Figure 2 three situations are being indicated, T1–T3, where coherence between dynamics of the above four types A, B, C and D has to be assured. In line with [7], we call such a situation a dynamic consistency problem type.

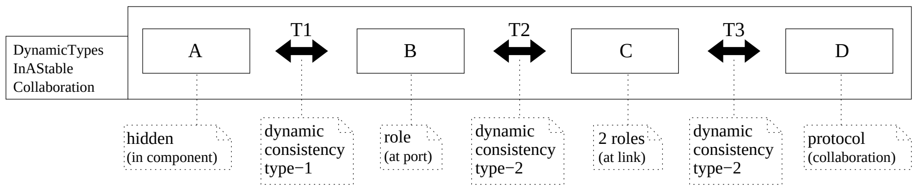

Fig. 2. Three dynamic consistency problem types in a stable UML collaboration.

In addition to the dynamics and consistency relevant for the foreseen situation of regular, stable collaboration, there is migration from the original, foreseen situation towards the initially unforeseen situation of a new collaboration. During migration in particular, a behavioural specification of any type A–D can change. But this should occur smoothly: consistent with what preceded as well as what is to come next. So behavioural execution of every type should be deflected sufficiently gradually. Figure 3 visualizes additional dynamic consistency problem types. T4–T7, resulting from migration. A primary question then is, how can the above four dynamic types be represented such that behavioural realizations remain coherent during their complete execution, satisfying dynamic consistency problem types T1–T3. In addition, such coherent execution should not only last during the original, stable situation, but also during the migration as well as during the new situation, uninterruptedly that is, be it changing grad-

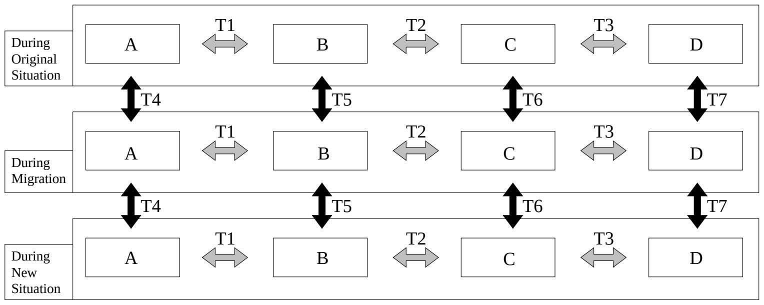

Fig. 3. Four more dynamic consistency problem types in case of migration.

ually. So, executions should continue while remaining consistent without any halting or other form of quiescence.

As solution of the above consistency problems, we propose the coordination modelling language Paradigm [4, 6] together with the special component McPal. A Paradigm model without McPal specifies coordination of stable, foreseen collaborations by dynamically restricting and re-enabling parts of component behaviour through their roles and protocols. In this manner, a Paradigm model provides coherence of type T1–T3 between dynamics of type A, B, C and D. In view of future, unforeseen migration of a Paradigm model as-is, McPal is to be incorporated into it. Component McPal, on the basis of the Paradigm notions for coordination, is designed as follows. First, McPal allows for extending the constraints and their dynamic compositions while keeping the execution of the system going as-is. Second, McPal coordinates migration from the as-is execution to the to-be execution aimed at on the basis of the new constraints and their new orchestration recently added. Third and last, once the execution situation aimed at has been established, McPal removes constraints and orchestrations no longer needed.

2 Solution Proposed: Foreseen Coordination

This section introduces the coordination modelling language Paradigm in terms of two behavioural constraint types, *subprocess* and *trap*, and of two types of dynamically composing such constraints, *global process* and *consistency rule*. Underlying these four notions, Paradigm has the notion of *process*. A *process* in its detailed form specifies hidden component behaviour: type A dynamics. It is visualized by an STD (state transition diagram). Although not always necessary, its transitions are usually labeled by actions. A *subprocess* is a constrained process: a constraint on a detailed process, imposed from elsewhere; the underlying detailed process is referred to as an employee process. A subprocess of a process only specifies a part of the behaviours of the underlying process. A *trap* of a subprocess is a constraint on it, committed to autonomously within the subprocess: by entering a certain subset of its states (the actual trap) which cannot be left by means of the subprocess' behaviours.

A *global process* is a sequential composition of constraints on the same detailed process, alternating between an imposed subprocess (a global state) and a trap committed to (labeling a global transition), actually leading into the subsequently imposed subprocess. A global process always occurs at a port of the component whose hidden dynamics are the employee process underlying this global process, so it specifies type B behaviour. Moreover, a global process brings forward type C behaviour as follows: information about a trap committed to is being sent from the port where the type B behaviour is occurring towards the other end of the link and information about a subprocess imposed is being received at the port where the type B behaviour is occurring from the other end of the same link. Thus, a mirrored version of the global behaviour is actually occurring at that other end of the same link, modulo possible delays between sending from one end and receiving at the other.

It is on the basis of protocol roles mirrored elsewhere (type C dynamics), a *consistency rule* synchronizes certain mirrored steps of them, thus parallelizing protocol roles and moreover coupling them to one step of a detailed, so-called manager process. The possible sequences of thus parallelized steps constitutes a protocol, type D dynamics. The consistency rules, suitably sequentialized, constitute the actual coordination of the collaboration by cleverly composing the relevant constraints. We call this *constraint orchestration.* On the basis of the Paradigm definitions and construction rules, the dynamic consistency problem types of the stable, foreseen coordination situation are solved in the context of a Paradigm model. Moreover, an easy to understand visualization of such coordination can be given, suggesting such coherence convincingly. Figure 4 presents a small example of a so-called Critical Section Management (CSM) problem.

Figure 4's upper layer gives the participants of the CSM collaboration CSM-Coll: three Worker components competing for permission to enter their critical section. The Scheduler component gives the permission to a Worker asking for it, exclusively; the permission is withdrawn only after the Worker indicates it does not need it any longer. The middle layer gives the processes (STDs) for the three Workers and the Scheduler. A Worker needs the permission for being in its state $Crit$ where it does its critical activity. In $Post$ it does some wrapping up, in $Free$ is does nothing in particular, in NCr it does non-critical work and in Pre it prepares its critical activity and as long as it does not have the permission to go to $Crit$, it waits there. In UML-style, the black dot with outgoing edge indicates the starting state $Free$. Likewise, process Scheduler starts in $Chck_1$. In a state $Chck_i$, Scheduler checks whether Worker$_i$ wants to have the permission. If so, it goes to Ask_i where it gives the permission to Worker$_i$; if not so, it goes to the next state $Chck_{i+1}$ in round robin fashion. In Ask_i it waits until Worker$_i$ does not need the permission any longer.

Figure 4's lowest layer visualizes constraints: three subprocesses $OutCS$, $OutBlck$ and $InCS$, each as partial STD of a Worker; four traps: *triv* of $OutCS$, $notYet$ and *started* both of $OutBlck$ and *done* of $InCS$, each drawn as poly-

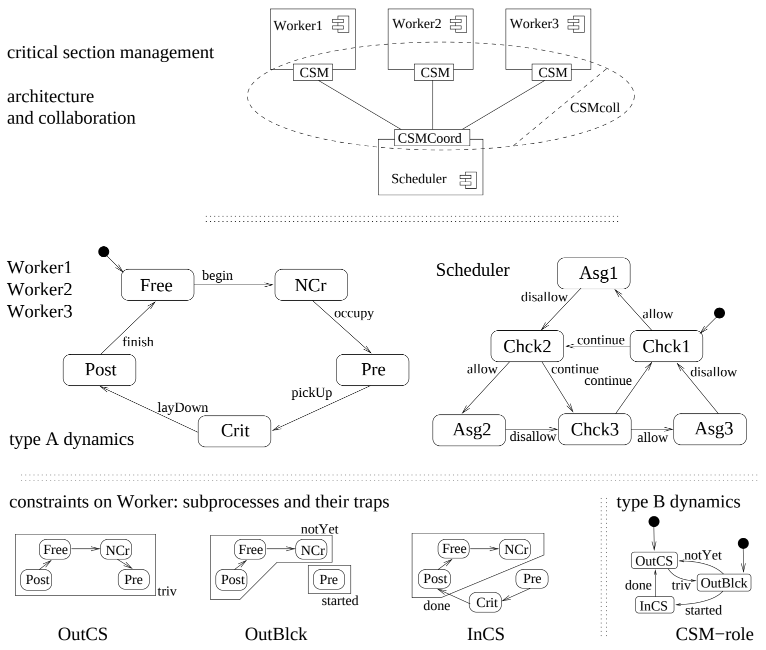

Fig. 4. CSM collaboration: Scheduler, 3 Workers, their 3 CSM-roles.

gon around the states it consists of. Being a commit within the subprocess, a trap once entered cannot be left as long as the subprocess constraint holds. The subprocesses and traps reflect the following intuition. Subprocess $OutCs$ covers the behavioural phase where a $Worker$ does not have the permission to enter its critical section. $OutCS$ reflects, it is as if state $Crit$ does not exists. Contrarily, $InCS$ covers having that permission. $InCS$ reflects, state $Crit$ can be entered, but once only, as state Pre is unreachable during this behavioural phase. Subprocess $OutBlck$ is an interrupted form of $OutCS$, as entering state Pre is no longer possible during it: either trap *started* has been entered or trap $notYet$, the former entering then is taken as CSM-permission request, the latter is taken as no CSM-permission request yet. The trivial trap *triv* of $OutCS$ reflects, $OutCS$ can be interrupted - towards subprocess $OutBlck$ - unconditionally; trap *done* of $InCS$ reflects, the CSM-permission can be withdrawn as it is no longer needed: state $Crit$ has been left. At the right of Figure 4's lowest level, the three global processes Worker$_i$(CSM) are given. Note, each can start only without having the CSM-permission, either in uninterrupted form $OutCS$ or in interrupted form $OutBlck$. Moreover, having the permission (in $InCS$) can occur only after, during the interrupted form of not having it (in $OutBlck$), it turns out trap *started* has been entered, which means a request for the CSM-permission is standing.

Synchronized (and mirrored, by the way) realizations of the three global behaviours, one for each Worker, constitute a protocol realization or scenario. Figure 5 gives such a scenario in the form of a UML-like activity diagram: three swimlanes for the three global processes coupled to one swimlane for Scheduler

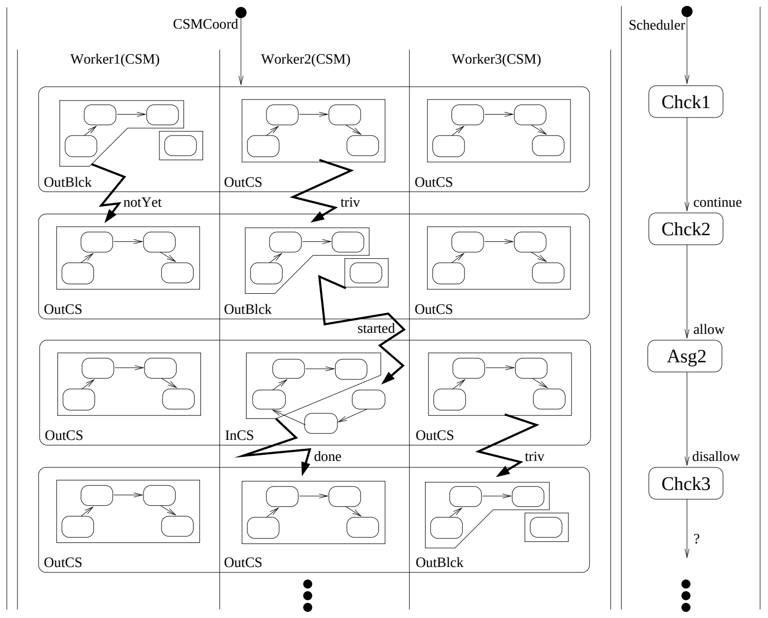

Fig. 5. CSM protocol: subprocess constraints enforced, trap constraints checked.

as Paradigm manager of the protocol, observing it (or even controlling it, but not in this example). In particular, our visualization reveals the behavioural consequences of the various subprocess constraints for the detailed behaviours after each protocol step. More specifically one sees: that, how and when the CSM-permission is given exclusively to one Worker at a time, indeed in round robin order, which can be considered as reasonably fair. Consistency rules specifying the separate protocol steps as visualized in Figure 5, synchronizing global process steps and coupling them to one detailed manager step, are as follows.

$$\begin{array}{lrcl}
Scheduler: & Chck_i & \overset{\text{allow}}{\rightarrow} & Asg_i \; * \\
\quad Worker_i(CSM): & OutBlck & \overset{\text{started}}{\rightarrow} & InCS \\
Scheduler: & Asg_i & \overset{\text{disallow}}{\rightarrow} & Chck_{i+1} \; * \\
\quad Worker_i(CSM): & InCS & \overset{\text{done}}{\rightarrow} & OutCS, \\
\quad Worker_{i+1}(CSM): & OutCS & \overset{\text{triv}}{\rightarrow} & OutBlck \\
Scheduler: & Chck_i & \overset{\text{continue}}{\rightarrow} & Chck_{i+1} \; * \\
\quad Worker_i(CSM): & OutBlck & \overset{\text{notYet}}{\rightarrow} & OutCS, \\
\quad Worker_{i+1}(CSM): & OutCS & \overset{\text{triv}}{\rightarrow} & OutBlck
\end{array}$$

In fact, Figure 5's first step illustrates the third rule: a detailed Scheduler transition from a *Chck* state –where Worker$_i$ is being checked– to a next *Check* state is coupled to two global process transitions: one for process Worker$_i$(CSM), returning from being interrupted in *OutBlck* to not having the permission in *OutCS* as trap *notYet* had been entered; the other for the next process Worker$_{i+1}$(CSM), changing from not having the permission in *OutCS* to being interrupted in *OutBlck* as trap *triv* had been entered trivially. Similarly, Figure 5's second step illustrates the first rule and Figure 5's third step illustrates the second rule.

In order for a rule to be applied, Paradigm's definitions in addition require: the one detailed transition mentioned in the rule, occurs in every currently valid subprocess constraint as specified by the various current states of the relevant global processes. On the basis of the notions of process, subprocess, trap, global process and consistency rule, Paradigm models for foreseen coordination situations succeed in guaranteeing dynamic consistency of type T1–T3. Please note, the complete Paradigm model discussed above has five more rules, governing a Worker's detailed transition each. As Workers are pure employees, no coupling to global process steps exists, so we have not listed the rules here. As a final remark, our modelling experience has shown, coordination of Paradigm's processes does not discriminate between modelling of machine activity, like software, and modelling of human activity, like business processes.

3 Solution Proposed: Unforeseen Coordination

In view of unforeseen change within an architecture, the special component McPal is to be added to it: for coordinating the unforeseen migration towards the new way of working. During a stable collaboration phase, McPal does not influence the other components at all. But by being there, McPal provides a pattern for dynamic evolution management in the distributed architectural constellation of the Paradigm model. To that aim, ports and links are in place, realizing rudimentary interfacing for the moment. As soon as a new way of working together with a migration towards it, has been modelled, typically just-in-time (JIT), McPal starts the migration coordination: its own migration begins, the migration of the others begins thereafter. Finishing the migration is done in reversed order. The others are explicitly left to their new stable collaborative work before McPal ceases to influence the others.

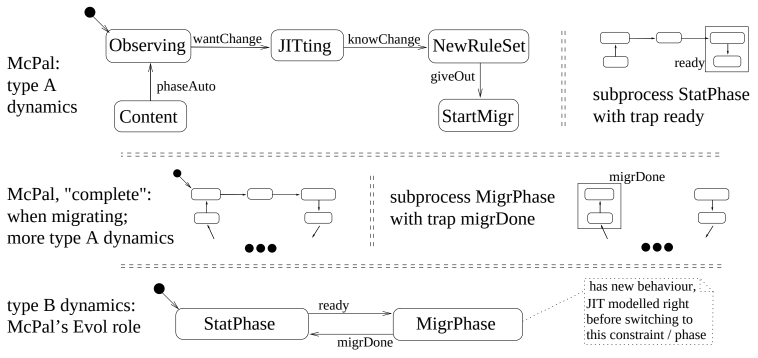

Fig. 6. McPal - during a stable collaboration situation 'mainly'.

Figure 6 visualizes McPal's hidden, detailed dynamics (type A) as follows. In its starting state *Observing*, McPal is doing nothing in particular. State *JITting* is where JIT foreseeing and JIT modelling occur. The extended model then is available in state $NewRuleSet$. So, upon leaving that state for state $StartMigr$, McPal is supposed to change its own subprocess $StatPhase$ into an originally unknown next subprocess $MigrPhase$, which by then is known indeed.

Figure 7 gives an overview of all components cooperating in collaboration EvolColl. McPal has the EvolCoord role, which here means, it is manager of five employees having an Evol port each: three Workers, Scheduler and itself, so it imposes the constraints on the employees according to the JIT modelled migration coordination details. Figure 8 visualizes a very small part of the type D

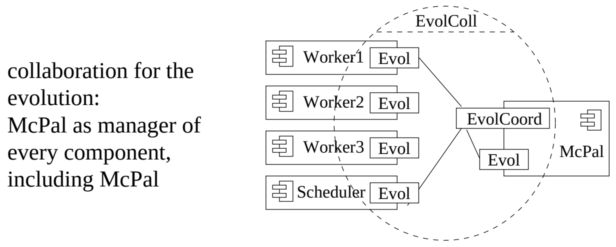
collaboration for the evolution:
McPal as manager of every component, including McPal
EvolColl
Worker1
Evol
Worker2
Evol
Worker3
Evol
Scheduler
Evol
EvolCoord
Evol
McPal

Fig. 7. The other components.

dynamics of collaboration EvolColl: the constraint orchestration fragment most essential for McPal's Evol role. Similar as before, it is built by synchronizing all five Evol roles of the respective components coupled in addition to the detailed steps of manager McPal. The Paradigm model for McPal has the following two consistency rules specifying the semantics for McPal's first two steps.

$$McPal\colon Observing \overset{wantChange}{\rightarrow} JITting$$
$$McPal\colon JITting \overset{knowChange}{\rightarrow} NewRuleSet \; * \; McPal\,[\,Crs := \; Crs + Crs_{migr} + Crs_{toBe}\,]$$

The first step from state $Observing$ to $JITting$ has no coupling, so Figure 8 does not show a corresponding role step. The second step from $JITting$ to $NewRuleSet$ has no coupling either, but here, via a so-called change clause, the set Crs of consistency rules for the stable original situation is extended with the rules Crs_{migr} for the migration only and with the rules Crs_{toBe} for the new, stable situation to migrate to. This means in particular, apart from all other migration coordination details, McPal from then on has at least two more rules:

$$McPal\colon NewRuleSet \overset{giveOut}{\rightarrow} StartMigr \; * \; McPal(\,Evol)\colon StatPhase \overset{ready}{\rightarrow} MigrPhase$$
$$McPal\colon Content \overset{phaseAuto}{\rightarrow} Observing \; *$$
$$McPal(\,Evol)\colon MigrPhase \overset{migrDone}{\rightarrow} StatPhase, \; McPal\,[\,Crs := \; Crs_{toBe}\,]$$

The first new rule says, on the basis of having entered trap ready, the subprocess change from $StatPhase$ to $MigrPhase$ can be made, coupled to McPal's transition from state $NewRuleSet$ to $StartMigr$. Figure 8 expresses this where the upper 'lightning' step draws a reader's attention. One clearly sees, the three Workers and Scheduler remain the same, as there is no constraint change for

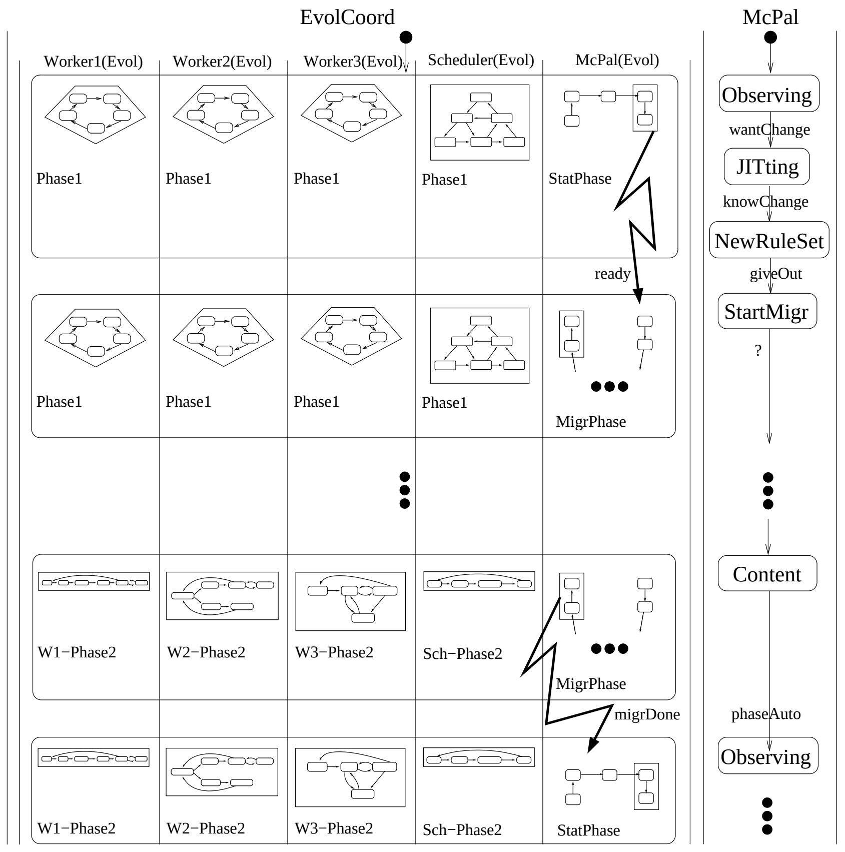

Fig. 8. Migration coordination as constraint orchestration.

them yet. From then on the migration is a matter of normal coordination only, exactly according the planning as modelled while in state *JITting*. Eventually, Scheduler is restricted to *Sch-Phase2* and each Worker$_i$ is restricted to W_i*-Phase2*. (The new subprocesses remain unexplained here; we mention them only.) Moreover, their new coordination has by then been adapted as planned. So, consistency is accounted for by normal Paradigm coordination execution. At the last migration step, after having phased out the dynamics being no longer needed for the other components, i.e. after having entered trap *migrDone* of its subprocess $MigrPhase$, McPal returns from $MigrPhase$ to *StatPhase* by making the (coupled) step from state *Content* to *Observing*. Then also the rule set *Crs* is reduced to Crs_{toBe}, by means of a suitable change clause. Once returned in state *Observing*, McPal is ready for a next migration. This can be seen in Figure 8 (lower 'lightning') and is expressed by the corresponding new rule for McPal.

4 Conclusion

In view of future, unforeseen migration of the model on-the-fly, McPal is to be incorporated into any Paradigm model. Thus McPal provides a pattern for dynamic evolution management in a distributed constellation of the architecture. McPal comprises the coordination needed for coordinating the migration collaboration of all components towards a new way of working. Modelled just-in-time, migration coordination is such that no form of temporary quiescence is needed to occur for whatever component during its migration. Any such component remains in execution, be it in a smoothly changing manner, gradually adapting to new circumstances arising along the migration trajectory. McPal's coordination, as specified in Paradigm, maintains consistency of the model both during and after migration, thereby dealing with the various dynamic consistency problem types. As Paradigm's processes can model both ICT activities and business activities, the McPal pattern is particularly promising for addressing all kinds of alignment situations between business and ICT and moreover for addressing general evolution of ICT and business in tandem.

McPal as introduced here, generalizes the McPal from [5], as it allows all kinds of complex behaviour in the JIT modelled subprocess $MigrPhase$, as long as it remains a correct Paradigm coordination. The McPal component proposed in [5] was restricted to a fixed migration coordination pattern. As a next and substantial generalization we are going to incorporate semantics for creating as well as deleting dynamics, in particular of type A and of type B, consistently. This will enable e.g. creation of a stand-by McPal when the original McPal is busy with a first migration: the stand-by can then initiate a different migration, even before the first is finished as JIT foreseen. By means of ParADE, a recently developed modelling and animating environment for Paradigm models, first migration examples have been implemented and tested.

Related work. Software evolution has numerous aspects (see [8, 1, 10]) for which various approaches and methods have been proposed (cf. [9, 15, 11]). The present work fits in the setting of unanticipated dynamic software evolution, focussing at the process level. For example, in a situation of dynamic co-evolution, where changing business rules and evolving software need to be aligned [12]. Architectural adaptation as graph transformations, as an implicit McPal, has been studied by various authors, [13] being reminiscent to our approach. Behavioral consistency and UML diagrams have been addressed in [14] too, where graph transformations techniques are exploited for a faithful reconstruction of activity diagrams from sequences diagrams. Consistency in the context of evolution and self-management is also addressed, e.g., in [3, 2, 16].

References

1. 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'04*, pages 28–33. ACM, 2004.
2. L. Desmet, N. Janssens, S. Michiels, F. Piessens, W. Joosen, and P. Verbaeten. Towards preserving correctness in self-managed software systems. In D. Garlan, J. Kramer, and A.L. Wolf, editors, *Proc. WOSS'04*, pages 34–38. ACM, 2004.
3. 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.
4. L. Groenewegen, N. van Kampenhout, and E. de Vink. Delegation modeling with paradigm. In J.-M. Jacquet and G.P. Picco, editors, *Proceedings Coordination 2005*, pages 94–108. LNCS 3454, 2005.
5. L. Groenewegen and E. de Vink. Evolution-on-the-fly with Paradigm. In P. Ciancarini and H. Wiklicky, editors, *Proc. Coordination 2006*, pages 97–112. LNCS 4038, 2006.
6. L.P.J. Groenewegen, A.W. Stam, P.J. Toussaint, and E.P. de Vink. Paradigm as organization-oriented coordination language. In L. van de Torre and G. Boella, editors, *Proc. CoOrg 2005*, pages 93–113. ENTCS 150(3), 2005.
7. J. Küster. *Consistency Management of Object-Oriented Behavioral Models*. PhD thesis, University of Paderborn, 2004.
8. M.M. Lehbman and J.F. Ramil. Software evolution: background, theory, practice. *Information Processing Letters*, 88:33–44, 2003.
9. T. Mens. ERCIM working group on software evolution. *ERCIM News*, 60:9, 2005.
10. T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, and M. Jazayeri. Challenges in software evolution. In *Proc. IWPSE'05, Lisbon*, pages 13–22. IEEE, 2005.
11. B. Meyer. Contract-driven development. In M.B. Dwyer and A. Lopes, editors, *Proc. FASE'07*, page 11. LNCS 4422, 2007. Invited talk.
12. R. Morrison, D. Balasubramaniam, F. Oquendo, B. Warboys, and R.M. Greenwood. An active architecture approach to dynamic systems co-evolution. In F. Oquendo, editor, *Proc. ECSA'07*, pages 2–10. LNCS 4758, 2007.
13. J. Padberg. Basic ideas for transformations of specification architectures. In R. Heckel, T. Mens, and M. Wermelinger, editors, *Proc. ICGT'02*, pages 46–58. ENTCS 72(4), 2003.
14. D.C. Petriu and Yimei Sun. Consistent behaviour representation in activity and sequence diagrams. In A. Evans, S. Kent, and B. Selic, editors, *Proc. UML 2000*, pages 369–382. LNCS 1939, 2000.
15. P. Wadler. Faith, evolution, and programming languages: from Haskell to Java to links. In *Proc. OOPSLA'06, Portland, Oregon*, page 508. ACM, 2006. Invited key-note.
16. J. Zhang and B.H.C. Cheng. Model-based development of dynamically adaptive software. In L.J. Osterweil, H.D. Rombach, and M.L. Soffa, editors, *Proc. ICSE'06*, pages 371–380. ACM, 2006.