Evolution On-the-Fly with Paradigm

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

[1] LIACS, Leiden University, The Netherlands
[2] Department of Mathematics and Computer Science
Technische Universiteit Eindhoven, The Netherlands
luuk@liacs.nl, evink@win.tue.nl

Abstract. The coordination language Paradigm allows for a flexible and orthogonal modeling of interprocess relationships at the architectural level. It is shown how dynamic system adaptation can be captured in Paradigm by means of a special evolution component and associated evolution coordination scheme. The component, called McPal, drives the migration following a just-in-time strategy in its own view of the system, independent of other coordination relations. During migration, dynamic consistency between components remains assured, even for mixtures of old, intermediate and new behaviour. A restricted scheme of McPal that supports various forms of self-adaptation is presented. A simple but generic example of a scheduler and workers illustrates on-the-fly updating of coordination and run-time adaptation of scheduling policies using McPal.

Keywords: evolution on-the-fly, dynamic consistency, self-adaptation, migration, software architecture, Paradigm.

1 Introduction

Dynamic aspects of coordination arise naturally when considering evolution on-the-fly of systems at the architectural level. Here, evolution on-the-fly, such as dynamic software updating, is in contrast to other unanticipated system adaptation where components are first put to a halt, subsequently supplied with new behaviour, and finally restarted, likely with their state restored. In the setting here, the execution of the system should continue as much as possible. While running, the system will adapt itself and evolve into a new one via a number of migration steps, taken by the different components in a well-coordinated fashion.

The system architectures addressed in this paper are given as Paradigm models. Paradigm is a coordination language distinguishing detailed and global behaviour of processes (see [8, 12]). Coordination is achieved, by properly connecting the detailed and global views via so-called consistency rules. These rules relate detailed transitions between states of a process in a manager role to global change of subprocess constraints of other processes in an employee role. In Paradigm, separate coordination solutions for multiple collaborations can be relatively easily combined into one single architecture.

P. Ciancarini and H. Wiklicky (Eds.): COORDINATION 2006, LNCS 4038, pp. 97–112, 2006.

The paper's main contribution lies in showing how a specific component, called McPal –abbreviating Managing changing Processes ad libitum (or at leisure)– allows for modifying the dynamics of the system on-the-fly, while all components remain in execution in a dynamically consistent manner. The important intuition here is, a process within a model is viewed as a subprocess of an unknown larger process. As long as this subprocess constraint is valid, it is irrelevant whether the remainder of that process is known or not. This allows for defining new fragments of the process in a lazy manner, by modeling them just in time. After having such new fragments defined in a suitable manner, McPal can, on the basis of newly added dynamics, start to coordinate global level behaviour, eventually leading into a new evolutionary phase for each component. Thereby, the Paradigm notions guarantee enduring consistency between the components' behaviours before, during and after the migration, even for mixtures of old, intermediate and new behaviours.

To keep our explanation clear and sufficiently brief, we restrict this paper to a relatively simple form of McPal, not changing its own behaviour. This way, an evolution pattern for Paradigm models emerges, illustrated by an example of a scheduler and workers involved in different evolution scenarios. The starting point is a scheduler that excludes almost all overlapping of activity of the workers. As a first illustration, the coordination evolves to a situation where this restriction is significantly alleviated, allowing for more parallelism amongst the workers. A second illustration of the evolution pattern focuses on the scheduler, that migrates from a non-deterministic selection of workers to a round-robin selection policy.

In view of the above, the remainder of the paper has the following structure. Section 2 briefly describes Paradigm and introduces a small coordination example model. First, the model is extended in Section 3 with McPal for coordinating future self-adaptation of the model. Sections 4 and 5 present two different evolutionary continuations, one for reducing the critical sections, the other for changing the scheduling policy of workers. Both evolutionary changes are on-the-fly. Finally, Section 6 gives conclusions and discusses both related work and future research.

2 Paradigm

This section provides a very brief introduction to the coordination language Paradigm by means of a small example. The example serves two purposes, of illustrating the notions explained and of preparing the evolution to be addressed later. For more detailed explanation, including formal operational semantics, see e.g. [8, 12, 11]. Consider the following coordination situation. A scheduler is coordinating the activities of three workers. The workers are performing the same life-cycle simultaneously: alternating between not working and working. A worker is idle in the state `free`. The activity working actually consists of four smaller consecutive activities: `nonCrit`, `pre`, `crit` and `post`. The scheduler is coordinating their working. For the moment, the scheduler allows at most one

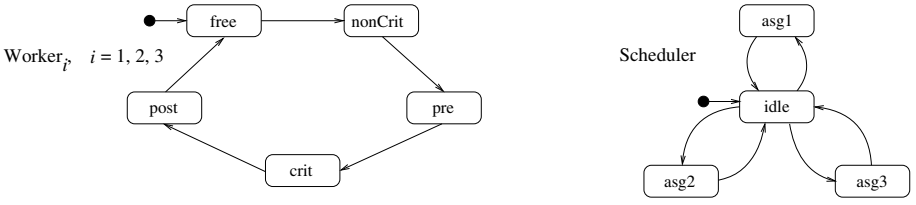

Fig. 1. Processes $\texttt{Worker}_i$ and `Scheduler`

worker outside of `free` and `nonCrit`. The activities of the workers and of the scheduler can be described by a process or state transition diagram (STD).

In general, a *process* or *STD* S is a triple $\langle \texttt{ST}, \texttt{AC}, \texttt{TS} \rangle$, with `ST` the set of *states*, with `AC` the set of *actions* or *labels*, and with $\texttt{TS} \subseteq \texttt{ST} \times \texttt{AC} \times \texttt{ST}$ the set of *transitions*. We write $x \xrightarrow{a} x'$ in case $(x, a, x') \in \texttt{TS}$, or even $x \rightarrow x'$ if the precise action is irrelevant.

Figure 1 visualizes the processes of a worker and of the scheduler as directed graphs: transitions as edges and states as nodes. Activities have been mapped to states (as time spent for an activity coincides with a sojourn in a state); actions have been left empty. Each worker process, starts in state `free`, its non-working activity. After state `free`, $\texttt{Worker}_i$ continues to work: in `nonCrit` he does non-critical work, in `pre` he prepares the critical activity, in `crit` he does his critical work and in `post` he does its follow-up. Thereby he finishes working and continues in `free` with non-working. Process `Scheduler` starts in state `idle` where he does not allow any worker to do non-critical working. From state `idle` he can go non-deterministically to any of his states $\texttt{asg}_i$, $i = 1, 2, 3$, where he only allows $\texttt{Worker}_i$ to enter and leave states `pre`, `crit` and `post` for performing one full turn of critical activity.

The coordination exerted by the scheduler is formulated in terms of three global processes, each constituting a coarse-grained view of a worker process. Global processes are built from subprocesses and traps of the process it corresponds to. In general, a *subprocess* of a process $S = \langle \texttt{ST}, \texttt{AC}, \texttt{TS} \rangle$ is a process $\langle \texttt{st}, \texttt{ac}, \texttt{ts} \rangle$ such that $\texttt{st} \subseteq \texttt{ST}$, $\texttt{ac} \subseteq \texttt{AC}$ and $\texttt{ts} \subseteq \{ (x, a, x') \in \texttt{TS} \mid x, x' \in \texttt{st}, a \in \texttt{ac} \}$. Furthermore, a *trap* θ of a subprocess $S = \langle \texttt{st}, \texttt{ac}, \texttt{ts} \rangle$ is a non-empty set of states $\theta \subseteq \texttt{st}$ such that $x \in \theta$ and $x \xrightarrow{a} x' \in \texttt{ts}$ imply that $x' \in \theta$. If $\theta = \texttt{st}$, the trap is called trivial. For the worker processes we model two subprocesses,

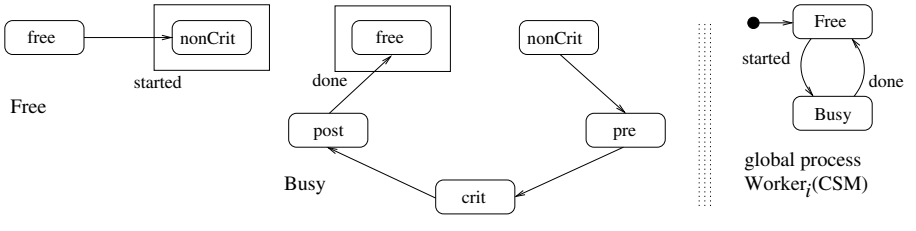

Fig. 2. Partition `CSM` and global worker process at the level of `CSM`

viz. `Free` and `Busy` with their traps `started` and `done` visualized as polygons surrounding the states belonging to it. See the left part of Figure 2.

Subprocesses and traps are both constraints: a subprocess of a process is a constraint on the possible behaviours of the process, meant to be temporary and meant to be imposed from outside the process by a *manager*. In the example, as we shall see, the process `Scheduler` will be the manager. A trap of a subprocess is a constraint on the subprocess' state space –once entered the trap cannot be left– valid only for the time the subprocess constraint holds, and committed to from inside the process in its *employee* role. So, a trap indicates a final stage of a subprocess.

To build suitable dynamics from such constraints, two more notions are needed: partition and connecting trap. In general, a *partition* π of a process $P = \langle \mathtt{ST}, \mathtt{AC}, \mathtt{TS} \rangle$ is a collection $\{ (S_i, T_i) \mid i \in I \}$ of subprocesses $S_i = \langle \mathtt{st}_i, \mathtt{ac}_i, \mathtt{ts}_i \rangle$ of P, each with a set T_i of its traps. Furthermore, for two subprocesses $S = \langle \mathtt{st}, \mathtt{ac}, \mathtt{ts} \rangle$ and $S' = \langle \mathtt{st}', \mathtt{ac}', \mathtt{ts}' \rangle$ of a partition π, a trap θ of S is called a *connecting* trap from S to S', if the states, of S, belonging to the trap θ are states in S' as well, i.e. $\theta \subseteq \mathtt{st}'$. If such a connecting trap θ from S to S' exists, the triple (S, θ, S') is called a *subprocess change* or *transfer*. In Figure 2, the left part presents the elements of partition `CSM` (abbreviating critical section management) of a worker process, comprising the two subprocesses and the two traps. In this case, trap `started` of `Free` is connecting from `Free` to `Busy` as its state `nonCrit` belongs to `Busy` too. Similarly, trap `done` of `Busy` is connecting from `Busy` to `Free` as its state `free` belongs to `Free` too. A connecting trap provides a kind of overlap between two consecutive subprocess constraints.

On the basis of a partition π of a process P, we construct a new process, referred to as the *global process at the level of partition* π. This process is denoted by $P(\pi)$. Its states are the subprocesses from π, its actions are connecting traps from π and its transitions are the subprocess changes corresponding to these connecting traps. The one subprocess expressing the constraint valid at a certain moment, is referred to as the current subprocess of a partition at that moment. By construction, the current subprocess of a partition corresponds to the current state of the global process at the level of the partition. A process not being global, is referred to as detailed. Figure 2's right part presents the global process $\mathtt{Worker}_i(\mathtt{CSM})$, with its starting state `Free` chosen such that, as subprocess, it contains starting state `free` of process $\mathtt{Worker}_i$. The process $\mathtt{Worker}_i(\mathtt{CSM})$ presents a behavioural view on the original $\mathtt{Worker}_i$ process; being less detailed than $\mathtt{Worker}_i$, process $\mathtt{Worker}_i(\mathtt{CSM})$ is more coarse-grained, which is referred to as global in our terminology. Note that, a process can have multiple partitions, for each of which a global process exists.

Given a number of processes with associated partitions, so-called consistency rules relate detailed transitions between states of a manager process to global transfer between subprocesses of employee processes. (In this context, 'consistency' refers to the notions of dynamic consistency, horizontal or vertical, as proposed by Küster [14].) For a consistency rule to be applicable, one has to keep track of the current state of detailed as well as global processes. Any applicable

consistency rule, no matter of what manager and partition, can be selected for application. In general, a consistency rule has the format

$$P\colon s \xrightarrow{a} s' \;\ast\; P_1(\pi_1)\colon S_1 \xrightarrow{\theta_1} S_1', \ldots, P_n(\pi_n)\colon S_n \xrightarrow{\theta_n} S_n' \;. \tag{1}$$

In consistency rule (1), process P is the manager and process P_1 to P_n are the employees. Paradigm is restricted to having exactly one manager in a consistency rule. In case there are no employees, i.e. $n = 0$, we simply write $P\colon s \xrightarrow{a} s'$. Note that the local transition of P in (1) does not refer to a partition. The requirement on P for the consistency rule to apply is that the transition $s \xrightarrow{a} s'$ is possible in every current subprocess of process P, with respect to the various partitions of P. The requirements on P_1 to P_n with respect to consistency rule (1) is that each process P_i in its partition π_i has S_i as the current subprocess and within this subprocess trap θ_i has been entered. After application of the consistency rule the current state of manager P becomes s', the current subprocesses of employees P_1 to P_n become S_1' to S_n', respectively. Because of the demand of traps θ_i being connecting for subprocesses S_i and S_i', it does not matter in which detailed state the employees reside precisely. For each connecting trap, the whole of it is admitted in the new subprocess as a possible state to continue from.

For the scheduler process of our example, we provide the consistency rules

$$\texttt{Scheduler}\colon \texttt{idle} \rightarrow \texttt{asg}_i \;\ast\; \texttt{Worker}_i(\texttt{CSM})\colon \texttt{Free} \overset{started}{\rightarrow} \texttt{Busy}$$
$$\texttt{Scheduler}\colon \texttt{asg}_i \rightarrow \texttt{idle} \;\ast\; \texttt{Worker}_i(\texttt{CSM})\colon \texttt{Busy} \overset{done}{\rightarrow} \texttt{Free}$$

The first consistency rule expresses that `Scheduler` may change its current state `idle` into $\texttt{asg}_i$, provided the current subprocess constraint on $\texttt{Worker}_i$ is `Free` and the trap `started` has been entered, i.e. $\texttt{Worker}_i$'s current state belongs to the trap `started`. If `Scheduler` changes its current state from `idle` to $\texttt{asg}_i$ indeed, then, according to the rule, global process $\texttt{Worker}_i(\texttt{CSM})$ changes its current state from `Free` to `Busy`, or, put otherwise, the subprocess constraint of $\texttt{Worker}_i$ in partition `CSM` becomes `Busy` instead of `Free`. This is an example of horizontal dynamic consistency [14], as the rule couples behaviours from different components. Analogously, the second rule says, `Scheduler` may return from state $\texttt{asg}_i$ to state `idle`, provided the current subprocess constraint on $\texttt{Worker}_i$ is `Busy` and $\texttt{Worker}_i$'s current state belongs to trap `done`. If `Scheduler` indeed changes its current state from $\texttt{asg}_i$ to `idle`, then also global process $\texttt{Worker}_i(\texttt{CSM})$ changes its current state from `Busy` to `Free`. We see how both consistency rules couple one local scheduler transition to a (simultaneous) global $\texttt{Worker}_i(\texttt{CSM})$ transition. As `Scheduler` does not have a partition, it neither has any process at such a level, so `Scheduler`'s state transitions are not restricted by any current subprocess constraint from such a global level.

The workers have no employees. Therefore, their consistency rules are simpler than for the scheduler, each rule containing one of $\texttt{Worker}_i$'s detailed transitions only.

$$\texttt{Worker}_i\colon \texttt{free} \rightarrow \texttt{nonCrit} \qquad \texttt{Worker}_i\colon \texttt{crit} \rightarrow \texttt{post}$$
$$\texttt{Worker}_i\colon \texttt{nonCrit} \rightarrow \texttt{pre} \qquad \texttt{Worker}_i\colon \texttt{post} \rightarrow \texttt{free}$$
$$\texttt{Worker}_i\colon \texttt{pre} \rightarrow \texttt{crit}$$

A $\texttt{Worker}_i$ transition is possible only if it belongs to the current subprocess constraint of detailed process $\texttt{Worker}_i$ or, equivalently, to the current state of global process $\texttt{Worker}_i(\texttt{CSM})$. This is an example of vertical behavioural consistency in the sense of [14] between a detailed process and the global processes at the levels of its partitions.

In addition to the consistency rule format of (1), one can also have a so-called *change clause,* a consistency rule used here solely for changing the total set of consistency rules, instead of a consistency rule prescribing subprocess changes. A change clause is formulated as

$$P{:}\, s \xrightarrow{a} s' \;*\; [\,\texttt{var := expr}\,] \tag{2}$$

concerning a variable `var`, typically holding the list of consistency rules. It specifies that after application of the rule, process P has moved to state s' and the value of `expr` in s has been assigned to variable `var`. This way consistency rules can be added or deleted dynamically.

The initial configuration of a Paradigm model does not only comprise the starting states of the various detailed processes, but also those of the various global processes. Therefore, for every detailed process a starting state has to be specified, together with a subprocess to start from, for each of its partitions. Our example illustrates this as follows. The combined starting states are, grouped per detailed process and with $i = 1, 2, 3$:

$$(\texttt{Worker}_i, \texttt{Worker}_i(\texttt{CSM})) : (\texttt{free}, \texttt{Free}),\ \texttt{Scheduler}:\texttt{idle}$$

Note, each detailed starting state belongs indeed to each current subprocess constraint for that particular detailed process. The consistency rules guarantee the following property to be invariant: each current state of a detailed process belongs for each partition of that process to the current subprocess.

In summary, the above is an example of a Paradigm model. In general, a *Paradigm model* is a collection of detailed processes and global processes at the level of their given partitions, together with a set of consistency rules and a combination of detailed and global starting states. Dynamics within such a model stem from subsequent application of consistency rules. Any change of constraint specified by the particular consistency rule applied, then yields a number of global process transitions, i.e. subprocess transfers.

3 Self-adaptation and McPal

In this section we shall present a different model for the original coordination problem from the previous section. Apart from solving the same coordination problem, the model can modify itself and evolve, while remaining in execution, into another model, unknown in the beginning. For this we add the special process `McPal`. This `McPal` starts with not influencing the model as-is, nothing special to begin with. But our `McPal`, by its careful design, has the property to adapt the Paradigm model it belongs to. Process `McPal` is visualized in Figure 3.

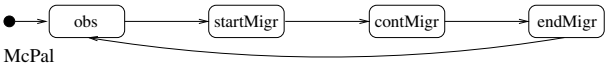

Fig. 3. Process `McPal`

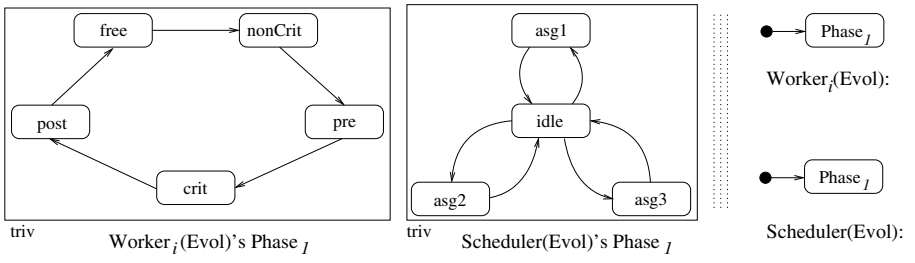

Fig. 4. Evolutionary phases and evolutionary processes to begin with

In state `obs`, `McPal` is observing the model as a whole and possibly computing or perhaps hearing from elsewhere how to change the model. As soon as `McPal` arrives in state `startMigr`, it knows the new consistency rules according to which the executing model is going to migrate uninterruptedly towards a new model. This migration is coordinated by `McPal` outside its state `obs`. To support the migration, every other detailed process has an additional partition called `Evol`. It contains the subprocesses reflecting the evolutionary phases so-far of that particular detailed process. As long as `McPal` is in state `obs` for the first time, each such partition consists of exactly one subprocess, here called $\texttt{Phase}_1$, always being the full, unconstrained process itself, typically with the trivial trap connecting from $\texttt{Phase}_1$ towards an as yet unknown subprocess reflecting an intermediate migration phase. Figure 4 presents the `Evol` partitions for $\texttt{Worker}_i$ as well as for `Scheduler` together with the degenerate global processes at the level of these partitions. All other partitions are unchanged. The consistency rules for the extended model are the original rules together with only one new rule for `McPal`, viz. consistency rule (3) below, for the moment.

$$\texttt{McPal:obs} \rightarrow \texttt{startMigr} \ * \ [\ \texttt{CR} \ := \ \texttt{CR} \cup \texttt{CR}_{mig} \cup \texttt{CR}_{next}\ ] \qquad (3)$$

This one rule is a change clause: as the set of consistency rules we start with, is going to change during the migration, the set of consistency rules is bound to a local variable of `McPal`, here denoted as `CR`. Change clause (3) should be taken parametrically, i.e., at the very moment process `McPal` decides to take the transition from its state `obs` to state `startMigr`, the collection of consistency rules gets updated via the assignment in the change clause with the whole of consistency rules in `CR`, $\texttt{CR}_{mig}$ and $\texttt{CR}_{next}$ at that moment. So the effect of the change clause (3) fully depends on the actual values of its two parameters $\texttt{CR}_{mig}$ and $\texttt{CR}_{next}$. As we shall see later, `McPal` determines the pace of evolution.

With respect to the new detailed process `McPal` it is important to note, only the first transition of `McPal`, from state `obs` to `startMigr` is supported by a

consistency rule. This means, in the first phase of the evolution the other transitions cannot occur. But, as a result of this, one possible transition in McPal, the set CR of consistency rules is extended with two more sets: one, $\mathtt{CR}_{mig}$, for the intermediate phase proper and the other, $\mathtt{CR}_{next}$, for the next evolutionary phase. In particular, once $\mathtt{CR}_{mig}$ is known, there are consistency rules for the other McPal transitions too, readily made, JIT-modeled (just in time), either by computational effort of McPal while in state obs, or by modeling effort from outside McPal and given as input to McPal while in state obs.

For this paper we shall keep process McPal unchanged. This means, we choose one particular form of sufficiently useful standard behaviour of McPal for actually coordinating the migration steps of other processes involved. By its first transition obs → startMigr, McPal extends via change clause (3) the rules, such that other processes' $\mathtt{Phase}_1$ constraint is going to be relaxed, if necessary. For the examples we want to discuss here, three more migration steps will do: a transition startMigr → contMigr for continuing the migration, only if necessary, by adjusting the migration already begun; a transition contMigr → endMigr for restraining the other processes' constraints to the $\mathtt{Phase}_2$ behaviours aimed at, thus closing their migration; the last one, transition endMigr → obs, for restraining the migrational freedom for McPal too, thus preventing McPal in some unknown future to repeat an old migration when a new one is in place. In the next two sections we present some concrete self-adapting Paradigm models and provide detail for McPal's remaining three transitions.

The combined starting states of the Paradigm model are grouped according to the five detailed processes:

$$(\mathtt{Worker}_i, \mathtt{Worker}_i(\mathtt{CSM}), \mathtt{Worker}_i(\mathtt{Evol})) : (\mathtt{free}, \mathtt{Free}, \mathtt{Phase}_1),$$
$$(\mathtt{Scheduler}, \mathtt{Scheduler}(\mathtt{Evol})) : (\mathtt{idle}, \mathtt{Phase}_1),$$
$$\mathtt{McPal} : \mathtt{obs}.$$

4 Reducing the Extent of Exclusive Behaviour

The example variants presented in this section exhibit a restricted form of self-adaptation through step-wise modification of non-evolutionary global behaviours only, i.e. exclusively at the level of the three $\mathtt{Worker}_i(\mathtt{CSM})$ partitions. For the moment, the detailed worker and scheduler processes are not going to change at all, so their $\mathtt{Phase}_1$ subprocess constraints remain unchanged during the evolution as discussed here. The non-evolutionary global processes $\mathtt{Worker}_i(\mathtt{CSM})$ do change, however.

To become more concrete, suppose we want to modify the original mutual exclusion management in two ways: by substantially restricting the exclusive behaviour as well as by making the return to non-critical working most asynchronous. These two improvements are realized by the new subprocesses OutCS and InCS, drawn in Figure 5, with their traps entering and left. Trap entering has been chosen as 'near' to state crit and as 'small' as possible. Trap left has been chosen as 'large' as possible –the larger the more asynchronous– and starting 'immediately after' state crit.

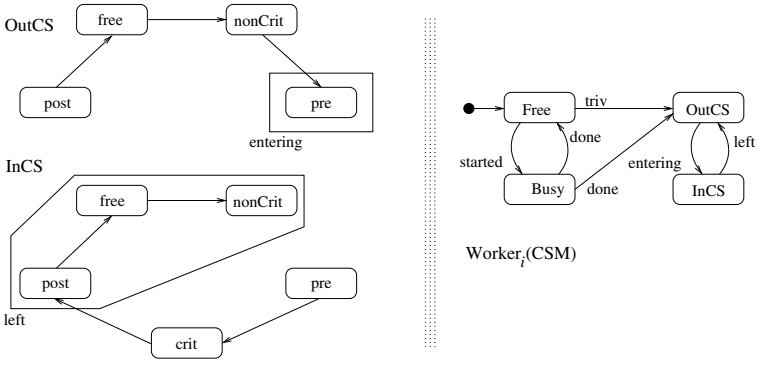

Fig. 5. Extensions of partition `CSM` and of global process $\texttt{Worker}_i(\texttt{CSM})$

The trivial trap `triv` of subprocess `Free` is self-evident, so we do not redraw `Free` with its newly added trap. Nevertheless, it now also belongs to partition `CSM`. What is even more important, trap `triv` is connecting from `Free` to `OutCS` and the original trap `done` is connecting from `Busy` to `OutCS`. This is used for constructing the global process $\texttt{Worker}_i(\texttt{CSM})$ on the basis of the newly extended partition `CSM`. See Figure 5. The consistency rules in play are the original consistency rules in `CR`, the consistency rules $\texttt{CR}_{mig}$ that guide the migration, and the consistency rules $\texttt{CR}_{next}$ for the next evolutionary phase. The original consistency rules are bound to `McPal`'s local variable `CR`, that is to say, `CR` has the rules mentioned in Section 3 as initial value. The consistency rules $\texttt{CR}_{mig}$ for the migration are the following.

$$\texttt{McPal}\colon \texttt{startMigr} \rightarrow \texttt{contMigr}$$

$$\texttt{McPal}\colon \texttt{contMigr} \rightarrow \texttt{endMigr} \ * \ \texttt{Worker}_1(\texttt{CSM})\colon \texttt{OutCS} \overset{triv}{\rightarrow} \texttt{OutCS}$$

$$\texttt{McPal}\colon \texttt{endMigr} \rightarrow \texttt{obs} \ * \ [\,\texttt{CR} := \texttt{CR}_{next}\,]$$

$$\texttt{Scheduler}\colon \texttt{asg}_i \rightarrow \texttt{idle} \ * \ \texttt{Worker}_i(\texttt{CSM})\colon \texttt{Busy} \overset{done}{\rightarrow} \texttt{OutCS},$$
$$\texttt{Worker}_{i-1}(\texttt{CSM})\colon \texttt{Free} \overset{triv}{\rightarrow} \texttt{OutCS},\ \texttt{Worker}_{i+1}(\texttt{CSM})\texttt{Free} \overset{triv}{\rightarrow} \texttt{OutCS}$$

Here $i-1$ and $i+1$ denote the usual predecessor and successor values of i in the cyclic order of $1, 2, 3$.

The first rule for `McPal` above states that, in this migration, there is no coordination task for `McPal` in the first step. The second rule for `McPal` expresses that the migration has been completed, when the first worker runs restricted to the `OutCS` subprocess at the level of its `CSM` partition. Once in state `endMigr`, `McPal` cleans up the old and intermediate consistency rules by binding `CR` to $\texttt{CR}_{next}$ and comes back in state `obs` again. Here the actual migration is in the new rule for `Scheduler`. When returning from any of the states $\texttt{asg}_1$, $\texttt{asg}_2$, $\texttt{asg}_3$, all workers, including the $\texttt{Worker}_1$ that is checked by `McPal`, are transfered to subprocess `OutCS` of the next evolution phase. Note, the actual migration is not really enforced: `Scheduler` might carry on with the old coordination forever. As

soon as the new consistency rule has been applied, however, migration has taken place irreversibly.

The consistency rules $\mathtt{CR}_{next}$ for the next evolutionary phase, the new solution we are actually aiming at, are

$$\begin{array}{ll}
\mathtt{Worker}_i\mathtt{:free} \rightarrow \mathtt{nonCrit} & \mathtt{Worker}_i\mathtt{:crit} \rightarrow \mathtt{post} \\
\mathtt{Worker}_i\mathtt{:nonCrit} \rightarrow \mathtt{pre} & \mathtt{Worker}_i\mathtt{:post} \rightarrow \mathtt{free} \\
\mathtt{Worker}_i\mathtt{:pre} \rightarrow \mathtt{crit} &
\end{array}$$

$$\mathtt{Scheduler:idle} \rightarrow \mathtt{asg}_i \ * \ \mathtt{Worker}_i(\mathtt{CSM})\mathtt{:OutCS} \stackrel{entering}{\rightarrow} \mathtt{InCS}$$

$$\mathtt{Scheduler:asg}_i \rightarrow \mathtt{idle} \ * \ \mathtt{Worker}_i(\mathtt{CSM})\mathtt{:InCS} \stackrel{left}{\rightarrow} \mathtt{OutCS}$$

$$\mathtt{McPal:obs} \rightarrow \mathtt{startMigr} \ * \ [\ \mathtt{CR} := \mathtt{CR} \cup \mathtt{CR}_{mig} \cup \mathtt{CR}_{next}\ ]$$

In the new evolution phase, the same scheduler continues to coordinate the same workers with more parallelism between the workers, now the extent of the exclusive, critical working interval has been reduced and trap `left` allows for a most asynchronous continuation of a worker after having finished his critical activity. The change clause for `McPal` for the transition from `obs` to `startMigr` caters for later evolution, at `McPal`'s leisure.

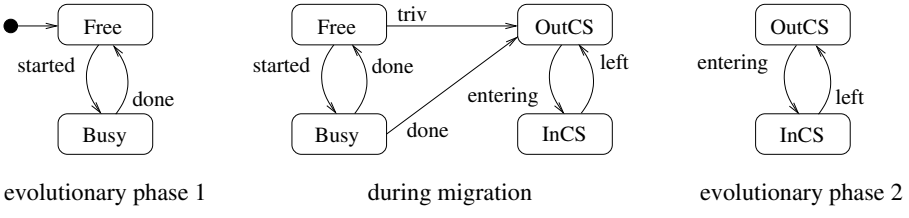

Fig. 6. Migration of process $\mathtt{Worker}_i(\mathtt{CSM})$, first variant

Figure 6 might be viewed as a movie of global process $\mathtt{Worker}_i(\mathtt{CSM})$ represented through three subsequent takes reflecting subsequent migration. The first take of the movie lasts until `McPal` leaves state `obs`. The second take lasts while `McPal` is in its three states `startMigr`, `contMigr` and `endMigr`. Upon arrival in state `endMigr` it is guaranteed that the worker process executes either subprocess `OutCS` or `InCS`. The third take starts when `McPal` returns in state `obs`.

An interesting variant, slightly different only, is `McPal`'s first migration rule in the above set $\mathtt{CR}_{mig}$ replaced by

$$\mathtt{McPal:startMigr} \rightarrow \mathtt{contMigr} \ * \ [\ \mathtt{CR} := \mathtt{CR} \backslash \mathtt{CR}_{help}\ ]$$

where $\mathtt{CR}_{help}$ has been computed or has been read from input in state `obs`, consisting of the 'migration avoiding' rules

$$\mathtt{Scheduler:asg}_i \rightarrow \mathtt{idle} \ * \ \mathtt{Worker}_i(\mathtt{CSM})\mathtt{:Busy} \stackrel{done}{\rightarrow} \mathtt{Free}$$

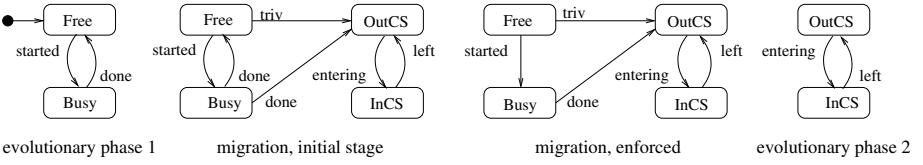

Fig. 7. Migration of process $\texttt{Worker}_i(\texttt{CSM})$, second variant

The consequence thereof is, from `McPal`'s arrival in state `contMigr`, the migration is more enforcing towards $\texttt{Phase}_2$, the actual evolutionary phase aimed at. The movie now consists of four takes, see Figure 7. The enforcing is an example of migration adjusting. Note, we have indeed specified the self-adaptation through new global behaviour at the level of $\texttt{Worker}_i(\texttt{CSM})$ only, leaving all detailed behaviours untouched.

5 Changing Scheduling Order

This section presents example variants of self-adaptation affecting a detailed process and involving non-trivial global behaviour at the level of partition `Evol` of that detailed process, really evolutionary behaviour in terms of subsequent phases.

To this aim, we reorganize the Paradigm model of Section 3 in a different way, namely by changing the non-deterministic selection policy of the scheduler into round robin selection. Figure 8 visualizes a combination of the original scheduler process and a new, envisaged one. The four upper states, of the seven states displayed, constitute the STD of the `Scheduler` process as in Figure 1. The six lower states, $\texttt{asg}_1$ to $\texttt{asg}_3$ and $\texttt{check}_1$ to $\texttt{check}_3$, will comprise the state space of the evolved `Scheduler` process. So, here we have a different type of change: a detailed process is going to get new behaviour. Round robin checking of a worker's wish to enter its critical section, is done in the states $\texttt{check}_i$, whereas the meaning of states $\texttt{asg}_i$ is kept unchanged. The combined behaviours as presented in Figure 8 actually obscure what transitions could be taken during

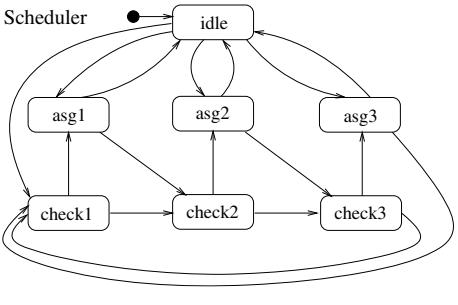

Fig. 8. Combined STD of two incarnations of `Scheduler`

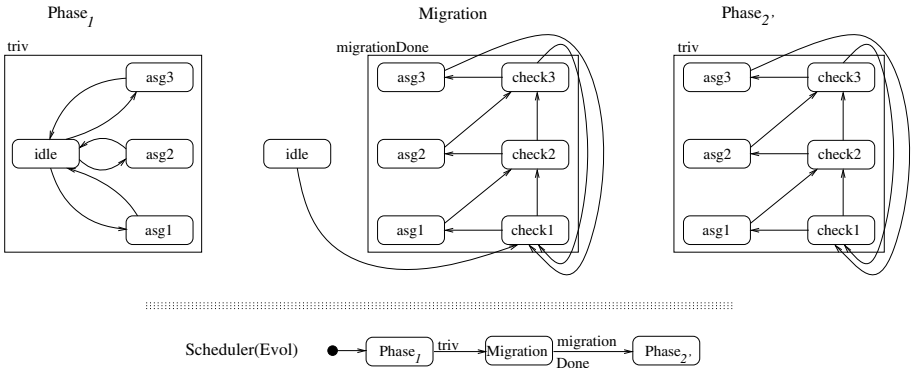

Fig. 9. Partition `Evol` of `Scheduler`, first variant

which evolutionary phase. At level of the partition `Evol` this becomes rather more clear from global process `Scheduler(Evol)`. Figure 9 presents the partition `Evol` of the extended process `Scheduler` under migration, together with global process `Scheduler(Evol)`. Without the traps and with `idle` added as starting state of the first evolutionary phase, it can be taken as a movie of process `Scheduler`'s evolution represented through three subsequent takes. Consistency rules for the migration as well as for the new way of scheduling are grouped into the two sets $\mathtt{CR}_{mig}$ and $\mathtt{CR}_{next}$, like before. Remember, we start with `CR` as modeled in Section 3. As before, $i+1$ denotes the successor of i from the cyclic values $1, 2, 3$.

In addition, we use the negative side rule $P(\pi)\colon S \overset{\theta}{\nrightarrow}$ for the condition that within partition π of process P either S is not the current subprocess, or trap θ of subprocess S has not yet been entered. This condition is necessary for the corresponding manager transition to occur and it leaves the current subprocess at the level of partition π of P unchanged. The consistency rules for the migration are now the following.

$$\begin{aligned}
&\mathtt{McPal}\colon \mathtt{startMigr} \rightarrow \mathtt{contMigr} \ * \\
&\qquad \mathtt{Scheduler(Evol)}\colon \mathtt{Phase}_1 \overset{triv}{\rightarrow} \mathtt{Migration} \\
&\mathtt{McPal}\colon \mathtt{contMigr} \rightarrow \mathtt{endMigr} \ * \\
&\qquad \mathtt{Scheduler(Evol)}\colon \mathtt{Migration} \overset{migrationDone}{\rightarrow} \mathtt{Phase}_2 \\
&\mathtt{McPal}\colon \mathtt{endMigr} \rightarrow \mathtt{obs} \ * \ [\, \mathtt{CR} := \mathtt{CR}_{next} \,] \\
&\mathtt{Scheduler}\colon \mathtt{idle} \rightarrow \mathtt{check}_1
\end{aligned}$$

Please note, from the above list, the first rule for `McPal` starts the evolutionary migration phase for `Scheduler`. The one rule of `Scheduler` expresses the actual migration step, by going from `idle` to $\mathtt{check}_1$. The second rule of `McPal` stabilizes the migration: `Scheduler` is in evolutionary phase $\mathtt{Phase}_2$ from the moment the rule is applied. `McPal`'s last rule then discards rules no longer needed, by keeping those from $\mathtt{CR}_{next}$ only. It may happen, that the actual migration step to be taken by `Scheduler` does not occur, as `Scheduler`, while in $\mathtt{asg}_i$, gets its

phase `Migration` as the current subprocess constraint. In that case, `Scheduler` migrates right then implicitly, without taking an explicit step. As an aside, we have a non-trivial management relation: `McPal` manages the scheduler, while the scheduler manages the workers.

The consistency rules $\mathtt{CR}_{next}$ for the envisioned evolutionary phase are the following

$$\mathtt{Worker}_i\colon \mathtt{free} \to \mathtt{nonCrit} \qquad \mathtt{Worker}_i\colon \mathtt{crit} \to \mathtt{post}$$
$$\mathtt{Worker}_i\colon \mathtt{nonCrit} \to \mathtt{pre} \qquad \mathtt{Worker}_i\colon \mathtt{post} \to \mathtt{free}$$
$$\mathtt{Worker}_i\colon \mathtt{pre} \to \mathtt{crit}$$

$$\mathtt{Scheduler}\colon \mathtt{check}_i \to \mathtt{asg}_i \ * \ \mathtt{Worker}_i(\mathtt{CSM})\colon \mathtt{Free} \overset{started}{\to} \mathtt{Busy}$$
$$\mathtt{Scheduler}\colon \mathtt{asg}_i \to \mathtt{check}_{i+1} \ * \ \mathtt{Worker}_i(\mathtt{CSM})\colon \mathtt{Busy} \overset{done}{\to} \mathtt{Free}$$
$$\mathtt{Scheduler}\colon \mathtt{check}_i \to \mathtt{check}_{i+1} \ * \ \mathtt{Worker}_i(\mathtt{CSM})\colon \mathtt{Free} \overset{started}{\nrightarrow}$$
$$\mathtt{McPal}\colon \mathtt{obs} \to \mathtt{startMigr} \ * \ [\ \mathtt{CR} := \mathtt{CR} \cup \mathtt{CR}_{mig} \cup \mathtt{CR}_{next}\ ]$$

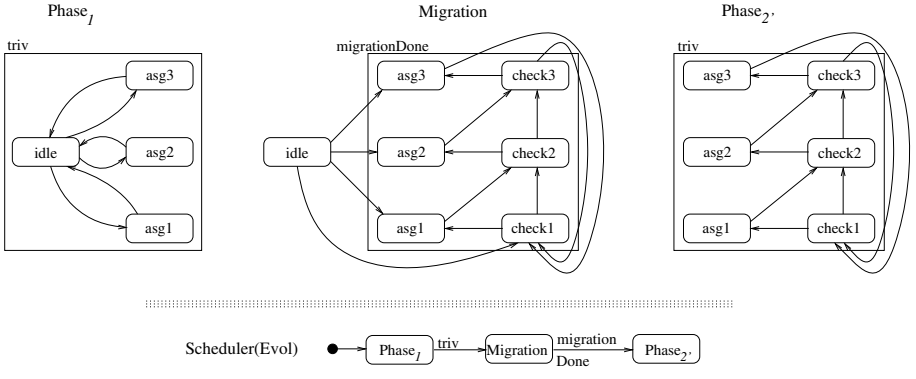

Fig. 10. Partition `Evol` of `Scheduler`, second variant

Figure 10 gives an alternative for partition `Evol` of the `Scheduler` process: during migration it allows for some extra delay of introducing the round robin approach, by a kind of last-orders possibility for at most one worker according to the old-fashioned non-deterministic selection mechanism. (See the transition from state `idle` to the states $\mathtt{asg}_i$ in subprocess `Migration`.) Note that by thus changing the definition of subprocess `Migration`, i.e. the concrete constraint, we do not have to change our consistency rule formulations for achieving the evolution as depicted in Figure 10.

6 Conclusions, Related Work and Future Work

The scheduler and worker examples illustrate evolution on-the-fly as supported in Paradigm. The detailed transitions of the special component `McPal` mark the

various migration steps from one evolutionary stage of the system to the next. Via consistency rules, new subprocesses of old or new behaviour can be introduced. Change clauses for `McPal` provide these consistency rules and dispose of coordination that is not desired any more. Although, the mechanism of subprocesses constrain the behaviour of components, the proposed evolution scheme itself does not require any component to stop, to be restarted later with new behaviour installed. In this sense, evolution is on-the-fly.

The evolution pattern as described allows for iterated evolution, as the `McPal` process is persistent in the model. A new migration can take place as soon as new consistency rules for intermediate and targeted behaviour and interaction are defined. With new semantics specified lazily, just-in-time, the self-adaptation is on-the-fly. In its present form, only one intermediate stage is foreseen in the evolution pattern, represented by the state `contMigr` of the detailed process of `McPal`. In the first evolution example variant, we have seen that this state was superfluous. The opposite, having more than one intermediate migration stage, is possible as well. Even stronger, the detailed behaviour of `McPal` can be specified just-in-time, determining the migration trajectory on-the-fly too. As such, our scheme provides unconstrained run-time selection of a migration trajectory. In general, any finite DAG with a unique starting state will do, as far as the structure of `McPal` is concerned.

Related work The contracts of Colman and Han [6] for the coordination of loosely coupled systems connect the organizational and functional views on an architecture, a distinction reminiscent to our managers and employees. The connection of Colman and Han, though, is by role instantiation, mapping the abstract to the concrete. Fundamental for Paradigm is the coupling of detailed state transitions with global subprocess transfer.

In the context of Component Based Software Engineering, adaptation is to be understood as component adaptation for interoperability purposes. Formal descriptional approaches are gaining impact in this field [2, 19, 16, 4], moving from IDLs based on finite state machines towards mobility-oriented process calculi and induced bisimulation equivalences. Although Paradigm is supported by a transition-based operational semantics [12], at present the behavioural theory to compare different evolutions or different stages within the same evolution, is not yet fully established. However, see [3, 13].

Oriol proposes to exploit asynchronous channels to drive unanticipated software evolution [17]. Leading principles are anonymity of entities, late binding and asynchronous communication in a setting of service-directed architectures. A variation geared towards tuple spaces has been reported in [18]. Because of the atomicity of the services involved, the granularity is more coarse-grained than in the approach presented here, however.

There is a vast amount of literature on dynamic updating at the code level, for example on concrete dynamic software updating systems design (see e.g. [15, 20]). In the context of declarative programming, dynamic logic programming (see [1] amongst others) involves sequences of logic programs to express the evolution of knowledge over time. Controls have to be put in place to deal with inconsistencies

among separate programs and to fine-tune asserts and retracts of Horn clauses. However, as with the work on imperative programming languages mentioned above, as yet no migration pattern or architectural support is provided to guide the evolution.

In addition, we like to mention [9, 10] as examples reporting on self-adaptive systems on an architectural level. In these papers there is the same tendency as noted above (service atomicity) of concentrating on forms of self-adaptation referred to as: reconfiguration, structured reorganization, data-driven readjustment, canned workflows being triggered, recomposition and the like. The survey [5] compares fourteen different approaches to self-adaptation, none of which appears to achieve so-called unconstrained run-time selection. Our McPal pattern however, allows for exactly this: the outcome of our JIT-modeling, occurring while being in state `obs`, fully determines such run-time freedom: nothing happening during migration or during later evolutionary phases has been foreseen from the beginning. The paper [7] is coming closest to our approach. It draws attention to the dynamic consistency problem, which should be solved by coordination; details about how to do this, are not given however, contrarily to our McPal component which, based on Paradigm, specifies such coordination in detail.

Future work Further research will be devoted to more elaborate migration schemes. A case study of the dining philosophers evolving from deadlock to starvation and beyond is under way. For the treatment of more intricate evolution patterns, dynamic creation and deletion of complete detailed processes is involved, which requires extension of Paradigm's formal semantics. Larger software architectures, for example in a setting of changes at the business level requiring software adaptation of lower-level support-systems, are likely to become multi-tier. Concise models will benefit from higher-order consistency rules and coordination patterns. Thesis work is devoted to tooling that supports the analysis of extensive examples as well as refactoring, transformation and refinement strategies of Paradigm models.

Acknowledgment. We are indebted to our colleagues of the FaST-group, including Fahrad Arbab, Frank de Boer, Marcello Bonsangue, Jetty Kleijn, and Andries Stam for various stimulating discussions on the subject.

References

1. J.J. Alferes, J.A. Leite, L.M. Pereira, H. Przymusinska, and T. Przymusinski. Dynamic updates of non-monotonic knowledge bases. *Journal of Logic Programming*, 45:43–70, 2000.
2. R. Allen and G. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering Methodology*, 6:213–249, 1997.
3. J.C. Augosto and R.S. Gomez. A temporal logic view of Paradigm models. In *Proc. SEKE 2002, Ischia, Italy*, pages 497–503. ACM, 2002.
4. A. Bracciali, A. Brogi, and C. Canal. A formal approach to component adaptation. *Journal of Systems and Software*, 74:45–54, 2005.

5. J. Bradbury, J. Cordy, J. Dingel, and M. Wermelinger. A survey of self-management in dynamic software architecture specifications. In *[10]*, pages 28–33. ACM Press, 2004.
6. A. Colman and Jun Han. Coordination systems in role-based adaptive software. In J.-M. Jacquet and G.P. Picco, editors, *Proc. Coordination 2005*, volume 3454 of *LNCS*, pages 63–78, 2005.
7. L. Desmet, N. Janssens, S. Michiels, F. Piessens, W. Joonsen, and P. Verbaeten. Towards preserving coorectness in self-managed software systems. In *[10]*, pages 34–38. ACM Press, 2004.
8. G. Engels, L.P.J. Groenewegen, and G. Kappel. Coordinated Collaboration of Objects. In M. Papazoglou, S. Spaccapietra, and Z. Tari, editors, *Advances in Object-Oriented Data Modeling*, pages 307–331. MIT Press, 2000.
9. D. Garlan, J. Kramer, and A. Wolf, editors. *Proceedings of the 1st Workshop on Self-Healing Systems, Charleston SC*. ACM, 2002.
10. D. Garlan, J. Kramer, and A. Wolf, editors. *Proceedings of the 1st ACM SIGSOFT Workshop on Self-Managing Systems, Newport Beach CA*. ACM, 2004.
11. L. Groenewegen, N. van Kampenhout, and E. de Vink. Delegation Modeling with Paradigm. In J.-M. Jacquet and G.P. Picco, editors, *Proc. Coordination 2005*, volume 3454 of *LNCS*, pages 94–108, 2005.
12. L. Groenewegen and E. de Vink. Operational semantics for coordination in Paradigm. In F. Arbab and C. Talcott, editors, *Proc. Coordination 2002*, volume 2315 of *LNCS*, pages 191–206, 2002.
13. N. van Kampenhout. Systematic Specification and Verification of Coordination: towards Patterns for Paradigm Models. Master's thesis, LIACS, 2003.
14. J. Küster. *Consistency Management of Object-Oriented Behavioral Models*. PhD thesis, University of Paderborn, 2004.
15. S. Malabarba, R. Pandey, J. Gragg, E. Barr, and J.F. Barnes. Runtime support for type-safe dynamic Java classes. In E. Bertino, editor, *Proc. ECOOP 2000*, volume 1850 of *LNCS*, pages 337–361, 2000.
16. Sun Meng and L.S. Barbosa. On refinement of generic state-based software components. In C. Rattray, S. Maharaj, and C. Shankland, editors, *Proc. AMAST'04*, volume 3116 of *LNCS*, pages 506–520, 2004.
17. M. Oriol. *An Approach to the Dynamic Evolution of Software Systems*. PhD thesis, Department of Information Systems, University of Geneva, 2004.
18. M. Oriol and M.W. Hicks. Tagged sets: A secure and transparent coordination medium. In J.-M. Jacquet and G.P. Picco, editors, *Proc. Coordination 2005*, volume 3454 of *LNCS*, pages 252–267, 2005.
19. P. Poizat. *Korrigan: un formalisme et une méthode pour la spécification formelle et structurée de systèmes mixtes*. PhD thesis, IRIN, University of Nantes, 2000.
20. G. Stoyle, M.W. Hicks, G. M. Bierman, P. Sewell, and I. Neamtiu. Mutatis mutandis: safe and predictable dynamic software updating. In *Proc. POPL 2005, Long Beach, Calefornia*, pages 183–194. ACM, 2005.