Delegation Modeling with Paradigm

Luuk Groenewegen[1] Niels van Kampenhout[1] Erik de Vink[1,2]

[1] LIACS, Leiden University
[2] Dept of Math. and Comp. Sc., Technische Universiteit Eindhoven
luuk@liacs.nl, nielsvankampenhout@wanadoo.nl, evink@win.tue.nl

Abstract. Within one model, behavioural consistency of its constituents is often problematic. Within UML such horizontal behavioural consistency between the objects of a concrete model, is particularly needed in the context of dynamic patterns. Here, we investigate delegation, which is fundamental to patterns that separate the locality of receiving a request, and one or more localities actually handling it. We specify delegation by means of the coordination language Paradigm. In particular, we present some variants of delegation in the context of a broker pattern and clarify how the Paradigm notions are the basis for understanding a solution as well as for adapting it to deal with other dynamic features.

1 Introduction

Software architectures are the major instrument to handle the size and complexity of today's software systems. Moreover, within the context of a business architecture, they pinpoint the software system's embedding in the non-digital world. Typically, an architecture consists of a number of components related via specific links. Components express certain aspects that contribute to the functionality of the system or the organization as a whole. Interaction among components is directed via their interfaces. To stress this even more, components are usually considered stateless. In the architectural description one abstracts away from the internal dynamics of a component in order not to clutter up the overall view. See, e.g., [19, 8]. Nevertheless, some dynamics survive in architectural descriptions, e.g. via protocols and protocol roles and other global dynamics, as these are relevant for dynamic consistency between components.

The problem of dynamic consistency between components constituting an architecture is, as yet, far from being solved. Even within the UML [3, 9], where the underlying, detailed dynamics of objects constituting a model contribute additional information to base dynamic consistency on, the problem of dynamic consistency is comparably far from being solved. Clarification of this problem situation is the more pressing, as increasingly often patterns are being used (both as means of design [10] and for business processes [5]) for consistently organizing and reorganizing the dynamics of the model's constituents.

Often, when modeling a software system with the UML, the use-cases act as 'glue logic' for the information carried by the respective description methods. Similarly, sequence diagrams restricting the dynamics to interactive steps only,

concentrate on 'gluing'. This is not only a matter of style: the concrete behaviour, as captured by the use-case or the sequence diagrams, has its consequences for the interaction of the components involved. More frequently rule than exception, the relevant information goes beyond the interface. Some of the internal dynamics of the component must be revealed in order to assess the correctness of the cooperation of a component and its software or non-software environment.

In order to judge the global behaviour of the system, the local behaviour of components is pivotal. For example, the interaction of a component should not be in conflict with its internal dynamics. This is consistency between different levels of description, more or less similar to Küster's vertical consistency (cf. [7, 16]). Also, the component should act in compatibility with the components from its surroundings. This is consistency between different model constituents, on the same level of description, suitably chosen to reflect the relevant collaboration; here, Küster's notion of horizontal consistency is more appropriate. Typically, such questions of consistency arise when components play multiple roles in multiple protocols that overlap in time. See, e.g., [17, 18].

The modeling technique that we propose for aligning global and local behaviour is Paradigm [6, 20, 12]. In Paradigm the coordination among a manager and its employees is the prime concern. It does so by relating the local behaviour of the manager to the global behaviour of the employees, the latter being decorated with just that little information that is necessary to maintain consistency of the system. In this way, via a manager, Paradigm addresses horizontal consistency between the manager's employees. Furthermore, via its special notions of subprocess and trap, Paradigm guarantees vertical consistency within an employee between its detailed and global behaviour. In the present paper we report on an application of Paradigm in business process modeling for a non-hierarchical organization. The example deals with delegation. Delegation, i.e. separating the components for starting behaviour and the component(s) continuing it, is behind many patterns [10, 5], thus requiring horizontal consistency between them.

Below, Section 2 introduces, informally, the key ingredients of Paradigm. A first description of the delegation example is covered in Section 3. An alternative model is presented in Section 4. Some other variants are discussed in Section 5. Finally, Section 6 wraps up with some concluding remarks.

2 Paradigm

Paradigm is a coordination specification language, concentrating on expressing behaviour and behaviour influencing. In this section we present Paradigm briefly and informally. Operational semantics of Paradigm have been presented in [12] and [11]. It is stressed that in the present paper Paradigm models do not require that coordination is organized in a strictly hierarchical manner.

Paradigm uses the notion of a process, together with a state-transition-diagram-like visualization for it. Usually, a process expresses a constituent's behaviour on the detailed level, corresponding to its inner, hidden behaviour. See,

e.g., Figure 3.1 for an example visualization as a directed graph: nodes are states and directed edges are transitions between two states.

For the behavioural description of a constituent on a more global level, Paradigm uses two additional notions: subprocess and trap. Whereas a process specifies all possible behaviours of a constituent —in Figure 3.1 called $\texttt{Client}(i)$— a subprocess (of a process) expresses a phase of that behaviour: a (temporary) restriction of that behaviour, relevant in the context of some collaboration between constituents. A trap of a subprocess, being a subset of the subprocess states, reflects a final, irrevocable stage of the subprocess: within a subprocess, a trap of it cannot be left once entered. So, a trap can serve as a kind of commit or acknowledge within the collaboration, e.g. declaring the subprocess behaviour has proceeded far enough to be changed from the (current) behaviour restriction into a suitable next one.

A partition then divides the full process behaviour into a set of subprocesses with their traps. Figure 3.4 gives a visualization of a partition of $\texttt{Client}(i)$ into 3 subprocesses. The relevant traps are drawn as polygons surrounding the states a trap consists of. We formalize these notions in the next definition.

Definition 2.1

(a) A process or STD S is a pair $\langle \texttt{ST}, \texttt{TS} \rangle$. Here $\texttt{ST}$ is called the set of states, or also the state space; $\texttt{TS} \subseteq \texttt{ST} \times \texttt{ST}$ is the set of transitions. We write $x \rightarrow x'$ in case $(x, x') \in \texttt{TS}$.

(b) A subprocess of S is a process $\langle \texttt{st}, \texttt{ts} \rangle$ such that $\texttt{st} \subseteq \texttt{ST}$ and $\texttt{ts} \subseteq \{ (x, x') \in \texttt{TS} \mid x, x' \in \texttt{st} \}$. A trap t of a subprocess $s = \langle \texttt{st}, \texttt{ts} \rangle$ is a nonempty set of states $t \subseteq \texttt{st}$ such that $x \in t$ and $x \rightarrow x' \in \texttt{ts}$ imply that $x' \in t$. If $t = \texttt{st}$, the trap is called trivial.

(c) Let $s = \langle \texttt{st}, \texttt{ts} \rangle$ and $s' = \langle \texttt{st}', \texttt{ts}' \rangle$ be two subprocesses of the same process. A trap t of s is called a connecting trap from s to s' if the states belonging to the trap t are states in s' as well, i.e., $t \subseteq \texttt{st}'$.

(d) A partition $\{ (s_i, t_i) \mid i \in I \}$ of a process $S = \langle \texttt{ST}, \texttt{TS} \rangle$ is a set of subprocesses $s_i = \langle \texttt{st}_i, \texttt{ts}_i \rangle$ with traps t_i such that $\texttt{ST} = \bigcup_{i \in I} \texttt{st}_i$ and $\texttt{TS} = \bigcup_{i \in I} \texttt{ts}_i$.

Although not explicitly defined, a global behaviour for a constituent, see, e.g., Figure 3.5, can be formulated in terms of a sequence of subprocesses glued together by means of a connecting trap. All phases occurring in such a sequence come from the same partition; we therefore say about such a global behaviour, it occurs on the level of that partition. Note that for a connecting trap all states in it belong to both subprocesses involved. For this paper we restrict ourselves to a single trap of any subprocess connecting it to a next subprocess.

The formal structure on which these semantics are defined (cf. [11]) are tuples of configurations, one per process. A configuration looks as follows:

$$[\, s_i, \langle S_{ij} \rangle_{j=1}^{m(i)} \,]_{i=1}^{n}$$

It consists of the local state s_i of the process P_i and a sequence of $m(i)$ subprocesses S_{ij}, one for each partition π_{ij} of the process. The local state belongs to

the detailed behaviour of the process whereas a subprocess belongs to the global behaviour of the process on the level of one of its partitions. Thus, for each process and its partitions the configuration gives the current state and the current subprocesses. Transitions in the various coordinates are governed by so-called consistency rules. The general format of a consistency rule is

$$\begin{aligned} &\texttt{ProcP} : \texttt{state\_a} \to \texttt{state\_b} \ * \\ &\quad \texttt{ProcQ}_1[\texttt{PART}_1] : \texttt{SubProc}_1 \to \texttt{SubProc}'_1, \\ &\quad \ldots \\ &\quad \texttt{ProcQ}_n[\texttt{PART}_n] : \texttt{SubProc}_n \to \texttt{SubProc}'_n \end{aligned} \tag{2.1}$$

Here $\texttt{state\_a} \to \texttt{state\_b}$ is a $\texttt{ProcP}$ transition, $\texttt{PART}_i$ is a partition of process $\texttt{ProcQ}_i$ and $\texttt{SubProc}_i \to \texttt{SubProc}'_i$ is a transition in the global behaviour or transfer on the level of partition $\texttt{PART}_i$, requiring the various connecting traps have been entered. Via a consistency rule, a combined transition occurs consisting of a state transition and zero or more subprocess changes. In the presence of the consistency rule (2.1) the process $\texttt{ProcP}$ is called manager of the processes $\texttt{ProcQ}_1, \ldots, \texttt{ProcQ}_n$. The latter processes are called employees of $\texttt{ProcP}$. So, an employee has at least one partition and, therefore, global behaviour.

If a process has one or more partitions, the semantics guarantee, a state change in the process only happens if that transition belongs to each current subprocess of the process. In other words, for an employee process the detailed transitions are consistent in all partitions with the current subprocesses for that process. The global transitions correspond to a detailed state transition in some manager process. Such a global transition can only happen if the traps of the relevant subprocesses have been reached. Informally, a manager prescribes new subprocesses to some of its employees by making a suitable state transition; similarly, an employee, by entering a suitable trap, allows a manager to prescribe a new subprocess to it. In other words, a global transition is consistent with the connecting trap that has actually been entered.

In the present setting based on the operational model of [11], in contrast to the operational semantics given in [12], we allow an employee to have more than one manager, even with respect to the same partition. This forms the basis for delegation. Even more extremely, an employee can be its own manager. This is self-management, which can be very useful in combination with delegation.

3 Delegation I

In this section, we consider a delegation example where n clients are served by m servers. For simplicity, all clients behave the same; similarly, all servers behave the same. A broker selects a client in round-robin order and assigns a server to it when necessary. This server is subsequently responsible for handling the needs of the client.

A client can state its interest in a service by 'approaching the desk'. When the needs of the client are clear –possibly after some interaction with the broker, not modeled here– a server is selected by the broker to handle the client's request.

After this delegation, the broker continues its activities. The server takes care of the clients it has been assigned to in a round-robin fashion. Once the client is being served, it releases the service by getting satisfied. The server does not inform the broker that it has become available for serving client i, but the broker will conclude so, if needed, when it sees this client at its desk again.

A formal description of the above in Paradigm involves three process types: client, broker, server. A client process is given by the state-transition diagram of Figure 3.1. It consists of a cycle of 5 states, viz. `no_needs`, `at_desk`, `need_clear`, `service` and `satisfied`, that are subsequently visited. The state `no_needs` is considered to be the starting state of the process. We distinguish n client processes named $\texttt{Client}(1), \ldots, \texttt{Client}(n)$. For presentational reasons we assume in the pictures below the number n to be equal to 5.

Each client process has a partition named `STATUS` that has the three subprocesses `WithoutService`, `Orienting` and `UnderService` given in Figure 3.4. The three subprocesses together describe the global or coarse-grained behaviour of the client process as pictured in Figure 3.5. It simply cycles through its three subprocesses.

The trap `asking` of the subprocess `WithoutService` comprises the local states `at_desk` and `need_clear`. If a client process has entered this trap, i.e. has control in one of the two local states mentioned, it signals that it is ready for moving to a next phase. The traps of the subprocesses `Orienting` and `UnderService` are likewise. When residing in state `need_clear` or in either of the two states `satisfied` and `no_needs`, respectively, the corresponding phase has reached its final stage and the client process is ready to be transferred (in its single `STATUS` partition).

The state-transition diagram of the broker process is given in Figure 3.2. The broker process checks all the client processes and mediates service on their desire. The broker process has no partition.

The state-transition diagram of the m server processes have a similar shape as the broker process. We assume that a server will check in a round-robin fashion whether a client has been assigned to it, see Figure 3.3. A server process has n partitions called $\texttt{CLIENT}(i)$, one per client. Each $\texttt{CLIENT}(i)$ partition has two subprocesses, `Assigned` and `NotAssigned`., see Figure 3.6. Thus, the current subprocesses of a server process together indicate, out of 2^n possibilities, the server's status: for each client whether it is to be served or not. See Figure 3.7 for the global behaviour of a server process in one of its n partitions.

Next, we have to describe the coordination of the n client processes, the broker process and the m server processes. This is done via the so-called consistency rules in Table 1. For the concrete case here, we explain the mechanism of a consistency rule as described abstractly in the previous section. E.g., the consistency rule $(B2)$ of the broker process

$$\begin{aligned}&\texttt{Broker} : \texttt{mediate}(i) \rightarrow \texttt{check}(i+1) \ast \\ &\quad \texttt{Client}(i)[\texttt{STATUS}] : \texttt{Orienting} \rightarrow \texttt{Orienting}, \\ &\quad \texttt{Server}(j)[\texttt{CLIENT}(i)] : \texttt{NotAssigned} \rightarrow \texttt{Assigned}\end{aligned}$$

$(B1)$ $\texttt{Broker} : \texttt{check}(i) \rightarrow \texttt{mediate}(i) *$
$\quad\quad \texttt{Client}(i)[\texttt{STATUS}] : \texttt{WithoutService} \rightarrow \texttt{Orienting}$

$(B2)$ $\texttt{Broker} : \texttt{mediate}(i) \rightarrow \texttt{check}(i+1) *$
$\quad\quad \texttt{Client}(i)[\texttt{STATUS}] : \texttt{Orienting} \rightarrow \texttt{Orienting},$
$\quad\quad \texttt{Server}(j)[\texttt{CLIENT}(i)] : \texttt{NotAssigned} \rightarrow \texttt{Assigned}$

$(B3)$ $\texttt{Broker} : \texttt{check}(i) \rightarrow \texttt{check}(i+1) *$
$\quad\quad \texttt{Client}(i)[\texttt{STATUS}] : \texttt{WithoutService} \not\rightarrow$

$(C1)$ $\texttt{Client}(i) : \texttt{no\_needs} \rightarrow \texttt{at\_desk}$

$(C2)$ $\texttt{Client}(i) : \texttt{at\_desk} \rightarrow \texttt{need\_clear}$

$(C3)$ $\texttt{Client}(i) : \texttt{need\_clear} \rightarrow \texttt{service}$

$(C4)$ $\texttt{Client}(i) : \texttt{service} \rightarrow \texttt{satisfied}$

$(C5)$ $\texttt{Client}(i) : \texttt{satisfied} \rightarrow \texttt{no\_needs}$

$(S1)$ $\texttt{Server}(j) : \texttt{check}(i) \rightarrow \texttt{serve}(i) *$
$\quad\quad \texttt{Client}(i)[\texttt{STATUS}] : \texttt{Orienting} \rightarrow \texttt{UnderService},$
$\quad\quad \texttt{Server}(j)[\texttt{CLIENT}(i)] : \texttt{Assigned} \rightarrow \texttt{NotAssigned}$

$(S2)$ $\texttt{Server}(j) : \texttt{serve}(i) \rightarrow \texttt{check}(i+1) *$
$\quad\quad \texttt{Client}(i)[\texttt{STATUS}] : \texttt{UnderService} \rightarrow \texttt{WithoutService}$

$(S3)$ $\texttt{Server}(j) : \texttt{check}(i) \rightarrow \texttt{check}(i+1)$

Table 1. Consistency rules I

expresses a transfer dependent on three conditions: (i) the broker resides in its local state $\texttt{mediate}(i)$; (ii) the i-th client process, in its single partition `STATUS`, has reached the trap of subprocess `Orienting`; (iii) the j-th server processes has reached in its partition $\texttt{CLIENT}(i)$ the trap of subprocess `NotAssigned`. The effect of the transfer is also threefold: (i) the broker process will move to its local state $\texttt{check}(i+1)$; (ii) the i-th client processes will continue adhering to the subprocess `Orienting` in its partition `STATUS` (though in fact it cannot do anything); (iii) the j-th server will adhere to the subprocess `Assigned` in its partition $\texttt{CLIENT}(i)$.

From the point of view of designing the coordination for the client-broker-server system the consistency rules of Table 1 can be interpreted as follows: The rule $(B1)$ allows for a local transition of the broker process in the local state $\texttt{check}(i)$ provided that the i-th client process has reached the trap `asking` of its subprocess `WithoutService`. So, the current state of client i is either `at_desk` or `need_clear`. The broker will move to the local state $\texttt{mediate}(i)$ to see what are the needs of the client; the client changes, on the level of the partition `STATUS`, from subprocess `WithoutService` to subprocess `Orienting`.

The rule $(B2)$ illustrates the coordination of three processes. If the broker is mediating service for the i-th client, i.e. it resides in the local state $\texttt{mediate}(i)$, and the needs of this client have become clear, i.e. client i has reached the trap `serverClear` of the `Orienting` subprocess that consists of the local state `need_clear`, and the j-th server has not been assigned to this client, i.e. it is prescribed the subprocess `NotAssigned` in the partition for the i-th client, then the $(B2)$ rule can fire. The choice of the particular server is non-deterministic.

The broker moves to the local state `client`$(i+1)$ as it considers its involvement with the i-th client to be finished for the moment. This has been delegated to the j-th server. The ith client is left in the subprocess `Orienting` waiting for service. The j-th server is notified to take care, at the appropriate time, of client i as it now follows the subprocess `Assigned` for this client.

The consistency rule $(B3)$ is an instance of the negative rule format. It expresses that the broker can make a local transition from state `check`(i) to `check`$(i+1)$ in case the i-th client does not reside in the trap `asking` of the subprocess `WithoutService`. Note that the non-determinacy of moving either to state `mediate`(i) or to state `check`$(i+1)$ for the broker process in state `check`(i) is resolved by the i-th client (and, strictly speaking, also involving the server processes). We claim, an equivalent Paradigm model without negative rules can be constructed as well, an issue not treated here.

The consistency rules of the client processes are rather simple. As the clients have not been assigned coordination tasks, their local transitions are unconditional, but for the overall requirement that the transitions belong to the current subprocess of the partition `STATUS`.

The consistency rule $(S1)$ of the server is similar to the rule $(B1)$ of the broker process. The rule covers the case where the server j has been delegated coordination of client i by the broker. Here, we also see a case of self-management: the server process will transfer itself from its subprocess `Assigned` to the subprocess `NotAssigned`. This way, the server will be available for the broker for assignment to client i again, when this client returns to the desk asking for brokerage of another service. By the delegation, the broker is leviated from keeping track of the precise stage of the clients and of the availability of the servers. Based on rule $(S2)$, server j will only move from state `serve`(i) to state `check`$(i+1)$ when client i has reached the trap `ready` of its subprocess `UnderService`. The server then transfers the client to the subprocess `WithoutService`. The local transition of the j-th server from state `client`(i) to state `client`$(i+1)$ has no side-conditions in rule $(S3)$. However, the transition is only possible if, on the level of partition `CLIENT`(i), the server's current subprocess is `NotAssigned`. Thus, the broker resolves the non-determinacy of server j in state `client`(i). If the server is assigned, it will only have the transition to its state `serve`(i) based on rule $(S2)$ as an option; if the server is not assigned, it can only move to state `client`$(i+1)$ by rule $(S3)$.

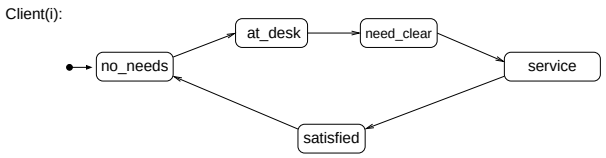

Fig. 3.1. Client STD

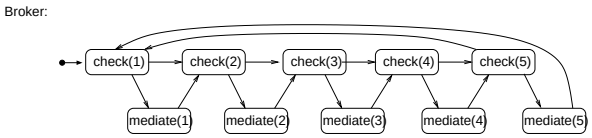

Fig. 3.2. Broker STD

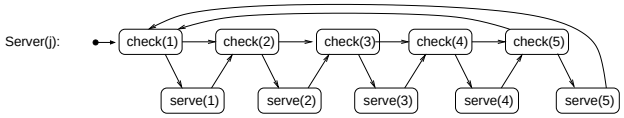

Fig. 3.3. Server STD

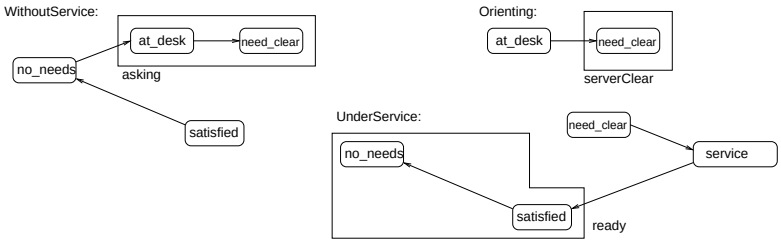

Fig. 3.4. Subprocesses of the Client process for partition `STATUS`

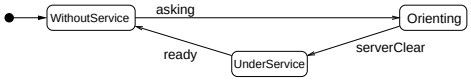

Fig. 3.5. Global behaviour of the Client process on the level of partition `STATUS`

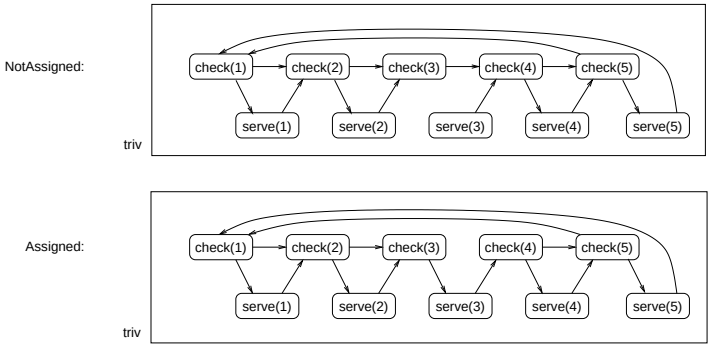

Fig. 3.6. Subprocesses of a Server process for partition `CLIENT`(3)

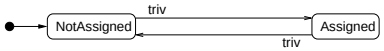

Fig. 3.7. Global behaviour of a Server process on the level of partition `CLIENT`(i)

4 Delegation II

As in the previous section, we have three process types: client, broker, server. Only the broker is slightly different, see Figure 4.1, as it has additional loops in its states `check`(i). Furthermore, we consider the same configuration of n client processes, 1 broker process and m server processes. See Figure 3.1 and Figure 3.3 for the other two process types.

The small difference of the broker has to do with the different details of the delegation. In the previous section, the broker delegated the actual service of a client to a server, without being informed explicitly about the precise beginning of such service. In the current section we let the broker be informed about such a beginning to serve `Client`(i) by `Server`(j). This enables the broker to withdraw the assignment of `Server`(j) to `Client`(i). So now it is the broker who changes subprocess `Assigned` into `NotAssigned`, instead of `Server`(j) doing it. So the (partial) delegation of coordinating `Server`(j)'s global behaviour on the level of partition `CLIENT`(i) does no longer exist: the broker does the complete coordination of this and similar global behaviours. This has the following consequences for the Paradigm model. Partition `STATUS` and the global behavior for it remain unchanged, see Figure 3.4 and 3.5. The servers remain unchanged, see Figure 3.3, but their partitions `CLIENT`$(1), \ldots,$ `CLIENT`(n) are rather different, see Figure 4.2.

Their traps `idle` and `busy` are apparently nontrivial. Trap `idle`, being very large, expresses that the server can do anything but starting to serve `client`(i). So, a new assignment of this very client can happen when needed. Trap `busy` is a small one, expressing that service can be started and completely given, but it cannot be terminated, so the client is not really released - although it can continue as far as state `no_needs`. The slightly adapted global behaviour is given in Figure 4.3. The coordination of the various detailed and global behaviours is described by the consistency rules in Table 2 (rules for Broker and Server processes only).

The differences between the rule set from Table 2 compared to those from Table 1 exactly reflect, on the basis of the new Paradigm model, the new coordination details. The delegation by the broker towards the individual servers of controlling a part of their global behaviour on the level of their partition `CLIENT`(i), is no longer there. Moreover, the delegation by the broker towards the individual servers of controlling a part of the global behaviours of the various clients on the level of partition `STATUS` is changed such that in the new situation any server explicitly informs the broker when it starts or finishes such a delegated task. The consistency rules changed to this aim, are as follows. Rule ($B4$) is added to guarantee the transition from partition `Assigned` to `NotAssigned`, which is no longer the responsibility of a server. Note that only after such a global transition, the corresponding server can release the particular client it is serving. Rule ($S1$) has been simplified, as the global transition from a subprocess `Assigned` to `NotAssigned` is taken care of by the broker. The explicit informing by a server to the broker when it starts or finishes its delegated task, occurs with the (detailed) transition in rules ($S1$) and ($S2$). Thus, a server enters its trap

$(B1)$ $\texttt{Broker} : \texttt{check}(i) \rightarrow \texttt{mediate}(i) *$
$\texttt{Client}(i)[\texttt{STATUS}] : \texttt{WithoutService} \rightarrow \texttt{Orienting}$

$(B2)$ $\texttt{Broker} : \texttt{mediate}(i) \rightarrow \texttt{check}(i+1) *$
$\texttt{Client}(i)[\texttt{STATUS}] : \texttt{Orienting} \rightarrow \texttt{Orienting},$
$\texttt{Server}(j)[\texttt{CLIENT}(i)] : \texttt{NotAssigned} \rightarrow \texttt{Assigned}$

$(B3)$ $\texttt{Broker} : \texttt{check}(i) \rightarrow \texttt{check}(i+1) *$
$\texttt{Client}(i)[\texttt{STATUS}] : \texttt{WithoutService} \not\rightarrow$

$(B4)$ $\texttt{Broker} : \texttt{check}(i) \rightarrow \texttt{check}(i) *$
$\texttt{Server}(j)[\texttt{CLIENT}(k)] : \texttt{Assigned} \rightarrow \texttt{NotAssigned}$

$(S1)$ $\texttt{Server}(j) : \texttt{check}(i) \rightarrow \texttt{serve}(i) *$
$\texttt{Client}(i)[\texttt{STATUS}] : \texttt{Orienting} \rightarrow \texttt{UnderService}$

$(S2)$ $\texttt{Server}(j) : \texttt{serve}(i) \rightarrow \texttt{check}(i+1) *$
$\texttt{Client}(i)[\texttt{STATUS}] : \texttt{UnderService} \rightarrow \texttt{WithoutService}$

$(S3)$ $\texttt{Server}(j) : \texttt{check}(i) \rightarrow \texttt{check}(i+1)$

Table 2. Consistency rules II

`busy` by rule $(S1)$ or its trap `idle` by rule $(S2)$. It is on the basis of a server having entered such a trap, the broker applies rule $(B2)$ or $(B4)$. The other rules do not change.

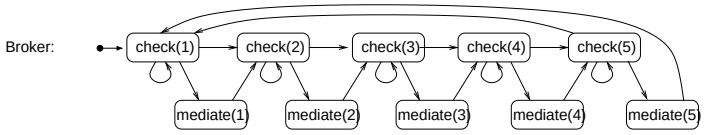

Fig. 4.1. Broker STD II

5 Variations

In this section we illustrate some more flexibility of Paradigm. We discuss three variations on the delegation example of Sections 3 and 4. We describe how one can add other processes in a clean way. First, by addition of a tool that is coordinated by the servers as manager; second, by extension of the configuration with a maintainer that coordinates the servers as its employees. As a third variation, we consider a refinement of the broker in its assignment of servers based on a parameter mechanism.

5.1 Adding an employee process

We consider the case where the servers share some resources that are needed for the servicing of clients. We add two tools, $\texttt{Tool}(1)$ and $\texttt{Tool}(2)$: the one

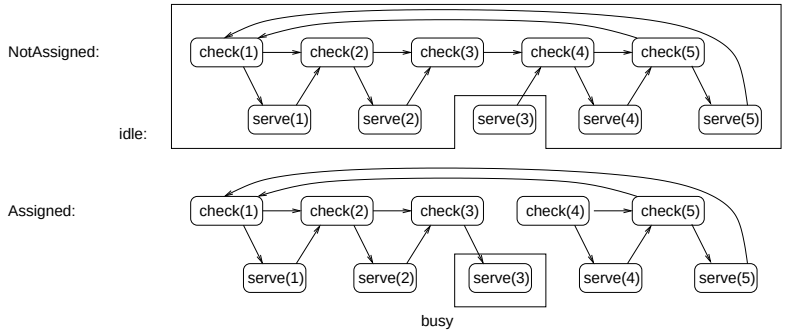

Fig. 4.2. Subprocesses of a Server for partition `CLIENT`(3)

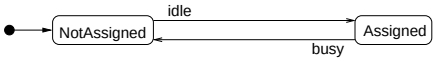

Fig. 4.3. Global behaviour of a Server on the level of partition `CLIENT`(i)

shared amongst the odd-numbered servers, the other shared amongst the even-numbered servers. Each tool will have one partition named `AVAILABILITY` representing its availability, being either `Released` or `Taken`. The state-transition diagram and subprocesses are pictured in Figure 5.1.

The tool alternates between its two states. Servers of the same parity are all managing the corresponding tool in the same partition. When a tool has been released, as signaled by reaching the trap `toBeTaken` of subprocess `Released`, the server can take the tool. The tool is then transferred to its subprocess `Taken`. When the tool has reached its state `occupied`, i.e. the trap `toBeReleased` of subprocess `Taken`, the server can use the tool at its leisure. The server then releases the tool by transferring it to the subprocess `Released`, so that it can move to its local state `free`, where it can be taken again. The transfer of tool subprocesses thus maps 1-1 on the transitions `check`(i) $\rightarrow$ `serve`(i) and `serve`(i) $\rightarrow$ `check`($i+1$).

In order to mix the coordination of the tools by the server and the existing client-broker-server dynamics, we add the signaling of traps and transfer of subprocesses of the tools to the consistency rules of Table 1 of the servers. The rules for the broker and client remain the same. The new rules for the two tool processes are simple as the tool processes have no manager role. See Table 3.

5.2 Adding a manager process

In the previous subsection the management of a tool was done by a collection of servers. Therefore, the consistency rules of the servers were adapted to cope with the new situation. Next, we show how to extend the system by the addition of a process that manages some existing ones. We introduce a maintenance process that influences the dynamics of the servers. The state-transition diagram of the

$(S1)$ $\texttt{Server}(j) : \texttt{check}(i) \rightarrow \texttt{serve}(i) *$
$\texttt{Client}(i)[\texttt{STATUS}] : \texttt{Orienting} \rightarrow \texttt{UnderService},$
$\texttt{Server}(j)[\texttt{CLIENT}(i)] : \texttt{Assigned} \rightarrow \texttt{NotAssigned},$
$\texttt{Tool}(j \bmod 2)[\texttt{AVAILABILITY}] : \texttt{Released} \rightarrow \texttt{Taken}$

$(S2)$ $\texttt{Server}(j) : \texttt{serve}(i) \rightarrow \texttt{check}(i+1) *$
$\texttt{Client}(i)[\texttt{STATUS}] : \texttt{UnderService} \rightarrow \texttt{WithoutService},$
$\texttt{Tool}(j \bmod 2)[\texttt{AVAILABILITY}] : \texttt{Taken} \rightarrow \texttt{Released}$

$(S3)$ $\texttt{Server}(j) : \texttt{check}(i) \rightarrow \texttt{check}(i+1)$

$(T1)$ $\texttt{Tool}(k) : \texttt{free} \rightarrow \texttt{occupied}$

$(T2)$ $\texttt{Tool}(k) : \texttt{occupied} \rightarrow \texttt{free}$

Table 3. Consistency rules for the Server and Tool processes

`Maintainer` process is given in Figure 5.2. (Again, for reasons of presentation, we choose in the figure the number of servers equal to 5 too.) The maintainer in its starting state `no_maint` selects non-deterministically one of the servers. If the selected server is servicing a client, it can finish this. Then the server is brought under maintenance; it resumes servicing as soon as the maintainer process returns to its initial position.

As, with respect to the design choices made here, the maintenance issues are orthogonal to the original dynamics, we simply add a new partition for the servers. This is partition `MAINTENANCE` with subprocesses as in Figure 5.3: subprocess `OutOfService` with trap `stalled` only allows to finish the current service (a graceful interrupt) and the subprocess `Running` with the trivial trap allows all behaviour.

The consistency rules for the `Maintainer` process are not surprising, see Table 4. Note, as the trap used is trivial, rule $(M1)$ is not biased to any of the servers. Any server process can be interrupted for maintenance, based on the maintainer's decision only.

$(M1)$ $\texttt{Maintainer} : \texttt{no\_maint} \rightarrow \texttt{maint}(j) *$
$\texttt{Server}(j)[\texttt{MAINTENANCE}] : \texttt{Running} \rightarrow \texttt{OutOfService}$

$(M2)$ $\texttt{Maintainer} : \texttt{maint}(j) \rightarrow \texttt{no\_maint} *$
$\texttt{Server}(j)[\texttt{MAINTENANCE}] : \texttt{OutOfService} \rightarrow \texttt{Running}$

Table 4. Consistency rules for the Maintainer process

5.3 Parameter-based refinement

Our last variation shows how load balancing or history-based allocation can be handled in Paradigm. A process can be decorated with a parameter representing the local variables or data of the process. The parameter mechanism is reminiscent to process languages as CCS or CSP. For a process with some parameter, $\texttt{Proc}(X)$ say, occurring at the left-hand side of a consistency rule, a so-called

change clause is added to the right-hand side of a consistency rule of the format $\texttt{Proc}(X) \Longrightarrow \texttt{Proc}(X')$. The idea is that the rule can only be fired if the data of `Proc` has value X. As an immediate consequence of firing the rule, the data X of `Proc` will be changed into the data X' on behalf of the relevant manager.

Consider, e.g., in the setting of Section 3, the case where the broker gives a client the same server as before. If the client has not been brokered yet, the broker simply selects one non-deterministically. We introduce the variable H (for history) containing a pair (i, j) if client i was served by server j before. The consistency rules are then augmented with the parameters and change clauses. See Table 5. Now there are two consistency rules in place corresponding to the local transition $\texttt{mediate}(i) \rightarrow \texttt{check}(i+1)$ of the broker: If there exists a pair (i, j) in H than the server j is allocated for client i by rule $(B2a)$; if no such pair (i, k) exists in H, any of the servers can be appointed to deal with client i by rule $(B2b)$. The other consistency rules remain the same.

$(B1)$	$\texttt{Broker}\{H\} : \texttt{check}(i) \rightarrow \texttt{mediate}(i) *$ $\quad \texttt{Client}(i)[\texttt{STATUS}] : \texttt{WithoutService} \rightarrow \texttt{Orienting}$
$(B2a)$	$\texttt{Broker}\{H\} : \texttt{mediate}(i) \rightarrow \texttt{check}(i+1) *$ if $(i, j) \in H$ $\quad \texttt{Client}(i)[\texttt{STATUS}] : \texttt{Orienting} \rightarrow \texttt{Orienting},$ $\quad \texttt{Server}(j)[\texttt{CLIENT}(i)] : \texttt{NotAssigned} \rightarrow \texttt{Assigned}$
$(B2b)$	$\texttt{Broker}\{H\} : \texttt{mediate}(i) \rightarrow \texttt{check}(i+1) *$ if $\nexists k\colon (i, k) \in H$ $\quad \texttt{Client}(i)[\texttt{STATUS}] : \texttt{Orienting} \rightarrow \texttt{Orienting},$ $\quad \texttt{Server}(j)[\texttt{CLIENT}(i)] : \texttt{NotAssigned} \rightarrow \texttt{Assigned}$ $\quad \texttt{Broker}\{H\} \Longrightarrow \texttt{Broker}\{H + (i, j)\}$
$(B3)$	$\texttt{Broker}\{H\} : \texttt{check}(i) \rightarrow \texttt{check}(i+1) *$ $\quad \texttt{Client}(i)[\texttt{STATUS}] : \texttt{WithoutService} \not\rightarrow$

Table 5. Consistency rules for the Broker process with parameter

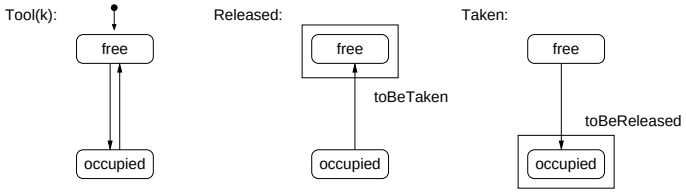

Fig. 5.1. Tool STD and subprocesses of the Tool process

6 Concluding remarks

In this paper we showed how delegation can be modeled with Paradigm. For two basic cases and variations we indicated what the Paradigm model looks like and

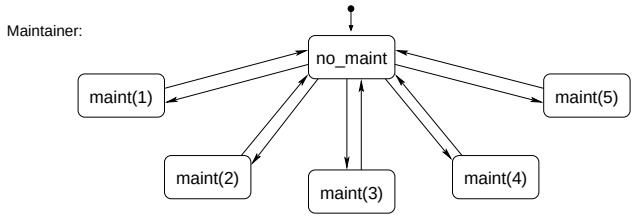

Fig. 5.2. Maintainer STD

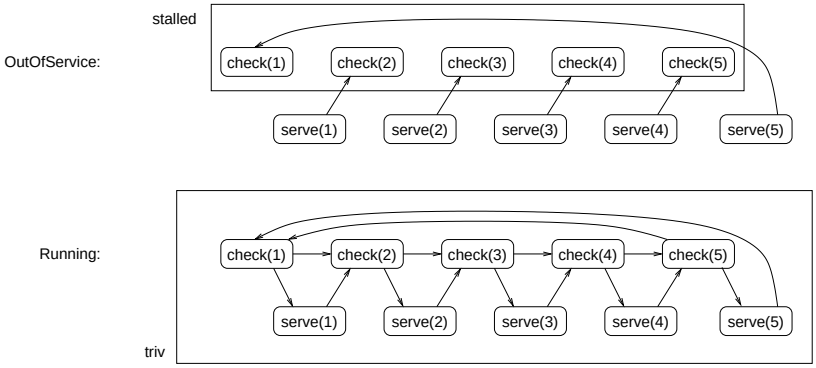

Fig. 5.3. Subprocesses of the Server process in `MAINTENANCE` partition

how the consistency rules capture the coordination of the processes involved. The main point is that local or detailed behaviour of a process that is manager of part of the system, is consistent with the global behaviour of its employee processes, thus assuring horizontal consistency in that part of the system. Manager role and employee role can change dynamically. Paradigm does not only allow for multiple employees of one manager, but also for multiple managers of one employee, thus allowing delegation and even self-management. The advantage of being able to relate local and global behaviour is that of abstraction. Modeling or reasoning about the behaviour of one process does not require to have knowledge in full detail of the other processes that are involved. Here it is vertical consistency between the local behaviour and the global behaviour that matters, as illustrated above for delegation.

In the master's thesis of Van Kampenhout [15], related to work of [1], some initial work has been performed on verification of Paradigm models. In a case-study concerning an insurance company typical properties such as allocation and fairness have been checked. This was done using SMV. It is plausible, that the software architecture arising from a Paradigm model by 'cutting along partitions' is amendable to architecture slicing as proposed in [4] in the context of the Charmy framework. It would be interesting to see how Paradigm and Spin can be exploited, e.g., for the case study reported in [13], where also the issue of coordination and UML is addressed. More generally, with the increased ex-

pressiveness and flexibility of Paradigm, the pattern trail is a promising line of research. Currently, in joint work with Andries Stam, we are adapting Paradigm models for the ToolBus machinery [2, 14] for prototyping purposes.

References

1. J.C. Augusto and R.S. Gómez. A temporal logic view of Paradigm models. In *Proc. SEKE 2002, Ischia*, pages 497–503. ACM, 2002.
2. J.A. Bergstra and P. Klint. The ToolBus coordination architecture. In P. Ciancarini and C. Hankin, editors, *Proc. Coordination '96*, pages 75–88. LNCS 1061, 1996.
3. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language user guide*. Addison Wesley, 1999.
4. M. Caporuscio, P. Inverardi, and P. Pelliccione. Formal analysis of architectural patterns. In *Proc. ESWA*, pages 10–24. LNCS 3047, 2004.
5. J.O. Coplien and N.B. Harrison. *Organizational Patterns of Agile Software Development*. Prentice Hall, 2004.
6. G. Engels and L.P.J. Groenewegen. *Software Process Modelling and Technology*, chapter SOCCA: Specifications of Coordinated and Cooperative Activities, pages 71–102. Research Studies Press, 1994.
7. G. Engels, R. Heckel, and J.M. Küster. The consistency workbench: A tool for consistency management in UML-based development. In P. Stevens, J. Whittle, and G. Booch, editors, *UML 2003*, pages 356–359. LNCS 2863, 2003.
8. P. Clements et al. *Documenting Software Architectures: Views and Beyond*. SEI Series in Software Engineering. Pearson Education, 2002.
9. M. Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd edition)*. Addison Wesley, 2003.
10. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.
11. L.P.J. Groenewegen, N. van Kampenhout, and E.P. de Vink. Coordination in networked organizations: the Paradigm approach. Technical Report CSR 03/13, Technische Universiteit Eindhoven, 2003.
12. L.P.J. Groenewegen and E.P. de Vink. Operational semantics for coordination in paradigm. In F. Arbab and C. Talcott, editors, *Proceedings Coordination 2002*, pages 191–206. LNCS 2315, 2002.
13. P. Inverardi and H. Muccini. A coordination process based on UML and a software architectural description. In H.R. Arabnia, editor, *Proc. PDPTA*, 2000. 7pp.
14. H. de Jong and P. Klint. Toolbus: The next generation. In F.S. de Boer et al., editor, *FMCO 2002, Revised Lectures*, pages 220–241. LNCS 2852, 2003.
15. N. van Kampenhout. Systematic specification and verification of coordination: towards patterns for Paradigm models. Master's thesis, Leiden University, 2003.
16. J.M. Küster. *Consistency Management of Object-Oriented Behavioral Models*. PhD thesis, University of Paderborn, 2004.
17. B. Nuseibeh, S.M. Easterbrook, and A. Russo. Leveraging inconsistency in software development. *IEEE Computer*, 33:24–29, 2000.
18. B. Nuseibeh, J. Kramer, and A. Finkelstein. Viewpoints: meaningful relationships are difficult! In *Proc. ICSE 2003, Portland, Oregon*, pages 676–683. IEEE, 2003.
19. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
20. P.J. Toussaint. *Integration of information systems: a study in requirements engineering*. PhD thesis, Leiden University, 1998.