Restructuring Paradigm models for the ToolBus architecture: a case study

A.W. Stam[1,a], L.P.J. Groenewegen[a] & E.P. de Vink[a,b]

[a] *LIACS, Leiden University, The Netherlands*

[b] *Dept of Math. and Comp. Sc., TU/e, Eindhoven, The Netherlands*

Abstract

In this paper we report on an implementation case study regarding the coordination description language Paradigm. We show how to restructure existing Paradigm models to facilitate their efficient implementation on a software coordination architecture like the ToolBus. The restructuring is in part achieved by transforming hierarchical manager-employee relations into symmetric mutual control and by exploiting structural component similarities. On top of this, we introduce the coordinator-weaver-performer pattern, which combines local separation of coordination and computation with a globally endogenous coordination strategy. The approach is illustrated for a generic servicing example.

Key words: Paradigm, coordination, software architecture, ToolBus

1 Introduction

Within the current software architecture practice, architectures are mostly used for describing static aspects of software systems. Techniques that allow system architects to describe the relation between global and detailed behaviour of software systems and to reason about the dynamics of the system in its entirety, are no common use in software industry.

We believe that the coordination description language Paradigm [7,10,17] is valuable for capturing these dynamics. Paradigm allows for the description of both detailed and global behaviour of individual components and is particularly helpful in enforcing consistency in the behaviour of large sets of interrelated components. However, its practical use is currently limited due to a lack of language and tool support.

[1] Corresponding author: Buijs Ballotsingel 39a, 3112 JB Schiedam, The Netherlands, `andries.stam@xs4all.nl`

In this paper, we investigate a way to implement Paradigm models. We focus on the restructuring of existing Paradigm models to ease their implementation: reducing the number of concepts and structures within Paradigm and decomposing models into separate similar entities. We also develop a specific pattern for the implementation of Paradigm models, which divides components into performers, weavers and coordinators, thereby separating computation from coordination [8]. For the case study, we have selected the ToolBus architecture because of the flexibility it offers in plugging components in and out.

Related work includes that of Kramer, Magee and co-authors. Their aim is to bridge the gap between formal software development and practical software engineering approaches [14,15]. Their architectural view focuses on component interconnection (where Paradigm stresses component interaction) and exploitation of model checking techniques. Successes are reported based on the embedding of the process algebra FSP into architecture description or coordination languages, such as Darwin and Linda, and the support for specification and reasoning provided by the LTS Analyser tool. Although several initial efforts have been carried out [1,12], a similar support for verification of Paradigm models is currently not available.

Other process algebraic approaches to coordination and tooling include the work of Cleaveland and Smolka et al. on the GCCS coordination language and the Concurrent Factory design environment and related software [4,5]. For example, executable code for system models with high-level communication mechanisms, as reported in [16], can be generated automatically and used as subject for further analysis.

The design and reuse of architecture families have been proposed by Gomaa using the concept of the Evolutionary Domain Life Cycle. In [9] this approach is illustrated, as in the present paper, for a client-server example. Also, an intermediate representation formalism called 'architectural type' has been studied in [3], in that case from a process algebraic perspective. Ricci c.s. advocate the usage of concepts from activity theory for coordination purposes in multi-agent systems [18,6]. These approaches focus on studying generic architecture 'patterns' rather than on the modeling as for Paradigm.

The layout of this paper is as follows. In Section 2 we provide an introduction to Paradigm and draw some notes on the ToolBus, as space is limited (however, see [2,13,11]). In Section 3 we introduce the Paradigm model for a client-server architecture, that will be used as a running example in the rest of the paper. In Section 4 we restructure the model by introducing a symmetric version of Paradigm, in which managers and employees are replaced by actors, hence reducing the amount of concepts and structures. In Section 5 we provide a way to decompose state transition diagrams within Paradigm models. In Section 6, the coordinator-weaver-performer pattern is explained.

We present conclusions in Section 7.

2 Background

Paradigm is a coordination description language based on the notions of subprocess and trap [7,10,17]. It ties the local, fine grained behaviour of a component to several global, coarse-grained behavioural views, seen from different points in its environment. A Paradigm model factors a software architecture into its basic manager-employee relations.

The modeling of component interaction starts from the description of the components as a state-transition diagram (STD). From the perspective of interaction and its coordination, not all detailed behaviours matter, but rather the global behaviour in terms of phases. The coordination of components requires a description of the interplay in terms of phases. The basic idea underlying Paradigm is that, for the particular view on the system, at every moment in time, there is a single component, called the manager, in charge of the phase changes of the other, relevant components, referred to as its employees.

In general, a software architecture can be sliced along several dimensions. This is reflected in Paradigm by a component being involved in several partitions. The partition captures a particular view on the components, seen from the surrounding system. Each partition has one or more managers and belongs to one employee. The actual management can switch from one manager to another, always one at a time. In this manner, the interplay of the local, state-based behaviour of the managers together with the global phase-based behaviour of the employees comprise the coordination for the partition. In Paradigm this is described using so-called consistency rules.

A phase of a component is represented in Paradigm by the notion of a subprocess. A subprocess is given by a sub-STD of the STD of the component; it consists of some of the states and some of the transitions between them. The component STD will have, for a particular partition, one or more subprocesses. These subprocesses are overlapping. When an employee is in a certain phase, i.e., the component is executing a subprocess, it may at some point be ready for a next phase. However, it is not at the employee's decision to transfer to the corresponding subprocess. It is up to the manager of the partition to coordinate this.

Paradigm provides the concept of a trap for employee-manager communication. Each subprocess of an employee distinguishes a number of subsets of its state space, called traps. Transitions of the subprocess can lead into a trap, but never go out, whence the name. When a component in a subprocess resides in a trap, this reflects its readiness for a transfer to a next phase.

Once the manager of the partition notices all the employee components having entered particular traps, the manager can execute a consistency rule. In Paradigm, a consistency rule has the format

$$\mathtt{M}\colon \mathtt{s} \to \mathtt{s}' \;*\; \mathtt{E}_1 : \mathtt{S}_1 \xrightarrow{\theta_1} \mathtt{S}_1', \;\ldots,\; \mathtt{E}_n\colon \mathtt{S}_n \xrightarrow{\theta_n} \mathtt{S}_n'$$

meaning that the manager component M can make a local transition from its state s to its state s' if the employee components $\mathtt{E}_1$ to $\mathtt{E}_n$ have reached the traps θ_1 to θ_n in their subprocesses $\mathtt{S}_1$ to $\mathtt{S}_n$, respectively, after which these components will continue with executing the subprocesses $\mathtt{S}_1'$ to $\mathtt{S}_n'$. State to state transitions $s \to s'$ describe the local behaviour of a component; transfers from subprocesses to subprocesses capture, for a particular partition, its global behaviour. In general, zero, one or more consistency rules may apply.

In our case study, we have used the ToolBus to illustrate our implementation strategy for symmetric, decomposed Paradigm models. The ToolBus is an integration platform for the coordination of components, possibly written in different programming languages, developed by Klint and co-workers for over a decade [2,13,11]. It provides a programmable bus for driving the cooperation of the component activities. In this way, components can be easily plugged in and out. ToolBus supports the connection of industrial-size information systems and is suitable for high-performance web-based distributed applications.

The ToolBus runs processes written in TScript, a process algebra based interpreted scripting language. The processes and external components, referred to as tools in ToolBus jargon and represented by interface processes on the bus, exchange data in a uniform ATerm based format using TCP/IP sockets. This way the tools are encapsulated; implementation details of the external components can be hidden completely. As such the ToolBus can be seen as an instance of the coordination vs. computation paradigm [8]: Coordination is captured by the processes on the bus, computation is embodied by the external components. Changing the coordination in the ToolBus is less flexible, though, as it can be done only by starting a new ToolBus.

3 A client/server example

Throughout this paper, we will use the following running example of a Paradigm model. It consists of a server that serves three clients. The server is modeled as a manager, while the clients are modeled as employees.

Clients are described by the STD of Figure 3.1. They continuously cycle through the following steps:

- `NoNeeds`: the client does not need attention of the server.
- `AtDesk`: the client wants to be served (is waiting at the desk).
- `NeedClear`: the client has told the server what kind of service it wants.

- `UnderService`: the client is being served by the server.
- `Satisfied`: the client has got enough attention from the server.

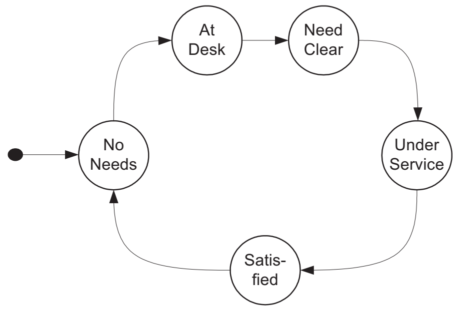

Fig. 3.1. The client state transition diagram

A partition for the clients is described in Figure 3.2. Within the partition, three subprocesses are identified: In `Without`, the client has no contact with a server yet, in `Orienting`, the client explains its needs to the server, while in `With`, the client is being served. The traps `asking`, `questionclear` and `ready` connect these subprocesses to each other. The corresponding global behaviour can be found in Figure 3.3.

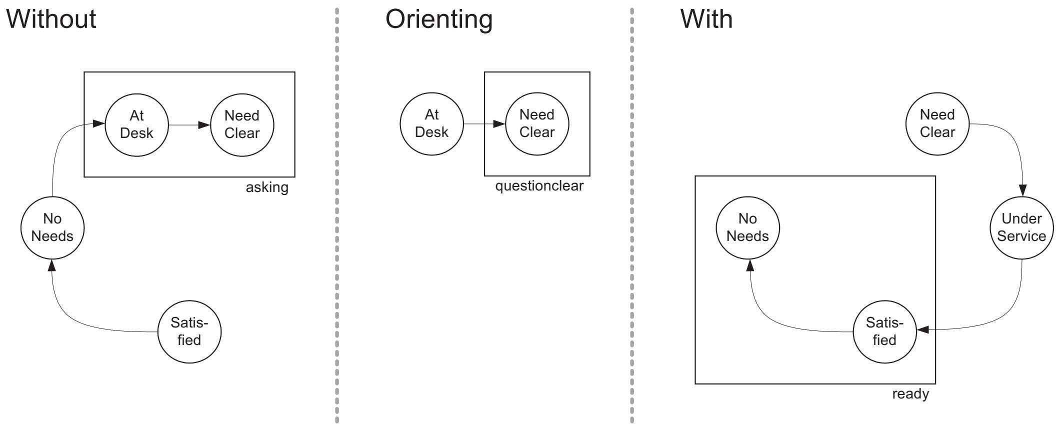

Fig. 3.2. The client partition

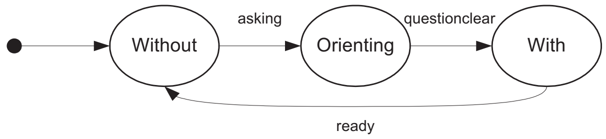

Fig. 3.3. The global behaviour of a client

The STD for the server is shown in Figure 3.4. The server uses a round robin strategy for addressing all clients, starting with Client 1. For each client, the server checks whether this client wants attention, and if so, it listens to its request and performs the requested service. If the client does not need attention, the server moves to the next client.

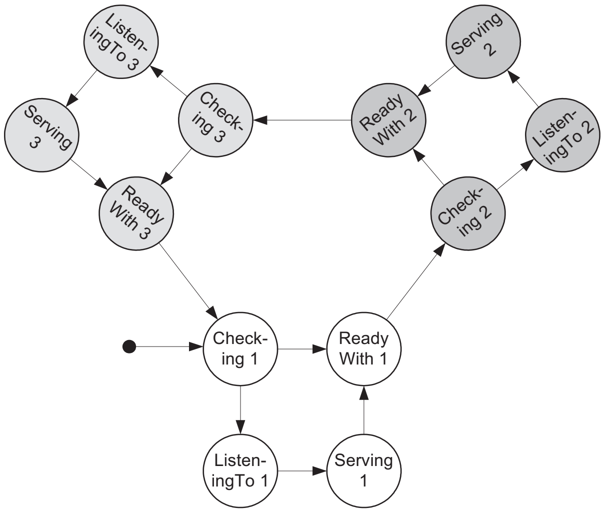

Fig. 3.4. The server state transition diagram

The manager-employee communication is described by the set of rules in Table 1. Rules ($R1$), ($R2$) and ($R3$) guarantee that a client asking for a service will be served correctly: If client i is in trap `asking` of its subprocess `Without` and the server is in state `Checking`(i), then the server will make a transition to state `ListeningTo`(i) and client i will be changed to subprocess `Orienting`, according to rule ($R1$). If client i reaches trap `questionclear` (and the server is in state `ListeningTo`(i)), client i will be changed to subprocess `With` and the server will go to state `Serving`(i), as rule ($R2$) prescribes. Rule ($R3$) ensures that if client i reaches trap `ready` and the server is in state `Serving`(i), then the server will go to state `ReadyWith`(i) and client i is changed to subprocess `Without`. Rule ($R4$) enables the server to serve the next client once the current client has been settled (calculating with indices modulo 3). The last rule, ($R5$), ensures that the server can skip a client only if that client is not asking for service.

($R1$) $\texttt{Server:Checking}(i) \rightarrow \texttt{ListeningTo}(i) \ * \ \texttt{Client}(i)\texttt{:Without} \overset{\texttt{asking}}{\rightarrow} \texttt{Orienting}$

($R2$) $\texttt{Server:ListeningTo}(i) \rightarrow \texttt{Serving}(i) \ * \ \texttt{Client}(i)\texttt{:Orienting} \overset{\texttt{questionclear}}{\rightarrow} \texttt{With}$

($R3$) $\texttt{Server:Serving}(i) \rightarrow \texttt{ReadyWith}(i) \ * \ \texttt{Client}(i)\texttt{:With} \overset{\texttt{ready}}{\rightarrow} \texttt{Without}$

($R4$) $\texttt{Server:ReadyWith}(i) \rightarrow \texttt{Checking}(i+1)$

($R5$) $\texttt{Server:Checking}(i) \rightarrow \texttt{ReadyWith}(i) \ * \ \texttt{Client}(i)\texttt{:Without} \overset{\texttt{asking}}{\not\rightarrow}$

Table 1
Client-Server Consistency Rules

4 Symmetrization of coordination

Within Paradigm, as the running example illustrates, there is a difference between the description of a manager and that of an employee. Typically, employees are modeled with detailed and global behaviour by adding partitions. For their detailed STD, subprocesses and connecting traps are defined, from which the global behaviour can be obtained. For managers, an STD directly describes its behaviour. The communication between managers and employees is described by rules that relate transitions of the manager STD to transitions in the global behaviour of employees.

However, for our purposes, it is more convenient to regard managers and employees as similar entities and describe them in the same way, both with local *and* global behaviour. This approach has technical benefits, since there is no need to implement managers different from employees. We reformulate the Paradigm model of our client-server example into a *symmetric* Paradigm model. Here, both managers and employees have partitions (hence detailed and global behaviour) and the communication rules are solely defined in terms of actors (referring to managers and employees collectively), moving from one subprocess to another, triggered by other actors that enter a trap. Thus, a manager-side partition and an employee-side partition are related to each other, but management role and employee role switch repeatedly back-and-forth between manager and employee.

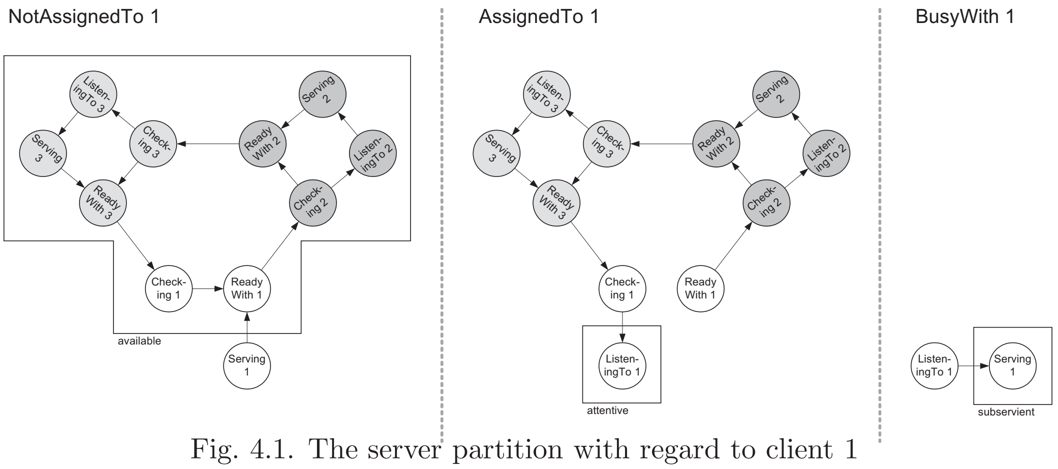

Fig. 4.1. The server partition with regard to client 1

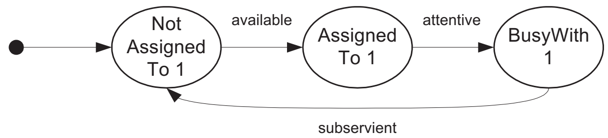

Fig. 4.2. The global behaviour of a server with regard to client 1

The original server is extended to a server with three partitions, one for each client it can serve. The relation between the behaviour of the server and that of the clients is ensured by the consistency rules for this symmetric Paradigm model, described in Table 2.

For client 1 we have the subprocesses as given in Figure 4.1. Three subprocesses are identified. In `NotAssignedTo`(1) the server is not assigned to client 1. Note the missing state `ListeningTo`(1). If the server is in trap `available` and client 1 enters trap `asking` in its subprocess `Without`, the server will change to subprocess `AssignedTo`(1) according to rule ($R2'$) of Table 2. There, it will eventually enter the trap `attentive`. As soon as client 1 enters trap `questionclear` in the subprocess `Orienting`, the server will change to subprocess `BusyWith`(1) by rule ($R4'$), in which it will eventually enter trap `subservient`. By rule ($R6'$) it will change back to subprocess `NotAssignedTo`(1) once client 1 has entered trap `ready` of the subprocess `With`. In Figure 4.2 the corresponding global behaviour of the server is shown.

($R1'$) $\quad \texttt{Server: NotAssignedTo}(i) \stackrel{\texttt{available}}{\longrightarrow} \texttt{NotAssignedTo}(i)\ *$
$\qquad \texttt{Client}(i)\texttt{: With} \stackrel{\texttt{ready}}{\longrightarrow} \texttt{Without}$

($R2'$) $\quad \texttt{Client}(i)\texttt{: Without} \stackrel{\texttt{asking}}{\longrightarrow} \texttt{Without}\ *$
$\qquad \texttt{Server: NotAssignedTo}(i) \stackrel{\texttt{available}}{\longrightarrow} \texttt{AssignedTo}(i)$

($R3'$) $\quad \texttt{Server: AssignedTo}(i) \stackrel{\texttt{attentive}}{\longrightarrow} \texttt{AssignedTo}(i)\ *$
$\qquad \texttt{Client}(i)\texttt{: Without} \stackrel{\texttt{asking}}{\longrightarrow} \texttt{Orienting}$

($R4'$) $\quad \texttt{Client}(i)\texttt{: Orienting} \stackrel{\texttt{questionclear}}{\longrightarrow} \texttt{Orienting}\ *$
$\qquad \texttt{Server: AssignedTo}(i) \stackrel{\texttt{attentive}}{\longrightarrow} \texttt{BusyWith}(i)$

($R5'$) $\quad \texttt{Server: BusyWith}(i) \stackrel{\texttt{subservient}}{\longrightarrow} \texttt{BusyWith}(i)\ *$
$\qquad \texttt{Client}(i)\texttt{: Orienting} \stackrel{\texttt{questionclear}}{\longrightarrow} \texttt{With}$

($R6'$) $\quad \texttt{Client}(i)\texttt{: With} \stackrel{\texttt{ready}}{\longrightarrow} \texttt{With}\ *$
$\qquad \texttt{Server: BusyWith}(i) \stackrel{\texttt{subservient}}{\longrightarrow} \texttt{NotAssignedTo}(i)$

Table 2
Symmetric Client-Server Consistency Rules

Note the difference with the rules for the asymmetric Paradigm model: the rules ($R1$), ($R2$) and ($R3$) in the asymmetric model are now replaced by rules ($R1'$) to ($R6'$) in the symmetric model. Rule ($R4$) in the asymmetric model is not needed in the symmetric version, since it concerns a local step. Rule ($R5$) in the asymmetric version is not needed as well, since in the symmetric version the subprocess-trap structure guarantees that the server will not skip a client to which it is assigned.

In Figure 4.3, a detailed view of the symmetric communication between the server and client 1 in our running example is shown. As can be seen from this picture, if the client enters a trap, the subprocess of the server changes.

Vice versa, if the server enters a trap, the subprocess of the client changes. The description of both actors is of the same kind, and so can be their implementation. Note that the communication between the actors consists solely of information about actors entering traps. So, all communication between actors is at a global level of behaviour.

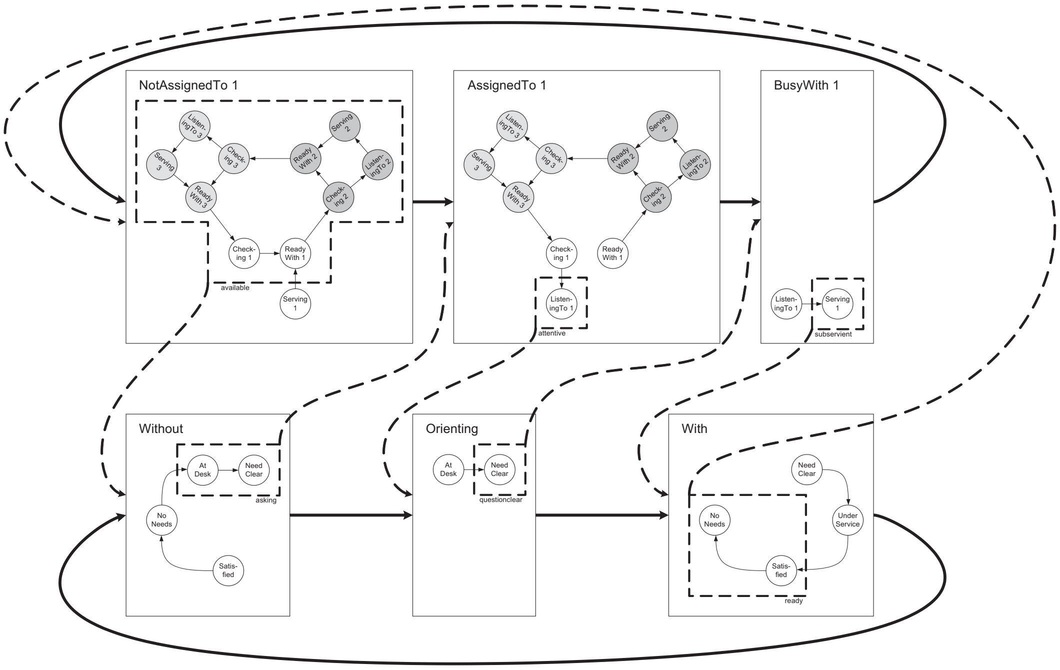

Fig. 4.3. The symmetric communication between a server and its client 1 in detail

5 Decomposition of coordination

The server in our running example serves exactly three clients. In the symmetric Paradigm-model, the server has three partitions: one for each client it serves. These partitions highly overlap, as the partition for one client also has to keep track of states that are relevant for the coordination of the two other clients. Moreover, the server uses a round robin strategy, but the coordination for this strategy is not separated from the coordination of the clients. This makes the entire model expand, makes it less comprehensible and hinders maintaining future changes to the model, e.g. regarding the number of clients or the strategy of servicing. More importantly, it also complicates an easy implementation of the Paradigm model, since all partitions have many states to keep track of, but few things to coordinate, whereas all client partitions have the same shape, but differ in their state names.

Therefore, we decompose the server STD into smaller parts. Instead of specifying the entire state transition diagram within each partition, we only mention the states that are relevant for the coordination and introduce an *environment* in which all states are kept that do not matter for the coordination. We keep the partition for each client (as we already had) and we introduce a new one for the round robin strategy.

Within Figures 5.1 and 5.2 we show the generic STD and partition for coordinating a client. Some arrows have no begin or end state: they are linked to the environment, which contain states that are irrelevant for this partition.

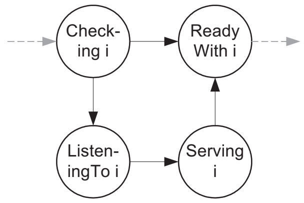

Fig. 5.1. The generic server state transition diagram with regard to clients

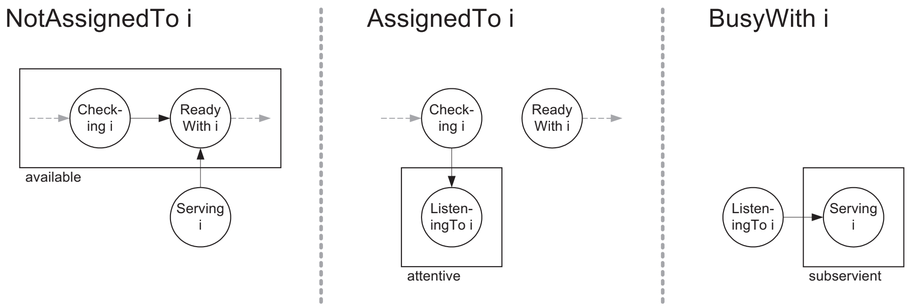

Fig. 5.2. The generic server partition with regard to clients

The round robin strategy is coordinated by means of a separate partition, shown in the left part of Figure 5.3. It consists of a single subprocess with one trivial trap. In the right part, nondeterministic selection is shown as an alternative for the round robin strategy. It illustrates the fact that decomposition of the entire STD facilitates the modeling of variants. The individual pieces can be combined to retrieve the original STD, by identifying certain states. E.g., in our running example, states `1a` and `1b` of the round robin partition correspond to states `Checking` and `ReadyWith` of the coordinator for client 1, respectively.

6 Separation of computation and coordination

In the previous two sections, we described transformations of the original Paradigm model in order to facilitate its implementation. We have experimented with the ToolBus to achieve this. In particular, we have exploited

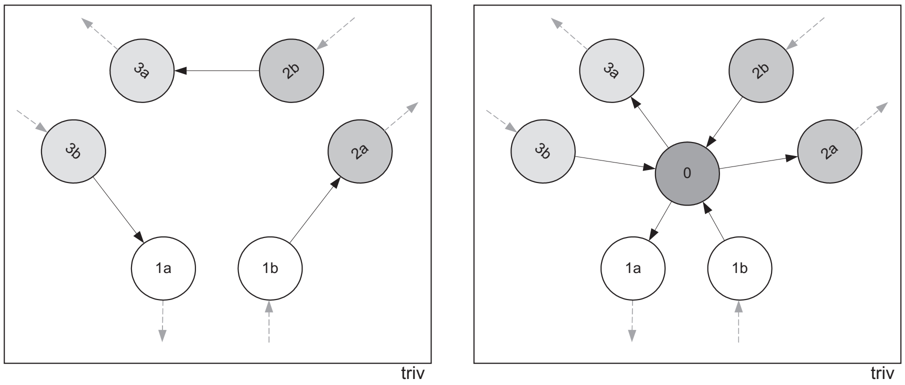

Fig. 5.3. Server partition for the round robin strategy and an alternative strategy

the generic communication mechanism between entities that it provides. We have used the coordinating processes within the ToolBus *solely* for coordinating the communication between components. Hence, they do not perform the coordination that is part of the Paradigm model (in terms of partitions and rules) – external components take care of this part. We have worked this way to ensure that changes in the Paradigm model can be applied flexibly by plugging components in and out, without having the necessity to change the processes within the ToolBus.

Nevertheless, also for the Paradigm model we have kept separation of coordination from computation. This is done in another way, which is shown in Figure 6.1.

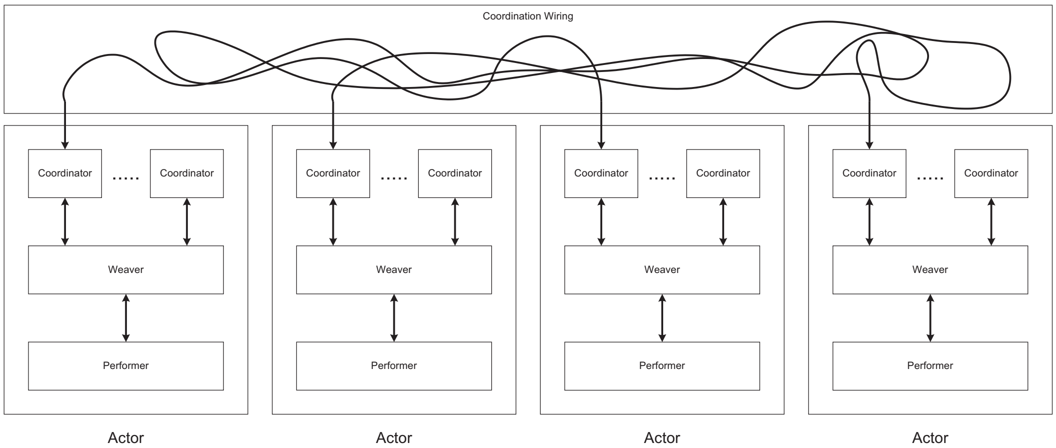

Fig. 6.1. The architecture of the current Paradigm implementation

The ToolBus is only concerned with the communication between components that reflects the coordination interaction (the wiring). In our approach, each actor in the symmetric Paradigm model is split up into several components: a *performer*, one or more *coordinators* and (in case there are more coordinators) a *weaver*.

Performers A performer is responsible for the execution of an STD that corresponds to the set of currently allowed steps within the local behaviour of the actor. It has knowledge of the meaning of each transition in it and, if necessary, makes a choice when multiple transitions are possible. The performer gets its state transition diagram from a coordinator or from the weaver. The state transition diagram is communicated as a set of rules, e.g., $A \to \{B, C\}$, which means that from state A, the performer can make a transition to either B or C. The performer communicates each state transition to the weaver. It is not aware of any coordination.

Coordinators A coordinator is responsible for the communication and enforcement of the information regarding a single partition. It gets information about the state the performer is in, either directly from the performer or via the weaver. It knows about traps and subprocesses within its partition. If the performer performs a transition that corresponds to entering a trap, the coordinator communicates this 'trap entering' to other coordinators to which it is connected. In turn, other coordinators communicate their trap information to this coordinator. Based on this trap information from other coordinators, the coordinator communicates the next current subprocess (as a rule set) to the performer (possibly via the weaver).

Weavers If within a Paradigm model, some actor has more than one partition, more coordinators are needed that will all communicate to a single performer. In such a case, a weaver combines the currently prescribed subprocesses (rule sets) of individual coordinators to a single subprocess (rule set) that is communicated to the performer. The weaver knows about the equality relations between states and knows when certain states lie in the environment of a coordinator. The weaver is fully transparent: both the coordinator and the performer communicate with it in the same manner as if they would communicate to each other directly. Note, the weaver only weaves the current subprocesses *as they are* without further consistency adjusting. It is the Paradigm model itself that takes care of consistency.

An example of the working of a weaver for the server in our running example is given in Figure 6.2. Suppose the coordinators prescribe the given subprocesses: `NotAssignedTo`(1), `AssignedTo`(2), `BusyWith`(3) and the single subprocess for the round robin pattern. By its knowledge of the state mappings, the weaver combines the separate prescriptions to a single prescription, which it communicates to the performer. If the performer performs a transition, it communicates its state to the weaver, which translates the actual state to states known by each coordinator.

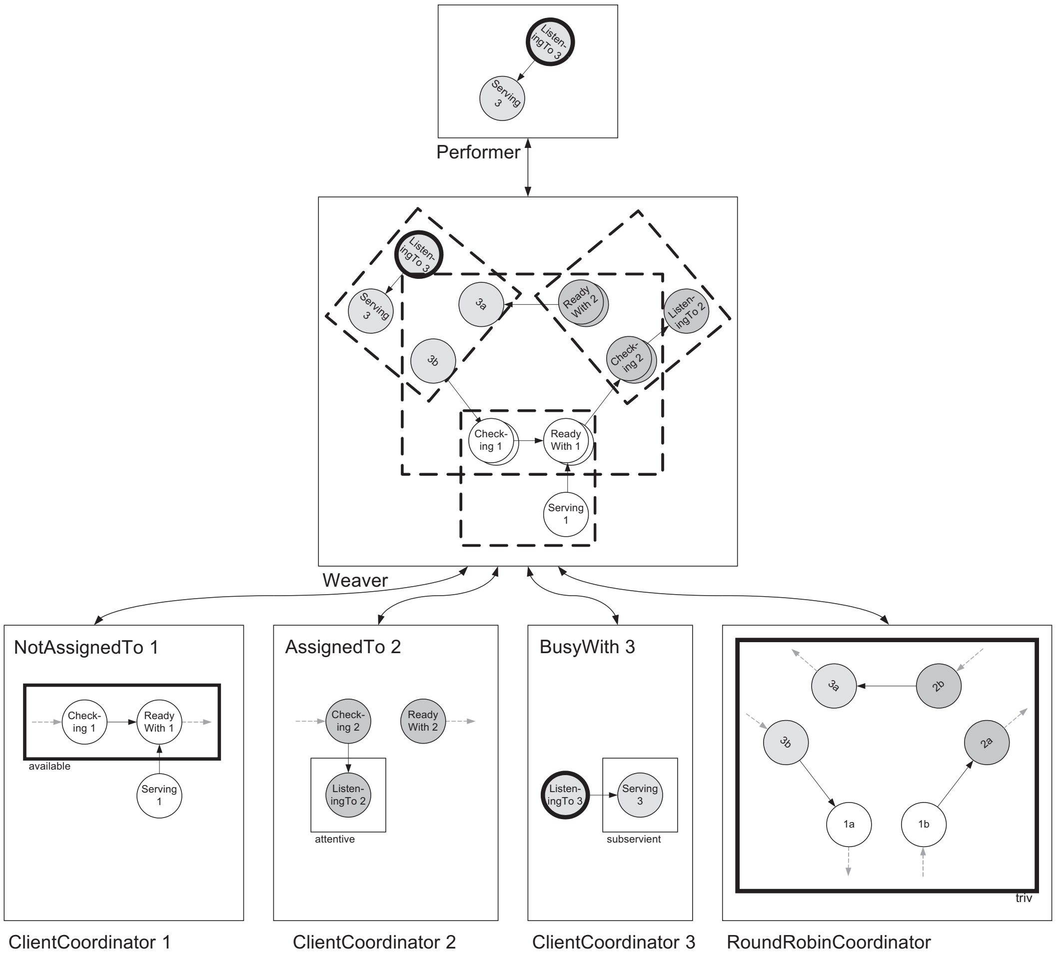

Fig. 6.2. The working of a weaver

7 Concluding remarks

Our research question concerned the restructuring of the coordination description language Paradigm. Typical for Paradigm are, amongst other, manager-employee relations. It is likely that the proposed realization method carries over to other IWIM-style coordination languages that allow multiple managers. We discussed three steps for efficiently implementing Paradigm models.

First, by making Paradigm models symmetric, we ease implementation by reducing the number of concepts and structures in Paradigm. Instead of developing code for managers and employees separately, we can now implement actors that are either manager or employee or both. Thus, as managers and employees have local as well as global behaviour, both have partitions, subprocesses and traps. All the coordination of the actors is done by communicating the entering of traps.

Second, by decomposing (large) state transition diagrams with several partitions into separate, smaller and similar entities, we can reduce the amount of states that needs to be known by an actor in the system. Thus we facilitate reuse of code and we gain in flexibility. Although the way of decomposing the entire STD of the server is fairly straightforward in the case of this example, a general decomposition strategy for state transition diagrams seems feasible, but remains future work.

Third, we have developed the coordinator-weaver-performer pattern, which splits each actor into three components: a Performer, which knows about the semantics of each transition and state, a set of Coordinators that each know about a certain partition (a certain set of global information, possibly needed by other actors) and a Weaver which translates information between coordinators and the performer. This way, we separate local computation from local coordination while keeping the global coordination of the model endogenous.

This case study is the first attempt to implement Paradigm models in an existing coordination framework. We have used the ToolBus to provide the communication between all components. On top of the flexibility of the ToolBus of plugging in and out of components, our approach adds variation of coordination to the ToolBus without having to start a reconfigured ToolBus architecture.

An interesting further topic would be to investigate the implementation of Paradigm on top of other coordination frameworks, like tuple spaces or channel architectures.

Within the near future, our work will go into two directions. On the one hand, we will investigate the general semantical foundation for the restructuring techniques described above. On the other hand, we plan to implement more complex Paradigm models with delegation and evolution examples in order to validate our approach to facilitate prototyping of, reasoning about and animation of Paradigm models.

References

[1] J.C. Augosto and R.S. Gomez. A temporal logic view of Paradigm models. In *Proc. SEKE 2002, Ischia*, pages 497–503. ACM Press, 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] M. Bernardo, P. Ciancarini, and L. Donatiello. On the formalization of architectural types with process algebras. In *Proc. SIGSOFT 2000, San Diego*, pages 140–148. ACM, 2000.

[4] R. Cleaveland, P.M. Lewis, S.A. Smolka, and O. Sokolsky. The Concurrency Factory software development environment. In T. Margaria and B. Steffen, editors, *Proc. TACAS '96*, pages 391–395. LNCS 1055, 1996.

[5] R. Cleaveland and S. Sims. Generic tools for verifying concurrent systems. *Science of Computer Programming*, 42:39–47, 2002.

[6] E. Denti, A. Ricci, and R. Rubino. Integrating and orchestrating services upon an agent coordination infrastructure. In A. Omicini, P. Petta, and J. Pitt, editors, *Proc. ESAW 2003*, pages 228–245. LNCS 3071, 2003.

[7] G. Engels and L. Groenewegen. SOCCA: Specifications of coordinated and cooperative activities. In A. Finkelstein et al., editor, *Software Process Modelling and Technology*, pages 71–102. Research Studies Press, 1994.

[8] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35:97–107, 1992.

[9] H. Gomaa and G.A. Farrukh. A reusable architecture for federated client/server systems. In *Proc. SSR '99, Los Angeles*, pages 113–121. ACM, 1999.

[10] L. Groenewegen and E. de Vink. Operational semantics for coordination in Paradigm. In F. Arbab and C. Talcott, editors, *Proc. Coordination 2002*, pages 191–206. LNCS 2315, 2002.

[11] H. de Jong and P. Klint. Toolbus: The next generation. In F.S. de Boer et al., editor, *Proc. FMCO 2002*, pages 220–241. LNCS 2852, 2003.

[12] N. van Kampenhout. Systematic specification and verification of coordination: towards patterns for Paradigm models. Master's thesis, LIACS, Leiden University, 2003.

[13] P. Klint. *The ToolBus Programming Guide*. CWI, 2002. Retrievable from `http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/ToolBus`.

[14] J. Kramer and J. Magee. Modelling for mere mortals. In W.R. Cleaveland, editor, *Proc. TACAS '99*, pages 1–18. LNCS 1579, 1999.

[15] J. Kramer, J. Magee, and S. Uchitel. Software architecture modeling and analysis: a rigorous approach. In M. Bernardo and P. Inverardi, editors, *Proc. SFM 2003*, pages 44–51. LNCS 2804, 2003.

[16] A. Ray and R. Cleaveland. Architectural Interaction Diagrams: AIDs for system modeling. In *Proc. ICSE 2003, Portland, Oregon*, pages 396–407. IEEE, 2003.

[17] P.J. Toussaint. *Integration of information systems: a study in requirements engineering*. PhD thesis, Leiden University, 1998.

[18] M. Viroli and A. Ricci. Instructions-based semantics of agent mediated interaction. In *Proc. AAMAS 2004, New York*, pages 102–109. IEEE, 2004.