Operational Semantics for Coordination in 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. Paradigm is the modeling language of SOCCA regarding communication, coordination and cooperation. A transition system or STD-based operational semantics for Paradigm is proposed and illustrated with basic producer-consumer examples. The proposed semantics for Paradigm, in contrast to other approaches, provides a sound basis for reasoning about system dynamics and system comparison.

1 Introduction

Coordination belongs to the key concepts of computer science. Software systems usually consist of many components, communicating with each other and with their environments. Coordination then is the consistent organization of the communication and its effects, such that required cooperation between all components involved is established.

As this consistent organization is crucial for all software systems, the importance of coordination, although only recently recognized as an issue of its own, is beyond dispute. Equally beyond dispute is coordination is being formulated and studied through an impressively large collection of formalisms, languages, approaches, architectural constructs. Examples thereof are team automata [2], Manifold [1], Statecharts [11], CORBA [3], blackboard architectures [14]. Thus, coordination in highly complex communicative systems is indeed specified, designed and built. And what is more, these highly complex systems are in use, to the perhaps not entire, but at least reasonable satisfaction of their stake holders.

Notwithstanding the substantial success of such systems, it is also clear that the essence of coordination, and of the communication it is based on, is still poorly understood. Concerning coordination one can make the following observation. We specify and build it, but we do not yet sufficiently understand it.

The observation follows for instance from the difficulties one has in object-orientation, formulating a proposal for a correct as well as satisfying semantics of all UML diagrams and diagram parts related to behaviour and coordination thereof. What is even more difficult, is providing some such proposal where in addition the semantics for the separate diagrams and diagram parts are well-integrated, resulting in necessary behavioural and communicative consistency between these parts and diagrams, such that required coordination can be ensured [9].

Among the many object-oriented approaches preceding the UML, there is one taking a somewhat unusual approach toward coordination. It is called SOCCA —Specification Of Coordinated Cooperative Activities, cf. [8]. For the communication, coordination and cooperation it uses a restricted form of Paradigm (PARallelism, its Analysis, Design and Implementation by a General Method) as a sublanguage. Recently, two different formal specifications of SOCCA have appeared, one in Z [12], and one in EER/Gral [16] which is also rather Z-like. Another, recent and independent approach to the formal description of Paradigm is reported in [7].

It is the merit of Paradigm, that a SOCCA model can provide an uncommonly clear insight in the coordination of the multi-threaded objects. The communication on which the coordination is built, thereby can range from synchronous to completely asynchronous, with many gradations of asynchronism in between. In addition, as communication often takes place distributed over a certain time interval, it makes sense to discriminate between communication at the start or at the end or somewhere in the middle of such an interval. It is Paradigm offering the notions to do so in a well-structured and clear way. Hence, a Paradigm modeler (or a SOCCA modeler for that matter) can devise and reason about the coordination he wants to achieve, based on these notions. See for example [10].

In this paper we want to concentrate on Paradigm, without any details as well as restrictions related to object-orientation. Through Paradigm one can reformulate the sequential behaviour within the various STD components (State Transition Diagrams) of a model in a more global, phase-like manner. This way the modeling perspective is lifted from behaviour to behaviour of behaviours.

It is the idea of behaviour of behaviours that is the kernel of Paradigm. Behaviour of (the underlying, original) behaviours is actually a global view on the original behaviours. Moreover, the global view is such, that even during execution the global behaviour as it is viewed as occurring, remains consistent with the original behaviour that is actually occurring on the underlying, more detailed level. To that aim, Paradigm has been constructed such that consecutive phases always have a certain overlap. The larger the overlap, the more asynchronous is the corresponding communication. This allows one to formulate the necessary coordination in terms of these global behaviour descriptions, the phases and their overlaps, thereby providing the cooperation one is after. What makes Paradigm so different from other approaches such as CCS and CSP (cf. [13, 6]) and statecharts, is the idea of an overlap between the phases. One can build the same communication by means of other formalisms, but one nearly looses the idea of the phases and one completely looses the idea of the overlaps, thus seriously hampering understanding the coordination achieved through the communication. By relating the two types of behaviour consistently to each other as in Paradigm, one can not only understand the cooperation and coordination one is after, but one can even enforce it.

In the literature Paradigm (without SOCCA) has been presented in an informal manner only [15]. Here we shall give a formal presentation of Paradigm based on STDs. By means of the Paradigm notions, coordination of sequential components (STDs) can be specified as a requirement in a rather clear way. Moreover we add semantic formality to the Paradigm notions by means of operational semantics. This provides the means to transform the coordination requirements straightforwardly into a program that meets the coordination requirements. So the programs for the components indeed behave and communicate sufficiently consistent. Paradigm's STD basis together with its clean operational semantics make formal reasoning about the programs and their interaction relatively straightforward, a goal also underlying motivating the operational semantics of [5] for Manifold.

The paper has the following structure. After Section 1, Section 2 presents Paradigm based on STDs, and introduces the various notions by means of a small example —a producer-consumer variant. The operational semantics is discussed in Section 3, exploiting the running on example. It also illustrates flexibility and clarity of the approach by varying the example. A rather more complicated variant can be found in Section 4, where flexibility and clarity are illustrated with respect to larger numbers of producer and consumer components. Conclusions are formulated in Section 5.

2 Paradigm

In this section we formally introduce Paradigm, and will present the notions on which the behaviour of behaviours and its required consistency with the underlying, more detailed behaviour are based. It seems reasonably straightforward to base the Paradigm notions on transition systems or STDs. Statecharts instead of STDs would have been an alternative, if we could easily prevent their superstates to be overlapping with other superstates. As we actually want to have a special form of overlap, statecharts seem to be less suited for our purpose. So our first definition specifies our notion of an STD and of its behaviours.

Definition 2.1

(a) An STD S is a triple $\langle \mathtt{ST}, \mathtt{LB}, \mathtt{TS} \rangle$. Here $\mathtt{ST}$ is called the set of states, or also the state space; $\mathtt{LB}$ is a set of transition labels or actions, also called the action space; $\mathtt{TS} \subseteq \mathtt{ST} \times \mathtt{LB} \times \mathtt{ST}$ is the set of transitions. We write $x \stackrel{a}{\rightarrow} x'$ in case $(x, a, x') \in \mathtt{TS}$. An STD is also referred to as a process.

(b) Given an STD $S = \langle \mathtt{ST}, \mathtt{LB}, \mathtt{TS} \rangle$ and a state $x_0 \in \mathtt{ST}$, a behaviour of S starting in x_0 is a finite or infinite sequence $x_0, a_0, x_1, a_1, x_2, \ldots$ such that $x_i \stackrel{a_i}{\rightarrow} x_{i+1}$ for all appropriate indices i.

Based on this concept of an STD, Paradigm introduces two notions which constitute its kernel: the notion of a *subprocess* of an STD and the notion of a *trap* of a subprocess. A subprocess is an STD, restricted to a subset of the state and action spaces of the original STD, with the actions having the same behavioural effect. This means, any behaviour of a subprocess is a subsequence of some behaviour of the original STD. This is the reason we informally refer to a subprocess as a model of a certain phase within the set of possible behaviours of the original STD. The trap, being a subset of a subprocess' state space, models a kind of final stage within such a phase: once entered, it cannot be left within that same phase. It is only through the behaviours of a next phase, i.e. in another subprocess, that a trap can be left. As the trap's states thus belong to the state spaces of two subprocesses, we informally refer to such traps as overlaps between two subprocesses or rather overlaps between two subsequent phases.

Definition 2.2 Let an STD $S = \langle \mathtt{ST}, \mathtt{LB}, \mathtt{TS} \rangle$ be given.

(a) A subprocess of S is an STD $\langle \mathtt{st}, \mathtt{lb}, \mathtt{ts} \rangle$ such that $\mathtt{st} \subseteq \mathtt{ST}$, $\mathtt{lb} \subseteq \mathtt{LB}$, and $\mathtt{ts} = \{\, (x, a, x') \in \mathtt{TS} \mid x, x' \in \mathtt{st}, a \in \mathtt{lb} \,\}$.
(b) A trap t of a subprocess $s = \langle \mathtt{st}, \mathtt{lb}, \mathtt{ts} \rangle$ is a nonempty set of states $t \subseteq \mathtt{st}$ such that $x \in t$ and $x \xrightarrow{a} x' \in \mathtt{ts}$ imply that $x' \in t$. If $t = \mathtt{st}$, the trap is called trivial, otherwise the trap is called nontrivial.
(c) If two traps t and t' of the same subprocess have the property $t \subseteq t'$, these traps are called nested; t is called an inner trap of t', and t' is called an outer trap of t.
(d) Let $s = \langle \mathtt{st}, \mathtt{lb}, \mathtt{ts} \rangle$ and $s' = \langle \mathtt{st}', \mathtt{lb}', \mathtt{ts}' \rangle$ be two subprocesses of the same process. A trap t of s is called a trap from s to s' if either $s = s'$, or the states belonging to the trap t are states in s' as well, i.e., $t \subseteq \mathtt{st}'$.

The fourth part of the above definition actually defines the overlap discussed above as a trap from a first subprocess to a second subprocess. Note that such a trap is, in general, not a trap of the second subprocess.

Based on the two central notions of subprocess and trap, i.e. of phase and overlap, a number of structuring notions are additionally defined. First of all, the framework needs to view the original, detailed behaviours of a process in terms of a high-level but still behaviour-like sequence of subprocess s_0, trap t_0, subprocess s_1, trap t_1, subprocess s_2, etc. In addition, the framework is also geared toward specifying required coordination between different processes (non-subprocesses) in terms of these global behavioral views. Some of these structuring notions are part of the above Definition 2.2, the others are formulated in the definitions below. Before we do so, we clarify the kernel notions by means of an example.

The example is the well-known Producer-Consumer problem, here with buffer capacity 2. Below, Figure 2.1 presents a tentative STD modeling the producer, called *TentProd*. State names refer to the effect of the last action by means of which the state was entered. There are two relevant actions: produce and fill —both of one unit— which speak for themselves. The same figure also contains a slightly more unfolded STD, called $ProdCap_2$, modeling the producer such that its behavioural freedom with respect to the buffer capacity of 2 is certainly less than one full behavioural cycle. State names are less relevant here, but the two same action names still are.

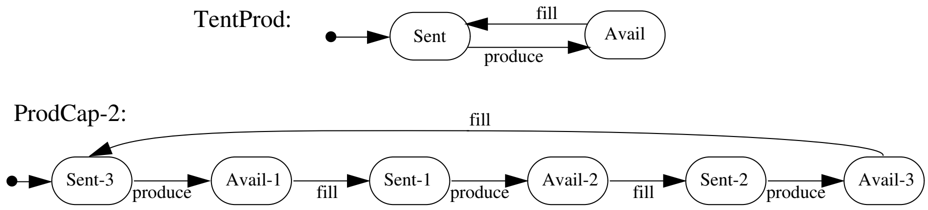

Fig. 2.1. Producer and consumer processes

Figure 2.2 visualizes the relevant subprocesses and traps of $ProdCap_2$. A first phase of the producer is: it may perform three producings and two fillings, because after that the buffer is full, unless the consumer has emptied at least one buffer slot. If indeed so, the phase of the producer slightly shifts from "immediately after its first fill until just before its fourth fill (which is the first because of the behavioural cycle)". Analogously, the producer's third phase shifts again from "immediately after its second fill until just before its fifth (which is its second) fill". The fourth phase then is equal to the first, and so on. In the visualization we use a polygon surrounding states to indicate a nontrivial trap consisting of these states. Circumstances causing such a shift are twofold: the producer has performed one fill, as well as the consumer has performed one empty.

We do not indicate the trivial traps as they are self-evident. As the names of the nontrivial traps suggest, they serve as overlap from subprocess $Phase_1$ to subprocess $Phase_2$, from $Phase_2$ to $Phase_3$ and from $Phase_3$ to $Phase_1$. We could have preferred to indicate an inner trap of the nontrivial traps, being entered by the fill that starts from a state inside the outer nontrivial trap. This would offer a possibility to skip one phase, so to say, but, for the sake of presentation, we leave the example as it is.

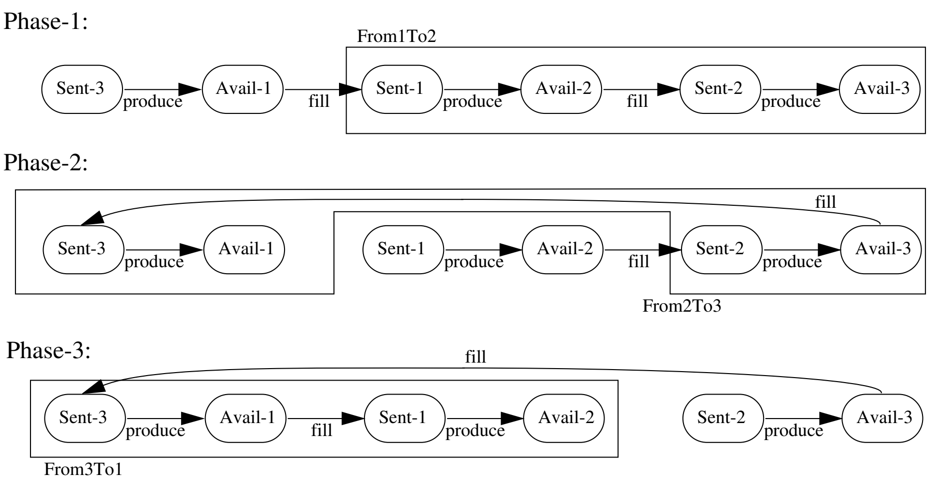

Fig. 2.2. Subprocesses and traps of $ProdCap_2$

It is indeed because of the subprocesses in Figure 2.2 that we prefer the second producer model from Figure 2.1, *ProdCap*$_2$, above the first, *TentProd*: each subprocess would have covered $2\frac{1}{2}$ cycles of *TentProd*, which frustrates a meaningful choice of nontrivial traps. (For general buffer capacity k the unfolding should be a cycle of $k+1$ produce-fill pairs, and the subprocesses are all subsequences containing $k+1$ producings and k fillings: $k+\frac{1}{2}$ cycles of *TentProd*.)

Figure 2.2 shows another very useful property: the three subprocesses together cover the full behaviours of the original STD *ProdCap*$_2$. Moreover, the nontrivial traps are such that they indeed form an overlap from *Phase*$_1$ to *Phase*$_2$, from *Phase*$_2$ to *Phase*$_3$ and from *Phase*$_3$ back to *Phase*$_1$. This is an example of the kind of structure we want to have for an STD which behaviours are to be viewed more globally in terms of subprocesses and traps: the structuring offers the possibility to order the subprocesses according to their overlaps. This kind of structuring is formalized in the next definition.

Definition 2.3

(a) Let $S = \langle \mathtt{ST}, \mathtt{LB}, \mathtt{TS} \rangle$ be an STD. A set $\mathtt{PT} = \{ s_i \mid i \in I \}$ of subprocesses of S, with $s_i = \langle \mathtt{st}_i, \mathtt{lb}_i, \mathtt{ts}_i \rangle$, is called a partition of S, if $\mathtt{ST} = \bigcup \{ \mathtt{st}_i \mid i \in I \}$, $\mathtt{LB} = \bigcup \{ \mathtt{lb}_i \mid i \in I \}$, and $\mathtt{TS} = \bigcup \{ \mathtt{ts}_i \mid i \in I \}$. We use the notation $\mathtt{st}(s_i)$ to denote $\mathtt{st}_i$.

(b) Let S be an STD with a partitioning $\mathtt{PT} = \{ s_i \mid i \in I \}$. Let, for each $i \in I$, $\mathtt{tr}_i$ be a set of traps for the subprocess s_i of S, such that the trivial trap $\mathtt{st}_i \in \mathtt{tr}_i$. Then $\mathtt{TR} = \{ \mathtt{tr}_i \mid i \in I \}$ is called a trap structure for S and $\mathtt{PT}$. We use the notation $\mathtt{tr}(s_i)$ to denote $\mathtt{tr}_i$. For $y \in \mathtt{st}_i$, $trap(y, s_i)$ is the set of all traps of s_i containing y, i.e. $trap(y, s_i) = \{ t \in \mathtt{tr}_i \mid y \in t \}$.

(c) An attainability rule or global transition for an STD S, with respect to a partitioning $\mathtt{PT}$ of S and a trap structure $\mathtt{TR}$ for S and $\mathtt{PT}$, is a triple of the form $s \xrightarrow{t} s'$ with $s, s' \in \mathtt{PT}$, $t \in \mathtt{tr}(s)$, and $t \subseteq \mathtt{st}(s')$, i.e., s and s' are subprocess of S and t is a trap from s to s'. If $s \xrightarrow{t} s'$ we say that s' is directly attainable from s via t.

(d) An employee or employee process $\mathcal{E}$ is an STD $S = \langle \mathtt{ST}, \mathtt{LB}, TS \rangle$ together with a partitioning $\mathtt{PT}$ of S, a trap structure $\mathtt{TR}$ for S and $\mathtt{PT}$ and a set of attainability rules $\mathtt{AT}$ for S with respect to $\mathtt{PT}$ and $\mathtt{TR}$. We will write $\mathcal{E} = \langle \mathtt{ST}, \mathtt{LB}, \mathtt{TS}, \mathtt{PT}, \mathtt{TR}, \mathtt{AT} \rangle$.

Based on Definition 2.3 we can say the following concerning Figure 2.2. The three subprocesses together are a partitioning of *ProdCap*$_2$. The three nontrivial traps together with the three trivial traps are a trap structure. *Phase*$_2$ is directly attainable from *Phase*$_1$; so is *Phase*$_3$ from *Phase*$_2$, and also *Phase*$_1$ from *Phase*$_3$. Finally, *ProdCap*$_2$ is an employee.

Up to here we have said nothing about an STD model for the consumer. But it is the consumer, which by emptying the buffer is supposed to change the freedom of *ProdCap*$_2$, e.g. it should cause *ProdCap*$_2$ to change its subprocess *Phase*$_1$ into the subprocess *Phase*$_2$ after *ProdCap*$_2$ has entered its trap *From1to2*. Figure 2.3 presents two such potential consumer STDs. The first, called *TentCons*, consists

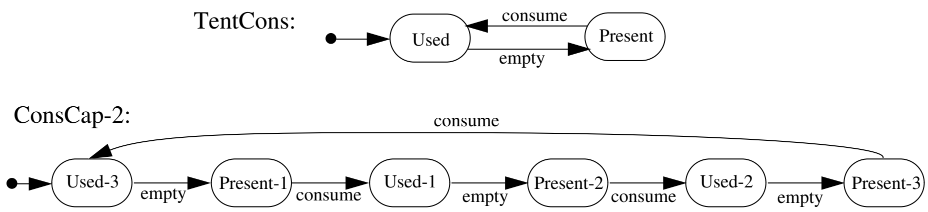

Fig. 2.3. Two STDs as consumer models

of the simple behavioural cycle of emptying and consuming. The second STD, called $ConsCap_2$, is an unfolded version of the first. Similar to the unfolded producer, the second STD suits us better than the first as we will point out.

When introducing the subprocesses of the producer by means of Figure 2.2, we already referred to a certain behavioural dependency between the consumer and the producer. This is the heart of the Producer-Consumer problem: in case of a buffer capacity 2, the producer may only continue to fill the buffer once more —one unit at a time— if the consumer has at least emptied the buffer up to, but not necessarily including the last unit put in it; in addition, the consumer may only continue to empty the buffer once —again, one unit at a time— if the producer has at least filled the buffer with a unit not yet taken from it. So, in view of the required coordination, there is a certain consistency needed between the behaviours of $ProdCap_2$ and $ConsCap_2$. As it turns out, this required consistency can be formulated rather elegantly in terms of the above subprocesses and traps of $ProdCap_2$, relating them to the states and to the transitions respectively, of $ConsCap_2$. Figure 2.4 illustrates this, by mirroring $ConsCap_2$'s structure and referring to $ProdCap_2$'s phases and their overlaps.

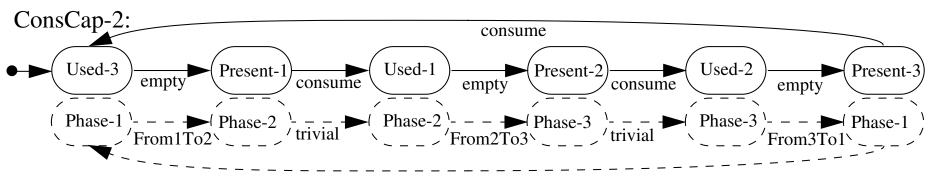

Fig. 2.4. Required consistency between behaviour and behaviour behaviour

This then means that the required behavioural consistency between $ProdCap_2$ and $ConsCap_2$ can be formulated as a required consistency between the detailed behaviours of $ConsCap_2$ and the global behaviours of $ProdCap_2$. In other words, $ConsCap_2$, through its detailed behaviours, models the required behaviour of behaviours of $ProdCap_2$. The formal structure for being able to formulate the required consistency, is expressed by means of the following definition.

Definition 2.4 An STD $M = \langle \mathtt{ST}, \mathtt{LB}, \mathtt{TS} \rangle$, together with a relation γ, is called a manager or manager process with employees $\mathcal{E}_1, \ldots, \mathcal{E}_n$, where $\mathcal{E}_i = \langle \mathtt{ST}_i, \mathtt{LB}_i, \mathtt{TS}_i, \mathtt{PT}_i, \mathtt{TR}_i, \mathtt{AT}_i \rangle$ for $1 \leq i \leq n$, if $\gamma \subseteq \mathtt{TS} \times \mathtt{AT}_1 \times \cdots \times \mathtt{AT}_n$. We will write $\mathcal{M} = (M, \gamma)$ and refer to γ as the consistency relation of $\mathcal{M}$.

Figure 2.4 implicitly suggests the consistency relation for the manager $ConsCap_2$ with respect to employee $ProdCap_2$, viz.

$$\gamma(Used_3 \stackrel{empty}{\rightarrow} Present_1, Phase_1 \stackrel{FromOneToTwo}{\rightarrow} Phase_2),$$
$$\gamma(Present_1 \stackrel{consume}{\rightarrow} Used_1, Phase_2 \stackrel{triv}{\rightarrow} Phase_2), \quad \ldots$$

If one, as a modeler, really thinks that the manager expresses the required consistency, this should be enforced upon the combined system of the manager and its employees. By means of the semantic rules to be presented in the next section, we shall indeed be able to guarantee the consistency as desired.

3 Operational semantics and basic examples

In the previous section we have introduced the notions of a manager and of an employee. Now we will consider systems of a manager process and a single or more employee processes, put together in one system as parallel threads.

Definition 3.1 Let $\mathcal{M}$ be a manager of employees $\mathcal{E}_1, \ldots, \mathcal{E}_n$. The statespace of a Paradigm-model $\langle \mathcal{M}, \mathcal{E}_1, \ldots, \mathcal{E}_n \rangle$ of $\mathcal{M}$ and $\mathcal{E}_1, \ldots, \mathcal{E}_n$ are all $n+1$-tuples of the form $[\![x, (s_i)_{i=1}^n \,;\, (y_i)_{i=1}^n]\!]$ with $x \in \mathtt{st}(\mathcal{M})$, $s_i \in pt(\mathcal{E}_i)$ and $y_i \in \mathtt{st}(s_i)$ for $i \in \{ 1, \ldots, n \}$.

An employee $\mathcal{E}$ consists, according to Definition 2.4, of six components. The underlying STD comprises three of these, viz. states, labels, and transitions. The remaining ones, subprocesses, trap structure and attainability rules, can be grouped together as well. This way an employee represents two STDs, or rather two views on the same STD: a low-level view in terms of states and action-labeled transitions, and, a high-level view in terms of subprocesses and attainability transitions between them that are labeled with traps.

The low-level behaviour is restricted by the subprocesses. The high-level behaviour is triggered by traps. Once a subprocess has reached a trap, it may be transferred to another subprocess by the manager. Therefore, when considering a system of a manager with employees we should distinguish between this two types of dynamics. The low-level operational semantics are the interleavings of the runs of the manager process and the subprocesses of the employees. For the high-level operational semantics one simply restricts to the moves of the manager, i.e. the traces obtained from the low-level model by abstracting away local employees transitions, thus focusing on the steps of the manager. We first give the low-level operational semantics.

Definition 3.2 The low level operational semantics of a Paradigm-model $\mathcal{P} = \langle \mathcal{M}, \mathcal{E}_1, \ldots, \mathcal{E}_n \rangle$, consisting of a manager $\mathcal{M} = (M, \gamma)$ and employees $\mathcal{E}_i = \langle \mathtt{ST}_i, \mathtt{LB}_i, \mathtt{TS}_i, \mathtt{PT}_i, \mathtt{TR}_i, \mathtt{AT}_i \rangle$, is the STD with states as given by Definition 3.1, labels as in $\mathcal{M}$ and $\mathcal{E}_1$ through $\mathcal{E}_n$ and the following transitions:

– *manager transitions* $[\![x, (s_i)_{i=1}^n \,;\, (y_i)_{i=1}^n]\!] \stackrel{a}{\to} [\![x', (s'_i)_{i=1}^n \,;\, (y_i)_{i=1}^n]\!]$ iff $\gamma(x \stackrel{a}{\to} x', s_1 \stackrel{t_1}{\to} s'_1, \ldots, s_n \stackrel{t_n}{\to} s'_n)$ holds for traps $t_i \in \mathit{trap}(y_i, s_i)$ and indices $i \in \{ 1, \ldots, n \}$;
– *employee transitions* $[\![x, (s_i)_{i=1}^n \,;\, (y_i)_{i=1}^n]\!] \stackrel{a}{\to} [\![x, (s_i)_{i=1}^n \,;\, (y'_i)_{i=1}^n]\!]$ iff for some index $j \in \{ 1, \ldots, n \}$, $y_j \stackrel{a}{\to} y'_j$ in s_j and $y'_i = y_i$ for all $i \neq j$, $i \in \{ 1, \ldots, n \}$.

The modeling with Paradigm thus support the factorization of a system in coordination and computational activity. Manager transitions represent coordination steps, employee transitions represent computation steps. As the manager and employees are regarded as autonomous processes, their execution is in parallel, restricted only by the partitioning into subprocesses and the corresponding attainability rules.

In the definitions above, one could also, straightforwardly, have allowed for multiple managers. Since this is at the expense of clarity at this point, we stick in this section to the case of exactly one manager. Hierarchical system are feasible as well, but are beyond the scope of this paper.

Consider again a producer process *Prod* and a consumer process *Cons* sharing a one-place buffer. We plan to have the consumer as the manager and, as discussed in the previous section, find it profitable to enfold the loop of the producer one time. See Figure 3.1. The producer process can be turned into an employee by partitioning it into two subprocesses s_1 and s_2 each with two traps, viz. the trivial trap triv_i (not shown) and an inner trap ok_i, $i = 1, 2$. See Figure 3.2 Furthermore, he producer is endowed with the following attainability rules:

$$s_1 \stackrel{\mathit{triv}_1}{\to} s_1,\ s_1 \stackrel{\mathit{ok}_1}{\to} s_2,\ s_2 \stackrel{\mathit{triv}_2}{\to} s_2,\ s_2 \stackrel{\mathit{ok}_2}{\to} s_1.$$

The consumer process is viewed as a manager of the employee process above by introducing the consistency relation

$$\gamma(c_1 \stackrel{\mathit{empty}}{\to} c_2, s_1 \stackrel{\mathit{ok}_1}{\to} s_2),\ \gamma(c_1 \stackrel{\mathit{empty}}{\to} c_2, s_2 \stackrel{\mathit{ok}_2}{\to} s_1),$$
$$\gamma(c_2 \stackrel{\mathit{cons}}{\to} c_1, s_1 \stackrel{\mathit{triv}_1}{\to} s_1),\ \gamma(c_2 \stackrel{\mathit{cons}}{\to} c_1, s_2 \stackrel{\mathit{triv}_2}{\to} s_2).$$

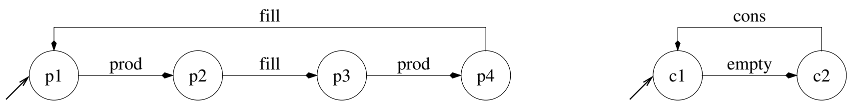

Fig. 3.1. One-place buffer: producer is employee, consumer is manager

The operational system semantics as captured by Definition 3.2 enforces a proper interleaving of the fill and empty actions of the producer and consumer, respectively. A *fill*-action can only be done by the producer by entering an inner trap. An *empty*-action will only be performed, according to the consistency relation, by the consumer after the producer has reached such a trap. Conversely, an

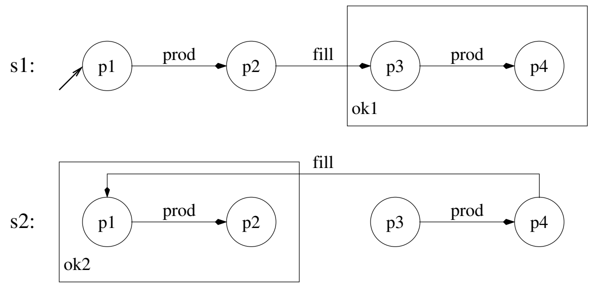

Fig. 3.2. One-place buffer: two producer subprocesses

empty-action from the consumer will result in a transfer of the producer to a subprocess which brings the producer out of the inner trap —thereby blocking an *empty*-action for the consumer— and with admits another *fill*-action.
The abstract description in terms of Paradigm results in a low-level system that complies with the behavioural requirements. In the simple case of this producer-consumer it is feasible to provide the complete STD for the whole system. See Figure 3.3.

$$\begin{array}{ccccccccccccccc}
[1] & & [2] & & [3] & & [4] & & & & & & & & \\
\downarrow \mathit{cons} & & \downarrow \mathit{cons} & & \downarrow \mathit{cons} & & \downarrow \mathit{cons} & & & & & & & & \\
\llbracket c_1, s_1 \,;\, p_1 \rrbracket & \xrightarrow{\mathit{prod}} & \llbracket c_1, s_1 \,;\, p_2 \rrbracket & \xrightarrow{\mathit{fill}} & \llbracket c_1, s_1 \,;\, p_3 \rrbracket & \xrightarrow{\mathit{prod}} & \llbracket c_1, s_1 \,;\, p_4 \rrbracket & & & & & & & & \\
 & & & & \downarrow \mathit{empty} & & \downarrow \mathit{empty} & & & & & & & & \\
 & & & & \llbracket c_2, s_2 \,;\, p_3 \rrbracket & \xrightarrow{\mathit{prod}} & \llbracket c_2, s_2 \,;\, p_4 \rrbracket & \xrightarrow{\mathit{fill}} & \llbracket c_2, s_2 \,;\, p_1 \rrbracket & \xrightarrow{\mathit{prod}} & \llbracket c_2, s_2 \,;\, p_2 \rrbracket & & & & \\
 & & & & \downarrow \mathit{cons} & & \downarrow \mathit{cons} & & \downarrow \mathit{cons} & & \downarrow \mathit{cons} & & & & \\
 & & & & \llbracket c_1, s_2 \,;\, p_3 \rrbracket & \xrightarrow{\mathit{prod}} & \llbracket c_1, s_2 \,;\, p_4 \rrbracket & \xrightarrow{\mathit{fill}} & \llbracket c_1, s_2 \,;\, p_1 \rrbracket & \xrightarrow{\mathit{prod}} & \llbracket c_1, s_2 \,;\, p_2 \rrbracket & & & & \\
 & & & & & & & & \downarrow \mathit{empty} & & \downarrow \mathit{empty} & & & & \\
 & & & & & & & & \llbracket c_2, s_1 \,;\, p_1 \rrbracket & \xrightarrow{\mathit{prod}} & \llbracket c_2, s_1 \,;\, p_2 \rrbracket & \xrightarrow{\mathit{fill}} & \llbracket c_2, s_1 \,;\, p_3 \rrbracket & \xrightarrow{\mathit{prod}} & \llbracket c_2, s_1 \,;\, p_4 \rrbracket \\
 & & & & & & & & \downarrow \mathit{cons} & & \downarrow \mathit{cons} & & \downarrow \mathit{cons} & & \downarrow \mathit{cons} \\
 & & & & & & & & [1] & & [2] & & [3] & & [4]
\end{array}$$

Fig. 3.3. low-level STD of the producer-consumer problem with consumer as manager

The low-level Producer-Consumer systems hints at a more abstract, manager-oriented operational semantics; the STD in Figure 3.3 is much too concrete. Instead we might want to focus on the steps of the manager, i.e. on the vertical transitions.

Definition 3.3 The high-level operational semantics of a Paradigm-model $\mathcal{P} = \langle \mathcal{M}, \mathcal{E}_1, \ldots, \mathcal{E}_n \rangle$, consisting of a manager $\mathcal{M} = (M, \gamma)$ and employees $\mathcal{E}_i = \langle \mathtt{ST}_i, \mathtt{LB}_i, \mathtt{TS}_i, \mathtt{PT}_i, \mathtt{TR}_i, \mathtt{AT}_i \rangle$, is an STD with states as given by Definition 3.1, labels from $\mathcal{M}$ together with a fresh label *skip*, and the following transitions:

– *manager transitions* $[\![x, (s_i)_{i=1}^n \,;\, (y_i)_{i=1}^n]\!] \xrightarrow{a} [\![x', (s'_i)_{i=1}^n \,;\, (y'_i)_{i=1}^n]\!]$ iff $\gamma(x \xrightarrow{a} x', s_1 \xrightarrow{t_1} s'_1, \ldots, s_n \xrightarrow{t_n} s'_n)$ holds for traps $t_i \in \mathit{trap}(y_i, s_i)$ and $y_i \to^* y'_i$ in s_i for all $i \in \{ 1, \ldots, n \}$;
– *employee transitions* $[\![x, (s_i)_{i=1}^n \,;\, (y_i)_{i=1}^n]\!] \xrightarrow{\mathit{skip}} [\![x, (s_i)_{i=1}^n \,;\, (y'_i)_{i=1}^n]\!]$ iff $y_i \to^* y'_i$ in s_i for all $i \in \{ 1, \ldots, n \}$.

A graphical representation of the high-level semantics does not differ that much when compared to its low-level model. In fact, assuming an appropriate transitive closure, only the employee transitions will be relabeled into *skip*. However, the employee steps are now considered internal, the system can now be analyzed e.g. using techniques for weak bisimulation [13, 4]. This way systems that may differ in their respective component, in particular in the employee behaviour, can be compared at the system level for equivalence.

Especially in the context of hierarchical Paradigm-models, in which managers on the one level can act as employees at a higher-level, such a notion of system equivalence will prove useful. Safety or liveness properties that are proved with respect to a certain layer, can be starting point for the analysis at a higher layer. E.g., the fact that the number of retrievals from a particular buffer does not exceed the number of deliveries, can be used as assumption in an analysis of a chain in which the producer-consumer system is only one single component.

From the point of view of the high-level semantics, our requirement on traps of Definition 2.3c can be explained once more. It will not matter whether the progress $y_i \to^* y'_i$ is made with respect to the subprocess s_i or with respect to the subprocess s'_i, as long as $t_i \subseteq \mathtt{st}(s'_i)$.

The high-level modeling of the producer-consumer example with Paradigm is more clear than the 'brute force' approach of the low-level transition system. At the abstract level it is also easy to switch the role of manager and employee. If we prefer in our example the producer as the manager and the consumer as its employee, we can model this as follows:

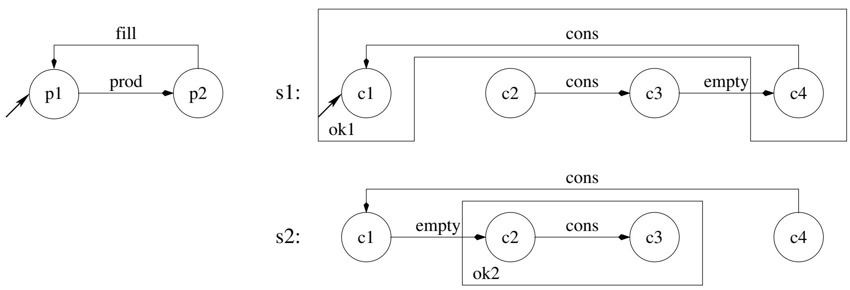

Fig. 3.4. One-place buffer: producer is manager, consumer is employee

Similar to the previous situation the employee is splitted into two subprocesses, each with the trivial and an inner trap. The inner trap signals that the consumer is ready for an empty action. The producer will allow this as soon it has put a new item in the buffer. This is then achieved by the transfer of the consumer from the one subprocess to the other. The corresponding consistency relation is:

$$\gamma(p_1 \stackrel{prod}{\to} p_2, s_i \stackrel{triv}{\to} s_i), \quad \gamma(p_2 \stackrel{fill}{\to} p_1, s_1 \stackrel{ok_1}{\to} s_2), \quad \gamma(p_2 \stackrel{fill}{\to} p_1, s_2 \stackrel{ok_2}{\to} s_1)$$

We next look at a variation of the above setting by considering the producer-consumer problem with a buffer of *infinite* capacity. Clearly, as we need to distinguish the infinitely many loadings of the buffer, it will be handy to start off with the producer and consumer processes as infinite linear processes. The producer will be the employee; the consumer will be the manager. See Figure 3.5. As consistency relation we now have

$$\begin{array}{lll} \gamma(c_1 \stackrel{empty}{\to} c_2, s \stackrel{t_1}{\to} s) & \gamma(c_3 \stackrel{empty}{\to} c_4, s \stackrel{t_2}{\to} s) & \gamma(c_5 \stackrel{empty}{\to} c_6, s \stackrel{t_3}{\to} s) \\ \gamma(c_2 \stackrel{cons}{\to} c_3, s \stackrel{triv}{\to} s) & \gamma(c_4 \stackrel{cons}{\to} c_5, s \stackrel{triv}{\to} s) & \ldots \end{array}$$

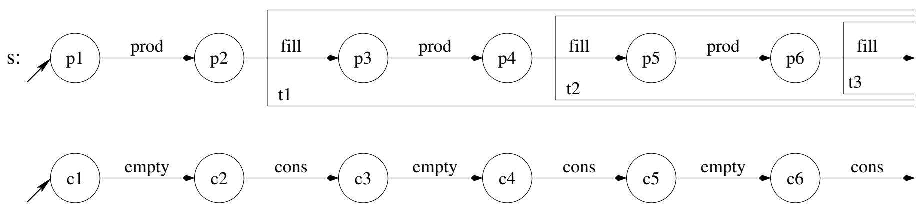

Fig. 3.5. Infinite buffer: producer is employee, consumer is manager

Note that there is a single subprocess for the producer that has an infinite nesting of traps. Intuitively this is obvious: there are no restrictions on the producer, and when the producer fills the buffer with another item, the consumer should be in a position to get this. The consumer can not retrieve more items as there are in the buffer, since the *empty*-transition is coupled with a *fill*-transition of the producer, via the consistency relation.

Dual to the situation above we can have the producer as the manager for the infinite buffer case. See Figure 3.6. As consistency relation for the producer is the following:

$$\begin{array}{lll} \gamma(p_1 \stackrel{prod}{\to} p_2, s_i \stackrel{triv}{\to} s_i) & \gamma(p_3 \stackrel{prod}{\to} p_4, s_i \stackrel{triv}{\to} s_i) & \gamma(p_5 \stackrel{prod}{\to} p_6, s_i \stackrel{triv}{\to} s_i) \\ \gamma(p_2 \stackrel{fill}{\to} p_3, s_1 \stackrel{triv}{\to} s_2) & \gamma(p_4 \stackrel{fill}{\to} p_5, s_2 \stackrel{triv}{\to} s_3) & \ldots \end{array}$$

Now we partition the consumer into infinitely many subprocesses: each next subprocess is allowed to retrieve, and also to consume, one item more from the buffer. The producer transfers the consumer to this next process in parallel with its own fill action. Note that the consumer is not enforced to handle any item from the buffer. However, via the inclusion of the states of the trivial trap of the one process, the local state is preserved, i.e., it is 'memorized' how many items there have been taken from the buffer already.

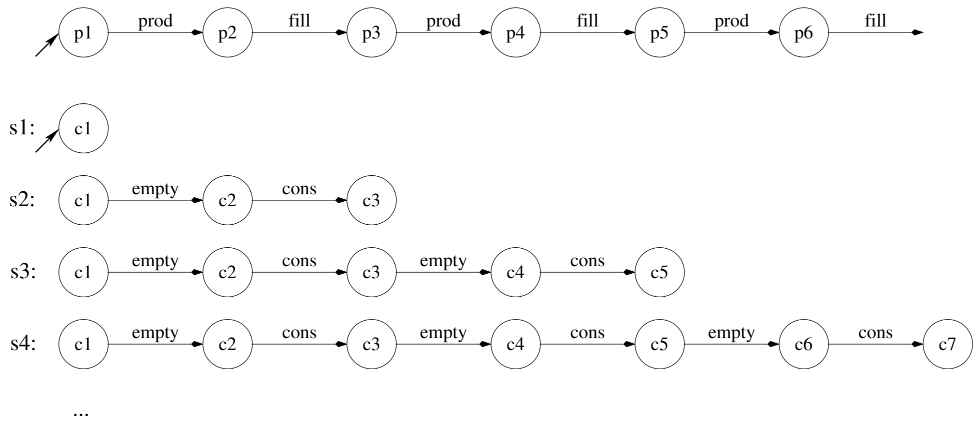

Fig. 3.6. Infinite buffer: producer is manager, consumer is employee

4 A More Complicated Example

The examples presented so far are relatively easy. However, really complicated examples can be handled in a similar fashion. As it turns out, once a manager has been modeled, together with its employees having suitable subprocesses, traps and attainability rules, a solution implementing this specification is easy to formulate. So the harder part of the problem is, finding a suitable Paradigm model. In our experience, also this is not very difficult, as the notions invite an sufficiently experienced modeler to remain on a level of adequate abstraction. This is, in fact, the main advantage of Paradigm. There seems to be some analogue with the way human managers in organizations actually manage, by checking and controlling not so much each step, but more global phases. Paradigm provides the notions to get the syntax of the solution right, thereby clarifying the consequences of the coordination requirements as specified. The structural operational semantics then guarantee the implementation to be in line with the required coordination.

To give an impression thereof, we present a variant of the producer-consumer problem, where the buffer capacity is 2, and where two producers and three consumers may try to fill and empty one of the two buffer slots. A producer may only do so if the slot is empty, and the consumer may only do so if the slot is full. Moreover, if two slots happen to be full, a consumer asking for emptying a slot, gets the one filled first.
Figure 4.1 visualizes the STD Producer for the two producers and the STD Consumer for the three consumers. In this case we have enlarged the the cycle of the producers and the consumers by the extra step —action, transition— called ask, meaning the asking for using the buffer once. Having this permission is the critical phase of a full behavioural cycle, modeled as subprocess *PCrit* for the producer and as *CCrit* for the consumer. In the critical phase a producer or consumer may fill or empty one buffer slot, but it may not ask for another permission (as it is having it). The safe phase, modeled as *PSafe* for the producer and as

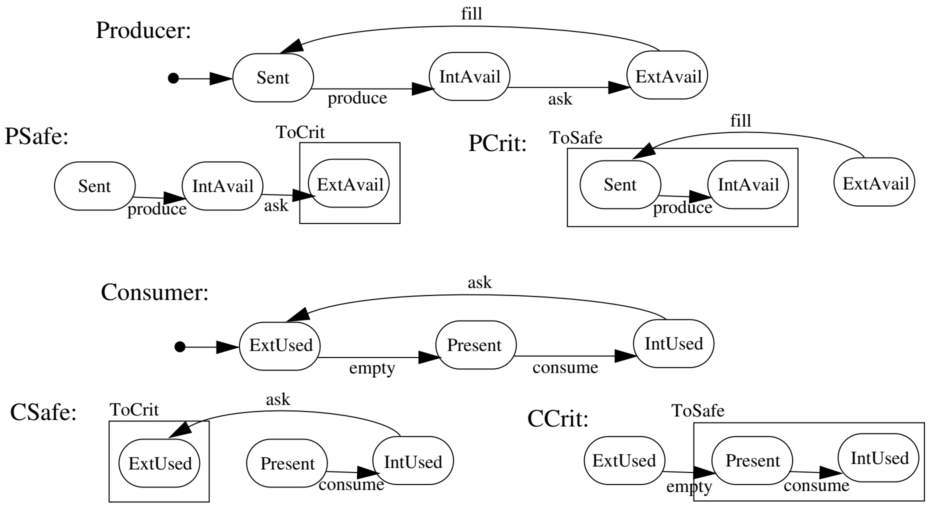

Fig. 4.1. A producer and a consumer, both as employees

CSafe for the consumer, is where filling or emptying are not allowed. Note that producing or consuming are allowed in every subprocess. Asking a permission corresponds to a small trap, in both cases called *ToCrit*: the communication is as synchronous as possible, as the permission should be given only if really needed. The other trap, in both cases called *ToSafe*, is substantially larger, to allow continuing to produce or to consume even if the permission has not yet been withdrawn.

A separate manager, called $Buffer_2$ and represented in Figure 5.2, suggests a rather clear solution. Each state name starts with a number indicating how many buffer slots are full; after the colon it either indicates that all producers and consumers are supposed to be in their safe phase —denoted as *AllSafe*— or it indicates how many producers and/or consumers are in their critical phase - denoted as 1*PCrit*, 2*PCrit*, 1*CCrit*, 2*CCrit* or 1*C*&1*PCrit*. Each action name starts with globally mentioning either one (arbitrary or particular) producer or one (arbitrary or particular) consumer; after the colon it mentions the trap this producer or consumer should have reached in order to make this action appropriate.

Manager $Buffer_2$ is indeed high-level in the Paradigm sense, as it specifies how to control subprocess combinations of its five employees, based on their traps reached. In addition, $Buffer_2$ is even more high level, as it does not specify any order of checking producers and consumers for having entered a nontrivial trap, as it does not discriminate between specific producers or consumers and as it does not indicate which slot has the oldest content. A fully fledged $Buffer_2$ should indeed do so, but our $Buffer_2$ concentrates on the essential part of the coordination, which is hardest. Moreover, based on this requirement specification also the requirements for buffers with a different capacity and/or different numbers of producers or consumers are rather straightforward to specify; and the producers and consumers do not change at all, which shows how the globalization of the behaviour in terms of subprocesses (as states) and traps (as transitions) supports designing sufficiently flexible components.

For brevity's sake, we omit a full description of the consistency relation, as the reader can easily guess its values from the intuitive state and action names.

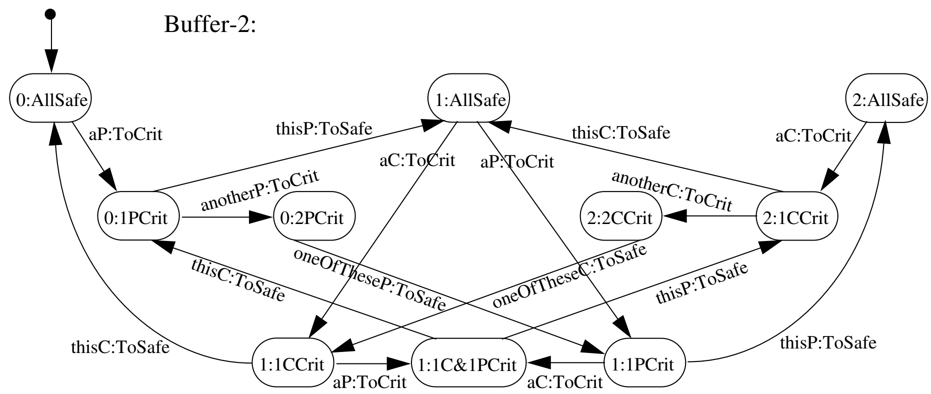

Fig. 4.2. A multi-producer multi-consumer Paradigm-model of a buffer of capacity 2

5 Concluding remarks

This paper discusses the high-level modeling language Paradigm and proposes an operational semantics for it. Paradigm provides a simple but powerful mechanism for the coordination of processes based on the notion of subprocess, trap and attainability rules.

Using Paradigm one can abstract away from the low-level details of processes by focusing on relevant phases, called subprocesses and on relevant overlaps between phases, called traps. Processes are divided into the manager and employees. An employee is partitioned into subprocesses, prescribing the allowed behaviour. A manager can be triggered to transfer one of its employees from one subprocess to another based on global information expressing which overlap, i.e. trap, has been reached. This is captured by the so-called attainability rules. A transition system-style low-level semantics for Paradigm systems is proposed, that couples the subprocess behaviour and steps of the manager based on the attainability rules. An abstraction of this model focusses on the latter steps, yielding a high-level semantics.

Several experiments in which we implemented our semantics using Prolog have been performed. These experiments provided insight in the trace semantics of the Paradigm-models at hand. Further work will concentrate on extending this and will seek for proof principles for the equivalence of Paradigm systems. It is likely that techniques employed for weak bisimulation will show profitable here. We also plan to address the interplay of the semantics and the accompanying equivalences for hierarchical systems.

References

1. F. Arbab. *Manifold version 2: Language reference manual.* CWI, Amsterdam, 1996.
2. M.H. ter Beek, C.A. Ellis, J. Kleijn, and G. Rozenberg. Team automata for spatial access control. In *ECSCW 2001, European Conference on Computer Supported Cooperative Work.* Kluwer, 2001.
3. R. Ben-Natan. *CORBA, a Guide to Common Object Request Broker Architecture.* McGraw-Hill, 1995.
4. J.A. Bergstra, A. Ponse, and S.A. Smolka. *Handbook of Process Algebra.* Elsevier Science, 2001.
5. M.M. Bonsangue, F. Arbab, J.W. de Bakker, J.J.M.M. Rutten, A. Scutella, and G. Zavattaro. A transition system semantics for the control-driven coordination language MANIFOLD. *Theoretical Computer Science*, 240:3–47, 2000.
6. S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31:560–599, 1984.
7. A. de Bruin and S. van der Made. Introduction to Paradigm. Technical Report ERS-2002-09-LIS, Erasmus University Rotterdam, 2002.
8. G. Engels and L.P.J. Groenewegen. Socca: Specifications of coordinated and cooperative activities. In A. Finkelstein, J. Kramer, and B.A. Nuseibeh, editors, *Software Process Modelling and Technology*, pages 71–102. Research Studies Press, Taunton, 1994.
9. G. Engels and L.P.J. Groenewegen. Object-oriented modeling: A roadmap. In A. Finkelstein, editor, *The Future of Software Engineering, 22nd International Conferenece on Software Engineering*, pages 103–116. ACM Press, 2000.
10. G. Engels, L.P.J. Groenewegen, and G. Kappel. Coordinated collaboration of objects. In M.P. Papazoglou, S. Spaccapietra, and Z. Tari, editors, *Advances in Object-Oriented Data Modeling*, pages 307–331. The MIT Press, 2000.
11. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
12. P.J. 't Hoen. *Towards Distributed Development of Large Object-Oriented Models.* PhD thesis, LIACS, Leiden University, 2001.
13. R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.
14. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, 1996.
15. M.R. van Steen, L.P.J. Groenewegen, and G. Oosting. Parallel control processes: Modular parallelism and communication. In L.O. Hertzberger, editor, *Intelligent Autonomous Systems*, pages 562–579. North-Holland, 1987.
16. R. Suettenbach. *Formalisierung visueller Modellierungssprachen Objektorientierter Methoden.* PhD thesis, Dept. of Comp. Sc., Universität Koblenz-Landau, 2001.