Dynamic Consistency in Process Algebra: From Paradigm to ACP

S. Andova[a,*], L.P.J. Groenewegen[b], E.P. de Vink[a]

[a] *Formal Methods Group, Department of Mathematics and Computer Science*
Eindhoven University of Technology, The Netherlands

[b] *FaST Group, Leiden Institute of Advanced Computer Science*
Leiden University, The Netherlands

Abstract

The coordination modelling language Paradigm addresses collaboration between components in terms of dynamic constraints. Within a Paradigm model, component dynamics are consistently specified at various levels of abstraction. The operational semantics of Paradigm is given. For a large, general subclass of Paradigm models a translation into process algebra is provided. Examples of a scheduling problem illustrate the approach. Once expressed in process algebra, relying on a correctness result, Paradigm models are amenable to process algebraic reasoning. Verification using the mCRL2 toolset is addressed as well.

Key words: branching bisimulation, collaboration, dynamic consistency, dynamic constraint, Paradigm, process algebra

1. Introduction

Process algebras are becoming an important stepping stone from software architecture and description formalisms to automated analysis and verification tools. E.g., see [1, 32, 30, 29, 27]. In this paper, we link the coordination modeling language Paradigm via the process algebra ACP with the `mCRL2` toolset. In this way, the flexibility of coordination regarding a software system as a loosely coupled, but structured aggregation of components, is connected to the computational rigor of process equivalence and model-checking. A systematic translation of Paradigm collaborations as recursive specifications of systems of parallel processes is presented. Central to Paradigm is the decomposition of dynamic constraints along two axes: (i) vertically, restrictions on a component with respect to the roles it fulfills in all collaborations it is engaged in, (ii) horizontally, coordination and synchronization of subbehavior enforced upon participants in a collaboration.

*Corresponding author, email `s.andova@tue.nl`.

The coordination modeling language Paradigm [22, 23] specifies roles and interactions within collaborations between components. Roles and interactions are defined in terms of temporary constraints on the dynamics of components. The constraints can be of two kinds, either purely sequential per component (vertical) or step-wise synchronizing for an ensemble of components (horizontal). A sequential constraint corresponds to a role of a component within a protocol, a behavioral view on the component's underlying, typically hidden dynamics. A synchronizing constraint corresponds to the distributed execution of the protocol by the components, a specific parallelization of their roles, constituting a dynamically consistent coordinated computation.

Specific for Paradigm is, via suitable constraint composition it allows for modeling evolution and self-adaptation [23, 2, 24, 3]. Translation of Paradigm into ACP as presented here, is a first step towards the long term research goal to provide formal underpinnings for originally unforeseen changes of systems. To give this effort a sound starting point, first of all thorough presentations of Paradigm and of its semantics are given.

Processes algebras (PAs for short), such as CCS, CSP, LOTOS and ACP [9], provide a powerful framework for formal modeling and reasoning about concurrent systems. Keystone is the notion of compositionality. Each component of the system is modeled separately, and the complete system is obtained as a parallel composition of its interacting components. In addition, process algebras have mechanisms to accommodate the synchronization and hiding of actions (by renaming them into the so-called silent action τ). To compare and relate processes in terms of their behaviors, various equivalences can be built upon process calculi. This paper exploits branching bisimulation to that aim.

The translation of Paradigm models into PA is first introduced through some example variants. Basically, the example system consists of n clients who try to get service from one server exclusively, a critical section problem. The server chooses the next client in a non-deterministic manner. In later variants this is done in a round-robin manner, where the server is not necessarily involved in the coordination. A translation into PA is given and subsequent analysis with the `mCRL2` toolset is discussed.

While the translation of the Paradigm models into PA for the examples is done manually, the state spaces of both entire systems (including all interleaving and interaction) grow exponentially and their analysis exceed human capabilities as the number of clients increases. The toolset `mCRL2` enables us to generate complete state spaces, on which further analysis can be done. For the examples, correctness of several functional properties is being checked. For instance, as expected, the non-deterministic server does not guarantee eventual access to the service, unless fairness is assumed. In contrast, the round-robin manner of serving guarantees, as is no surprise either, access within one cycle. However, the main point is, that once translation into ACP has been achieved, formal methods for analysis of a Paradigm collaboration are within reach. Thus, the embedding of the collaboration into process algebra brings model-checking and analysis, together with its rigor, at the abstraction level of the coordination. Our translation into PA preserves the general structure and dynamics of the original

Paradigm model, a property we do not expect to hold when e.g. translating into Petri-nets.

The paper's structure is as follows. In Section 2 Paradigm is presented in detail, illustrated by different example solutions for the same problem situation. Section 3 then formulates operational semantics for Paradigm and it compares Paradigm with other such approaches. Section 4 briefly introduces process algebra and gives some pilot translations and verifications. In Section 5 the general translation of a large and important subclass of Paradigm models into ACP is given. Section 6 concludes the paper.

2. Introduction to Paradigm

This section introduces the coordination modeling language Paradigm. In particular, it defines and discusses the basic notions in terms of which Paradigm models are formulated. By means of a coordination model for a concrete collaboration among components, basic notions and their usage are explained: notions for a single component in Subsection 2.1; notions for integrating what the components contribute to a collaboration in Subsection 2.2. To illustrate Paradigm's modeling flexibility, some variants of the example are presented in Subsection 2.3.

2.1. Paradigm, the notions for a single component

The name Paradigm is an abbreviation of PARallelism, its Analysis, Design and Implementation by a General Method. Paradigm's subject of interest is parallelism as arising in collaborations, a constellation of dynamic components having to achieve a common goal. The dynamicity of the components consists of their own behaviour and their mutual behavioral influencing. Depending on domain and fashion, components are also referred to as agents, units, objects or entities. Within a collaboration, components together behave simultaneously but not necessarily synchronized. Considered in separation, each component behaves in a piece-wise strictly sequential manner, thread-like. A component's behaviour then might consist of one sequence of steps, one thread, or it might consist of more such threads in parallel. Within the collaboration, all such threads, either from different components or from the same one, have to be woven, well-integrated, into the larger dynamic outcome of the collaboration. In view thereof, Paradigm provides special notions as well as special relations between them, facilitating the modeling of the various kinds of dynamics relevant for such collaborations. Example collaborations for which Paradigm models exist, are from operating systems, simulation, software processes, security, self-adaptation, see e.g. [36, 16, 17, 21, 4, 3].

Similar to modeling languages like UML, Paradigm has a strongly visual representation, intuitively as well as effectively expressing what any particular model specifies. In addition, the visual representations are underpinned by precise mathematical constructs, constituting the formal definitions of the Paradigm notions and their dependencies.

To model coordination solutions for collaborating components, Paradigm has five basic notions: STD, phase, (connecting) trap, role and consistency rule. We shall discuss them with the help of the example of a critical section problem and a particular solution for it. In this subsection we shall restrict attention to the first four of Paradigm's basic notions only, as they are the structuring notions of Paradigm, each related to a component on its own, although involved in a collaboration. In Subsection 2.2 the consistency rules will be addressed, integrating component involvement into collaborative effort. The rigorous treatment of the notions paves the way towards operational semantics for Paradigm as given in Section 3.1.

The example collaboration used for explaining the notions, is the following. In and around a shop, n different potential Client_i, with $i = 1, \ldots, n$, are active. By entering the shop, Client_i starts competing for a service turn, to be provided by the one Server present in the shop and exclusively to only one Client_i per turn. By leaving the shop, Client_i finishes the service turn. We shall refer to this collaboration as CS, for Critical Section, as a Client_i's activities belonging to a service turn can be viewed as her critical section.

A Paradigm model, hence each Paradigm model for the CS collaboration too, is built from STDs, component descriptions in terms of sequential, possibly non-deterministic, behaviours.

- A *state-transition diagram* or *STD* Z is a triple $Z = \langle \mathtt{ST}, \mathtt{AC}, \mathtt{TR} \rangle$ with $\mathtt{ST}$ the set of states of Z, $\mathtt{AC}$ the set of actions of Z and $\mathtt{TR} \subseteq \mathtt{ST} \times \mathtt{AC} \times \mathtt{ST}$ the set of transitions of Z. A transition $(x, a, x') \in \mathtt{TR}$ is denoted as $x \xrightarrow{a} x'$.

An STD is a step-wise description, from state to state, of *possible* behaviors a component can have, like a simple, purely sequential state machine in UML, with the above mentioned thread-like behaviour. An instance of an STD Z then has one realization or concrete behaviour

$$x_0 \xrightarrow{a_0} x_1 \xrightarrow{a_1} \ldots \xrightarrow{a_{n-1}} x_n \xrightarrow{a_n} \ldots$$

being a sequence of steps $x_i \xrightarrow{a_i} x_{i+1}$ of Z, where x_0 is the starting state. In this paper all behaviours are infinite, so no final states occur.

An STD is visualized as a directed graph, where nodes are states and where action-labeled directed edges are transitions, pointing to the next state. In addition, if all instances of the STD have the same initial state, this is graphically indicated by a black-dot-and-arrow pointing from the dot to the initial state.

Figure 1 gives the STD for any Client_i, $i = 1, \ldots, n$, in and around a shop. As the figure suggestively expresses, any Client_i starts in state Out where she is supposed to be outside the shop. By taking action enter, she reaches state Waiting: being in the shop while waiting for a service turn. By taking action explain, she reaches state Busy where she spends her service turn, until she takes thank. She then reaches AtDoor from where taking leave let her return to Out. Thus, each Client_i clearly has purely sequential behavior: infinitely often

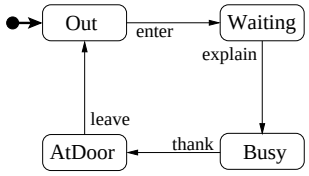

Figure 1: STD of Client_i.

repeating the cycle of these four steps. Per cycle her critical section consists of an explain-step followed by sojourning in Busy and closed by a thank-step.

In view of coordinating whatever collaboration between STDs, Paradigm specifies behavioral influencing between STDs by means of temporary behavioral constraints of two kinds. The first kind of constraint is, a behavioral constraint *imposed on* an STD from "elsewhere", called a phase of the STD. As we shall see later, the semantical effect of the phase as constraint is: it restricts the choice there is of taking steps within the STD.

- A *phase* S of STD $Z = \langle \mathtt{ST}, \mathtt{AC}, \mathtt{TR} \rangle$ is an STD $S = \langle \mathtt{st}, \mathtt{ac}, \mathtt{tr} \rangle$ such that $\mathtt{st} \subseteq \mathtt{ST}$, $\mathtt{ac} \subseteq \mathtt{AC}$ and $\mathtt{tr} \subseteq \mathtt{TR}$.

Please note, phase S being an STD $S = \langle \mathtt{st}, \mathtt{ac}, \mathtt{tr} \rangle$ implies, any step $\vartheta \in \mathtt{tr}$ of S is a triple $\vartheta = (x, a, x') \in \mathtt{st} \times \mathtt{ac} \times \mathtt{st} \subseteq \mathtt{ST} \times \mathtt{AC} \times \mathtt{ST}$. So, a phase of an STD is itself a subSTD of the larger STD. As such, a phase S of an STD Z is a STD-wise description of subbehaviors of the behaviors of Z. Precisely these subbehaviors exhaustively specify, in which part of its state space the larger STD Z has to stay and which steps STD Z is allowed to take there. Thus the dynamics of Z is restricted, but only as long as phase S of Z remains constraint in force. Any decision which phase of Z is *constraint in force*, comes from "elsewhere", therefore we say, a phase of an STD is a temporarily valid constraint to be imposed on the STD it is a phase of.

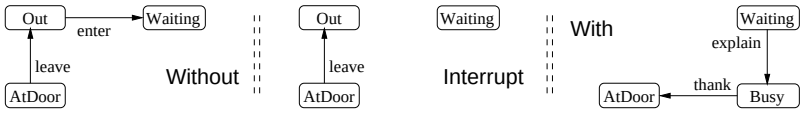

Figure 2: Phases of each Client_i.

Visualized, a phase of STD Z is a fragment of Z preserving the form of Z in the fragment. See Figure 2 for three phases of each Client_i, viz. Without, Interrupt, With. The three phases have been purposely chosen in view of the coordination solutions we are after for the CS collaboration. Phase Without prohibits Client_i to be in state Busy as well as to take the explain-step and the thank-step, thus keeping Client_i out of her critical section. Contrarily, phase With allows going to state Busy, staying there and leaving it, all this only once. Finally, the intermediate phase Interrupt is an interrupted form of Without, as action enter cannot be taken, but being in state Waiting is allowed, though.

So, phase Without specifies the behavioral freedom any Client_i has, when not having the permission to enter her critical section. Phase With specifies the behavioral freedom any Client_i has, when having the permission. Phase Interrupt then specifies a behavioral freedom of Client_i even further restricted than by phase Without, useful when not having the permission yet but being under consideration for getting it: Client_i is not allowed to start waiting for a permission when already under consideration. Please note, defining phases of a given STD is a matter of modeling choice, to be guided by the collaborative problem and the way one wants to solve it, not unlike programming a solution for a certain problem.

In view of enabling whatever "elsewhere" to impose phases as constraints on an STD, the same "elsewhwere" is to be informed about certain progress within a phase. To this aim Paradigm has the notion of a trap of a phase of an STD. A trap of a phase of a larger STD constitutes Paradigm's second kind of behavioral constraint, *committed to* by the larger STD towards "elsewhere".

- A *trap* t of phase $S = \langle \mathtt{st}, \mathtt{ac}, \mathtt{tr} \rangle$ of STD Z is a non-empty set of states $t \subseteq \mathtt{st}$ such that $x \in t$ and $x \xrightarrow{a} x' \in \mathtt{tr}$ imply $x' \in t$. If $t = \mathtt{st}$, trap t is called *trivial*, denoted as $\mathsf{triv}(S)$. A trap t of phase S of STD Z *connects* phase S to a phase $S' = \langle \mathtt{st}', \mathtt{ac}', \mathtt{tr}' \rangle$ of Z if $t \subseteq \mathtt{st}'$. Such trap-based connectivity between two phases of Z is called a *phase transfer* and is denoted by $S \xrightarrow{t} S'$.

A trap of a phase of an STD is a subset of the states of the phase, such that once entered, the subset cannot be left according to the dynamics of the phase. So, within a phase, the entering of a trap is irrevocable, thus marking the beginning of a final stage of the phase. Within the phase of an STD, the entering of a trap of it can serve as commit from the STD not to leave the trap. Such a commit can be used as guard for imposing a new phase. In that case the states of the trap are shared between the phase it is a trap of and the newly imposed phase (of the same larger STD): from *whatever* state the larger STD is in, it can continue in a sufficiently smooth manner according to the new constraint imposed.

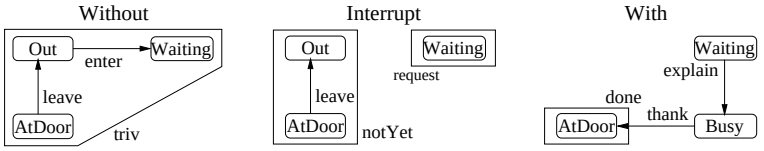

Figure 3: Phases and their traps, of each Client_i.

Visualizing traps is done by drawing a polygon around the states of a phase belonging to the trap. Figure 3 adorns the phases from Figure 2 with some traps. Like the phases, also the traps have been chosen on purpose, in view of the coordination we want to establish for the CS collaboration. In the figure, phase Without has trap triv, the trivial trap of it. It expresses trivial progress within the phase towards a next phase to be imposed. In other words, while being

within phase Without and hence being prohibited to enter the critical section, every progress is good enough for Client_i, even no progress, as the progress is not so much towards being fit for the critical section, but only towards being checked for being fit. Phase Interrupt has two traps notYet and request, indicating different kinds of progress towards being fit for the critical section: trap notYet for not enough progress yet and trap request for being fit indeed. Phase With has trap done; as during this phase Client_i is permitted to enter her critical section, progress is towards giving-up the permission; thus trap done indicates being fit for granting the permission to another $\mathsf{Client}_j, j \neq i$, so withdrawing it from Client_i.

To enable smooth consecutive imposing of phases on a Client_i, the following connectivity of traps is needed. Trap triv is connecting from Without to Interrupt, so phase transfer $\mathsf{Without} \xrightarrow{triv} \mathsf{Interrupt}$ is well-defined. It actually means, once Without is constraint in force, the phase transfer from Without to Interrupt can occur unconditionally, at any moment, as trap triv means that every progress within Without is good for being interrupted. Similarly, two phase transfers, $\mathsf{Interrupt} \xrightarrow{notYet} \mathsf{Without}$ and $\mathsf{Interrupt} \xrightarrow{request} \mathsf{With}$, are well-defined on the basis of two different connecting traps of Interrupt: towards two different next phases for smooth continuation, exactly according to the actual progress made. Also, phase transfer $\mathsf{With} \xrightarrow{done} \mathsf{Without}$ is well-defined and it only occurs after necessary progress has been made.

As the above informal explanation suggests for any separate Client_i in the CS collaboration, suitably modeled phases and their connecting traps of one underlying larger STD have their own dynamicity in terms of constraints in force: subsequently imposed phases, one at a time, alternated with connecting traps committed to and moreover used as guard for a phase transfer. The Paradigm notion of role specifies such dynamicity, based on a selection of phases of the same STD combined with a selection of traps per phase, everything assembled into a so-called partition.

- A *partition* $\pi = \{ (S_i, T_i) \mid i \in I \}$ of an STD $Z = \langle \mathtt{ST}, \mathtt{AC}, \mathtt{TR} \rangle$, with a nonempty index set I, is a set of pairs (S_i, T_i) of a phase $S_i = \langle \mathtt{st}_i, \mathtt{ac}_i, \mathtt{tr}_i \rangle$ of Z and of a set T_i of traps of S_i with $\mathsf{triv}(S_i) \in T_i$.
- The *role* at the level of a partition $\pi = \{ (S_i, T_i) \mid i \in I \}$ of an STD Z is an STD $Z(\pi) = \langle \widehat{\mathtt{ST}}, \widehat{\mathtt{AC}}, \widehat{\mathtt{TS}} \rangle$ with $\widehat{\mathtt{ST}} \subseteq \{ S_i \mid i \in I \}$, $\widehat{\mathtt{AC}} \subseteq \bigcup_{i \in I} T_i$ and $\widehat{\mathtt{TS}} \subseteq \{ S_i \xrightarrow{t} S_j \mid i, j \in I, t \in \widehat{\mathtt{AC}} \}$ a set of phase transfers. Z is called the *detailed* STD underlying *global* STD $Z(\pi)$, the π-role of Z.

Thus, any role of an STD is based on a partition, a particular set of phases of the STD and of connecting traps between them. Per partition there is exactly one such role. As a role is an STD, always exactly one state of it is current: the current state of a π-role of STD Z, being a phase from partition π of Z, is the constraint from π *in force* on Z. Note, for this paper we can assume for every partition π of STD Z as defined above, $\bigcup_{i \in I} \mathtt{st}_i = \mathtt{ST} \wedge \bigcup_{i \in I} \mathtt{ac}_i = \mathtt{AC} \wedge$

$\bigcup_{i \in I} \mathtt{tr}_i = \mathtt{TR}$, as coordination within one model remains unchanged. In cases as self-adaptation [3] however, not addressed here, the assumption is abandoned.

In view of the CS example, we assemble phases and traps from Figure 3 into partition CS of Client_i, and we add to CS the two (not drawn) trivial traps triv(Interrupt) and triv(With). In line with our above observations concerning enabling of smooth consecutive imposing of phases on a Client_i, we model role $\mathsf{Client}_i(\mathsf{CS})$ as in Figure 4. Please note, no initial state is indicated.

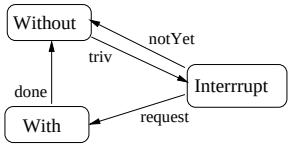

Figure 4: role STD $\mathsf{Client}_i(\mathsf{CS})$ of Client_i.

In particular, the four phase transfers discussed earlier in the context of connectivity of traps, reappear in role $\mathsf{Client}_i(\mathsf{CS})$ as the four steps CS-role behaviour exists of: $\mathsf{Without} \xrightarrow{triv} \mathsf{Interrupt}$, $\mathsf{Interrupt} \xrightarrow{notYet} \mathsf{Without}$, $\mathsf{Interrupt} \xrightarrow{request} \mathsf{With}$ and $\mathsf{With} \xrightarrow{done} \mathsf{Without}$. Thus, in a pleasantly condensed form, dynamicity of constraints imposed on a particular Client_i is as follows: from Without via Interrupt either directly back to Without or continued via With and then back to Without. In a more enriched form, with connecting traps in place as guards for phase transfers, dynamicity of constraints imposed on Client_i alternated with constraints committed to by Client_i is as follows: from Without and after trivial progress within Without via Interrupt either after progress as far as trap notYet directly back to Without or after progress as far as trap request continued via With and after progress as far as trap done then back to Without. Note how our explanation of role dynamics leans heavily on progress actually made at the detailed level of the underlying STD. Specifying these and other behavioural dependencies between different STDs, is determined by Paradigm's operational semantics, the topic of the subsection 3.1.

As a further detail of the example we require, each role STD $\mathsf{Client}_i(\mathsf{CS})$ has as initial state either Without or Interrupt. In the situation of a complete concrete coordination solution, we still have to indicate the initial state for every $\mathsf{Client}_i(\mathsf{CS})$ role separately. Here too, behavioural dependency between different STDs is involved.

2.2. Paradigm, the notions for architecturing collaborations

Not only for the example role of Figure 4 but for every role in general, a connecting trap marks the readiness of a phase to be changed into another phase within the role. Such readiness expresses that sufficient progress has been made within the detailed STD underlying the role. In view of behavioural influencing or dependency between an underlying STD and a role thereof, this means, if a phase is the imposed constraint in force –the current role state– and if within the *underlying detailed* STD the connecting trap has been entered, the condition is fulfilled to take the transition labeled with that connecting trap.

Such a role transition, with the connecting trap as transition label, establishes a phase transfer within the *global role* STD. The idea here is, a role is to provide a consistent and global view on the ongoing detailed dynamics of the underlying STD. If phases and traps have been chosen well, such a global view, through its dynamic character, expresses precisely the dynamics essential for coordinating the underlying STD via its role.

So far, we have discussed the sequential composition of constraints into a role: imposed phases, alternated with traps committed to and subsequent phase transfer. Bringing constraints, as composed into a role STD, into the effect Paradigm is aiming at, comes down to maintaining dynamic consistency. Arrived at this point of our introduction into Paradigm, we summarize the part of such dynamic consistency specifying the behavioral dependencies between any underlying STD and every role STD of it. As we shall point out, this part of dynamic consistency, based on the notions for a single component as given in Subsection 2.1, makes up a solid half of Paradigm's behavioural dependencies or behavioural influencing.

On the one hand, the current state of any role STD, being a phase, constrains the actions that can be taken by the underlying STD in its current state, to those belonging to the current phase. The same holds if two or more role STDs dynamically constrain their common, underlying STD. On the other hand, the current detailed state belonging to a particular trap, is a commit towards the role STD at the level of the partition the trap belongs to: the detailed STD shall stay within the trap until a next phase is imposed by that role STD.

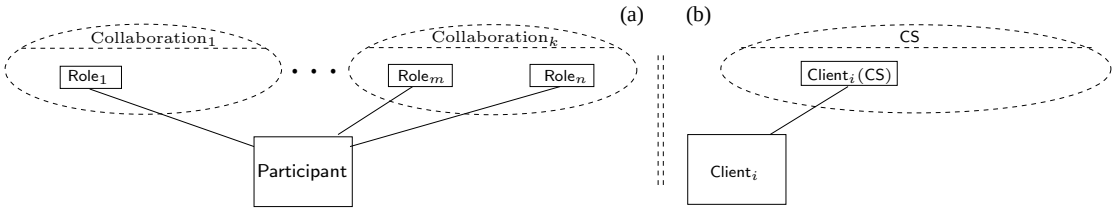

Figure 5: Vertical consistency between participant and its roles: (a) general, (b) CS case.

In the structural, architectural style of UML's component and collaboration diagrams (version 2.0 and higher), Figure 5 visualizes such behavioural dependency between a single component participating in various collaborations through its different roles. Part (a) depicts the general situation and part (b) specializes this for the CS collaboration with respect to a single Client$_i$ and its Client$_i$(CS) role.

The figure is structural only, lacking all dynamical dependency details between a participant and its roles. But the above description of the half of Paradigm's behavioural influencing, through phase constraints imposed by a role and trap constraints committed to towards a role, complements precisely these dynamical dependency details. In this manner –i.e. the way Paradigm defines its constraints as well as their constraining effects– any role remains dynamically consistent with the underlying detailed STD and any detailed STD

remains dynamically consistent with every role of it. By assuming that a starting state of an underlying STD belongs to every phase being starting state of a role of the underlying STD, it then follows: at any later moment the current detailed state of the underlying STD belongs to any of its current phases too. This is particularly true, at the very moment a step is taken and the new state is reached: a detailed step cannot leave the trap of the current phase already entered, Paradigm's trap feature; a global step cannot lead to excluding any state belonging to the connecting trap labeling the global step, Paradigm's connectivity feature. This is Paradigm's dynamic consistency between an underlying STD and all of its roles, also called *vertical dynamic consistency*: defined in terms of the Paradigm notions of phases and traps and without any further synchronization. See Figure 5b for a declarative, structural announcement where in the CS collaboration the vertical consistency is to be found. Thus, maintaining vertical consistency is fully non-synchronous, as it only depends on what phase or trap constraints are in force already, on the moment a decision on a next step has to be taken.

The vertical consistency only partially covers dependencies between the various STD behaviors within and around a collaboration: between one underlying detailed STD and each of its own roles. So, no dependencies have been addressed as yet, between different underlying STDs or between roles of different underlying STDs. To provide a clear and complete overview of underlying STDs participating in the same collaboration and their relevant dependencies, Figure 6 complements the overview from Figure 5 with the dependencies between all roles contributed to one collaboration.

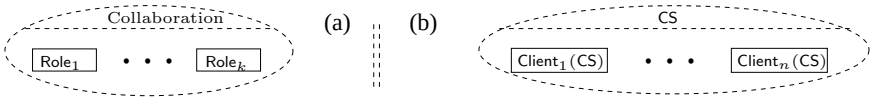

Figure 6: Horizontal consistency between all roles in a collaboration: (a) general, (b) CS case.

The control of actually taking a role step, being a phase transfer, is governed by the consistency rules. Via a consistency rule different roles can be taken into account, relating the behaviors of individual components via their roles only and, moreover, relating these behaviors in view of the coordination one wants to achieve. Thus, consistency rules govern *horizontal dynamic consistency*. Other than with vertical consistency which is communicated asynchronously, consistency rules establish communication between roles via synchronization only. In more detail, a consistency rule synchronizes single steps of arbitrarily many global steps from different role STDs. In addition, the ensemble of synchronized roles steps as occurring in a single consistency rule, can be extended with at most one step of a detailed STD, involved in the synchronization of the ensemble too. Such a single detailed step synchronized with one or more roles steps, is not conceived as participation in the collaboration, but as conducting of the collaboration. Such guidance was not yet expressed in Figure 6, so Figure 7 repairs this as follows. For one collaboration additional detailed STDs can be

involved, not as Participants but as so-called Conductors, meaning that at least one detailed step of a Conductor is synchronized with one or more role steps of Participants. To that aim, roles involved are grouped inside a so-called protocol box and a Conductor is connected to that protocol box, not via a role, but via itself: one or more single detailed steps of itself, at most one per ensemble of synchronized role steps. Graphically the involvement of a Conductor is expressed via a thin box at the border of the protocol box, so the involvement is positioned within the collaboration. The thin box is connected to the Conductor component contributing the involvement, which like the Participants is positioned outside the collaboration.

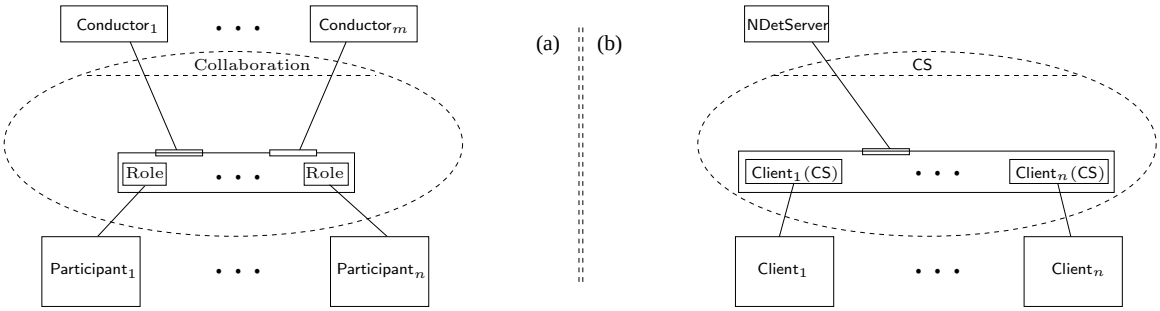

Figure 7: Paradigm's collaboration architecture: (a) general, (b) CS case.

Although part (a) of the figure is general for a single collaboration, it does not express cases where a Conductor and a Participant actually coincide with one component. In such cases in the context of a single collaboration, the component is connected to its role(s) as well as to its thin box. Note, more than one role per component within one protocol box is possible although not very common. Part (b) of Figure 7 announces the existence of a not yet specified component NDetServer involved in the CS collaboration as a Conductor only.

Via consistency rules, horizontal consistency is achieved by synchronizing role steps and by never synchronizing two or more detailed steps. As general consistency rule format we use:

detailed state change $*$ phase transfer , ... , phase transfer

with the various phase transfers taken from different roles. Consistency rules can have rather different appearances. We use the following additional requirements for their format:

- at least one phase transfer is present
- commas are omitted if and only if one phase transfer is present;
- the detailed state change may be omitted, but the "$*$" is always present.

To single out a particular consistency rule or parts thereof on the basis of its format, we use the following terminology (cf. [35]).

- *protocol step*: synonym for consistency rule;
- *orchestration step*: protocol step with a detailed state change;
- *choreography step*: protocol step without a detailed state change;
- *protocol*: a set of protocol steps covering all steps of the roles involved;
- *choreography*: a protocol with choreography steps only.
- *orchestration*: a protocol with at least one orchestration step;
- *conductor*: detailed STD of which a state change occurs in the protocol;
- *participant*: detailed STD with a role covered by the protocol.

Moreover, in [21] it is mentioned how to add so-called change clauses for incorporating data updates into consistency rules. In this paper however, we shall not use change clauses.

The terminology implies, every consistency rule or protocol step is recognizable by the $*$, followed by a non-empty series of phase transfers; choreography steps are recognizable by the $*$ in front; orchestration steps are recognizable by the conductor step in front. When developing a Paradigm model, it can be very useful to have the disposal of an easy to understand sequentialization of protocol steps, either sequentialized completely or at least piece-wise. The taking of the various protocol steps then can be considered as the execution of one or more threads: one thread if the sequentialization is complete and if the sequentialization is piece-wise, then one thread per sequential piece. To achieve such an easy to understand sequentialization, it is often useful to have components involved for conducting the various protocol steps, commonly one such conducting component per collaboration. It then is as if a particular conductor, specifically developed to that aim, takes the initiative in performing each protocol step separately, thus realizing the sequentialization as wanted, exactly in accordance to the thread-like structure of the conductor's detailed behaviours specified through its detailed STD. Precisely this happens to be the case for the CS example in this subsection. But as we shall see in Subsection 2.3, for this example we can get rid of the conducting steps, by turning our orchestration steps into choreography steps, thereby making the one conductor of the example superfluous.

So, before formulating the orchestration steps of the Paradigm model for the CS collaboration, we introduce NDetServer first, in view of its involvement as a conductor. In view of the clarity of understanding the sequentialization of the consistency rules, we let NDetServer conduct every phase transfer of the CS role of every Client_i. So the detailed behaviours of NDetServer are to mirror the separate global behaviours of every $\mathsf{Client}_i(\mathsf{CS})$ role. Moreover, in view of the critical section requirement, composition of role behaviours thus mirrored, has to guarantee: at most one role at a time is in its global state (phase) With (mutual exclusion). For this subsection we shall not take fairness into account, as we shall be sufficiently content with meeting the mutual exclusion

requirement. But in Subsection 2.3, we shall improve our coordination solution in that respect.

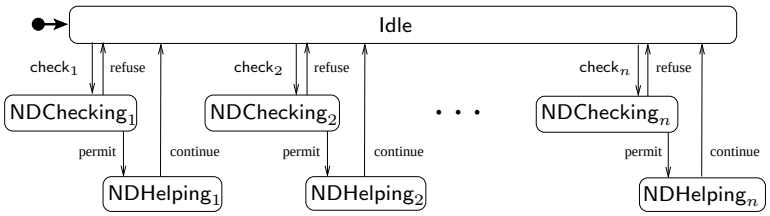

Figure 8: STD non-deterministic server NDetServer.

The STD of the non-deterministic server NDetServer is drawn in Figure 8. The NDetServer component starts in Idle where neither it allows any Client_i to do anything critical nor it checks any Client_i for having a wish to do so. In $\mathsf{NDChecking}_i$ it checks Client_i only, where a negative result of the checking leads to taking action refuse back to Idle and where a positive result of the checking leads to taking action permit, thus proceeding to $\mathsf{NDHelping}_i$. In $\mathsf{NDHelping}_i$ the permission for doing anything critical is given exclusively to Client_i. Only after Client_i does no longer need the permission, action continue is taken back to Idle again. Having explained NDetServer's initial state Idle as disallowing all critical activity without so much as checking for wishes to do so, we moreover choose global state Without as initial state of each $\mathsf{Client}_i(\mathsf{CS})$ role.

The above informal explanation is exactly so covered by the following four consistency rules, each being an orchestration step, and moreover synchronizing the one detailed conductor step with no more that one phase transfer. Remember we have $1 \leq i \leq n$.

$$\mathsf{NDetServer}\colon \mathsf{Idle} \xrightarrow{check_i} \mathsf{NDChecking}_i \ * \ \mathsf{Client}_i(\mathsf{CS})\colon \mathsf{Without} \xrightarrow{triv} \mathsf{Interrupt} \tag{1}$$

$$\mathsf{NDetServer}\colon \mathsf{NDChecking}_i \xrightarrow{refuse} \mathsf{Idle} \ * \ \mathsf{Client}_i(\mathsf{CS})\colon \mathsf{Interrupt} \xrightarrow{notYet} \mathsf{Without} \tag{2}$$

$$\mathsf{NDetServer}\colon \mathsf{NDChecking}_i \xrightarrow{permit} \mathsf{NDHelping}_i \ * \ \mathsf{Client}_i(\mathsf{CS})\colon \mathsf{Interrupt} \xrightarrow{request} \mathsf{With} \tag{3}$$

$$\mathsf{NDetServer}\colon \mathsf{NDHelping}_i \xrightarrow{continue} \mathsf{Idle} \ * \ \mathsf{Client}_i(\mathsf{CS})\colon \mathsf{With} \xrightarrow{done} \mathsf{Without} \tag{4}$$

The above four rules are all protocol steps we have in the CS example, so they specify all horizontal consistency we have in our coordination solution. In view thereof it is worthwhile to make the following observations. According to rule 1, NDetServer checks the Clients in an arbitrary order, as in Idle it selects its action check_i non-deterministically. In this manner, rule 1 is immediately applicable if NDetServer is in state Idle for any of the n different values of i. Contrarily, according to rules 2 and 3, NDetServer chooses deterministically between its two actions in every state $\mathsf{NDChecking}_i$, as the choice is being determined by the two disjoint connecting traps notYet and request of phase Interrupt of Client_i. Also in rules 2 and 3 there is immediate applicability, in this case either of rule 2 or of rule 3; but choosing between the two rules has to be done before, as distinction

between the two traps has to be established first, on the basis of which of the two traps has indeed been entered. According to rule 4, in every state NDHelping$_i$, only one action, continue, can be chosen and will be chosen eventually, but only after progress within phase With has gone so far as entering trap done. So, there is no immediate applicability of rule 4, but there is eventual applicability under mild progress assumptions.

The above four rules constitute the protocol of our unfair coordination solution: unfair as the nondeterministic action selection in state Idle is not necessarily fair. As can be verified straightforwardly, the protocol does cover indeed all four phase transfers in every Client$_i$(CS) role.

In the next subsection other, in some senses better solutions of the same critical section situation will be presented. These examples will in addition cover the case of a choreography. What will not be addressed however, is conducted conducting, where a component is involved in a collaboration both as participant and as conductor. Hence, also self-conducting will not be discussed here.

2.3. More Paradigm model examples

In this subsection we present two variant Paradigm models for the same problem situation as sketched by the CS collaboration given in Figure 6b. Where we introduce new model details or where we encounter not yet illustrated Paradigm features, we shall clarify them. For the rest we keep our discussions short. As we shall reuse the above model, we first summarize the above model very briefly, by listing model fragments in Figure 9. Note, the supplementary collaboration diagrams given do not belong to Paradigm, but they are borrowed from UML for being useful as architectural overview.

Model 1: collaboration diagram CS-NDetServer	
CS-solution with NDetServer, introduced as CS in Figure 7b	
1.1	participants Client$_i$ with $i = 1, \ldots, n$, see Figure 1
1.2	partition CS of Client$_i$, see Figure 3
1.3	role Client$_i$(CS), see Figure 4
1.4	each Client$_i$(CS) role has Without as initial state
1.5	conductor NDetServer, see Figure 8
1.6	rules 1—4, see Subsection 2.2

Figure 9: collaboration CS-NDetServer

Compared with model 1, the main difference of model 2 lays in the conductor RoRoServer, as one can conclude from comparing Figures 7b and 10. On the basis of Figure 11 we see, how sequentialization of the application of the new consistency rules is done in view of new policy for allowing a Client to perform its critical activities: a round robin policy. Please note, Clients and their partitions and roles for CS remain unchanged. The idea is, the actual permission is given by RoRoServer to a specific Client$_i$ through being in state RRHelping$_i$, whereas the preceding checking whether it makes sense to give such permission, is done while being in state RRChecking$_i$. This informal description is formally underpinned

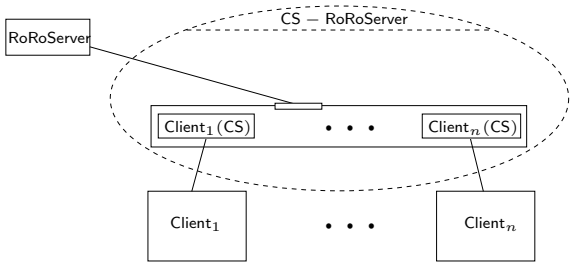

Figure 10: Model 2's collaboration architecture: CS example with a round robin conductor.

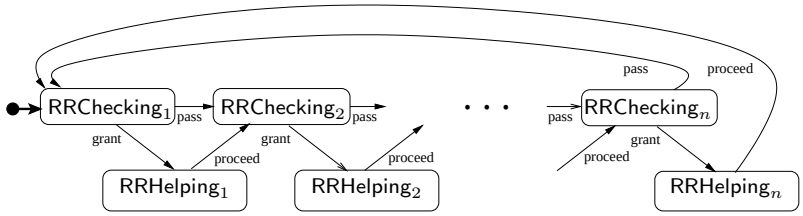

Figure 11: STD round-robin server RoRoServer.

by the following three consistency rules, referred to as rule 5—7.

$$\mathsf{RoRoServer}\colon \mathsf{RRChecking}_i \xrightarrow{grant} \mathsf{RRHelping}_i \ * \\ \mathsf{Client}_i(\mathsf{CS})\colon \mathsf{Interrupt} \xrightarrow{request} \mathsf{With} \tag{5}$$

$$\mathsf{RoRoServer}\colon \mathsf{RRHelping}_i \xrightarrow{proceed} \mathsf{RRChecking}_{i+1} \ * \\ \mathsf{Client}_i(\mathsf{CS})\colon \mathsf{With} \xrightarrow{done} \mathsf{Without},\ \mathsf{Client}_{i+1}(\mathsf{CS})\colon \mathsf{Without} \xrightarrow{triv} \mathsf{Interrupt} \tag{6}$$

$$\mathsf{RoRoServer}\colon \mathsf{RRChecking}_i \xrightarrow{pass} \mathsf{RRChecking}_{i+1} \ * \\ \mathsf{Client}_i(\mathsf{CS})\colon \mathsf{Interrupt} \xrightarrow{notYet} \mathsf{Without},\ \mathsf{Client}_{i+1}(\mathsf{CS})\colon \mathsf{Without} \xrightarrow{triv} \mathsf{Interrupt} \tag{7}$$

Note the difference with the non-deterministic protocol. For instance, in rule 6, the synchronization between RoRoServer's step proceed with the global step done of $\mathsf{Client}_i(\mathsf{CS})$ *and* the global step triv of $\mathsf{Client}_{i+1}(\mathsf{CS})$, exactly expresses the simultaneous events of Client_i no longer being allowed to do her critical section activities and Client_{i+1} being interrupted to be checked. In contrast, in the non-deterministic case, analogous coordination is split in two consistency rules, rules 4 and 1, viz. first a return of NDetServer to idling after helping Client_i, followed by a check of a next Client not necessarily being Client_{i+1}. Model 2 is summarized in Figure 12.

According to the round robin strategy, checking whether a Client_i wants to do its critical activities, is occurring in the strict order from $i = 1, \ldots, i = n$ and then again from $i = 1, \ldots, i = n$, etc. Moreover, upon checking, permission is given if asked for only. Intuitively, this solution is fair.

Compared with models 2 and 1, the main difference of model 3 lays in having

Model 2: collaboration diagram CS-RoRoServer	
CS-solution with RoRoServer, see Figure 10	
2.1	participants Client_i with $i = 1, \ldots, n$, see Figure 1
2.2	partition CS of Client_i, see Figure 3
2.3	role $\mathsf{Client}_i(\mathsf{CS})$, see Figure 4
2.4	the $\mathsf{Client}_1(\mathsf{CS})$ role has Interrupt as initial state, every other $\mathsf{Client}_i(\mathsf{CS})$ role has Without as initial state
2.5	conductor RoRoServer, see Figure 11
2.6	rules 5—7, see below

Figure 12: collaboration CS-RoRoServer

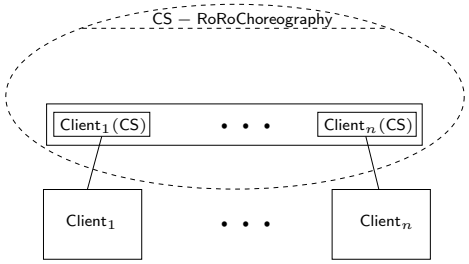

Figure 13: Model 3's collaboration architecture: CS example of round robing choreography.

no conductor at all as one can conclude from comparing Figures 7b, 10 and 13, in particular it has no RoRoServer at all. Please note, Clients and their partitions and roles for CS remain unchanged once more. The consistency rules from model 2 do change necessarily, as the conducting steps have to be omitted. But apart from that, we can really reuse them (the parts behind their $*$-es), as the same sequentialization as in model 2 emerges from these parts, see the rules 8—10 below.

$$* \ \mathsf{Client}_i(\mathsf{CS}) \colon \mathsf{Interrupt} \xrightarrow{request} \mathsf{With} \tag{8}$$

$$* \ \mathsf{Client}_i(\mathsf{CS}) \colon \mathsf{With} \xrightarrow{done} \mathsf{Without}, \ \mathsf{Client}_{i+1}(\mathsf{CS}) \colon \mathsf{Without} \xrightarrow{triv} \mathsf{Interrupt} \tag{9}$$

$$* \ \mathsf{Client}_i(\mathsf{CS}) \colon \mathsf{Interrupt} \xrightarrow{notYet} \mathsf{Without}, \ \mathsf{Client}_{i+1}(\mathsf{CS}) \colon \mathsf{Without} \xrightarrow{triv} \mathsf{Interrupt} \tag{10}$$

It is noted, for model 1 it is possible to formulate a choreography too, be it with some slightly more complex details in the rules. Figure 14 presents model 3 in brief.

3. Paradigm: operational semantics and general positioning

This section strengthens Paradigm's formality in view of analysis. It does so in isolation, by introducing operational semantics for Paradigm in Subsection 3.1. It moreover does so in relation to other coordination approaches, by clarifying Paradigm's position amidst them in Subsection 3.2. Paradigm's operational

Model 3: collaboration diagram CS-RoRoChoreography	
CS-solution with round robin choreography, see Figure 13	
3.1	participants Client_i with $i = 1, \ldots, n$, see Figure 1
3.2	partition CS of Client_i, see Figure 3
3.3	role $\mathsf{Client}_i(\mathsf{CS})$, see Figure 4
3.4	the $\mathsf{Client}_1(\mathsf{CS})$ role has Interrupt as initial state, every other $\mathsf{Client}_i(\mathsf{CS})$ role has Without as initial state
3.5	rules 8—10, see below

Figure 14: collaboration CS-RoRoChoreography

semantics expresses the effects component dynamics have on each other; the positioning underlines the structure these effects have, being so characteristic for Paradigm.

3.1. Paradigm: operational semantics

To formulate operational semantics for all behavioural dependencies relevant for collaborations, we introduce the following Cartesian product space for a Paradigm model: any single dimension of the product space consists of the state space of a single STD, detailed or global, every STD its own dimension. Formally, assume a collaboration Coll consisting of n, $(n \geq 1)$ participants, each having a detailed STD, $Z_i = \langle \mathtt{ST}_i, \mathtt{AC}_i, \mathtt{TR}_i \rangle$, $i = 1, \ldots, n$. Assume that the ith component has m_i roles $(m_i \geq 0)$. Role $Z_i(\pi_{ij}) = \langle \widehat{\mathtt{ST}}_{ij}, \widehat{\mathtt{AC}}_{ij}, \widehat{\mathtt{TS}}_{ij} \rangle$ is at the level of a partition π_{ij}, for $j = 1, \ldots, m_i$. The state space of the entire system Coll is the Cartesian product of the state spaces of each STD, detailed or global,

$$\mathsf{Coll}\mathtt{ST} = \langle s_i, \langle s_{ij} \rangle_{j=1}^{m_i} \rangle_{i=1}^{n} \tag{11}$$

for $s_i \in \mathtt{ST}_i$ and $s_{ij} \in \widehat{\mathtt{ST}}_{ij}$. According to the definition, thus, $s_{ij} = \langle st_{ij}, ac_{ij}, tr_{ij} \rangle$ is a phase of Z_i and $st_{ij} \subseteq \mathtt{ST}_i$, $ac_{ij} \subseteq \mathtt{AC}_i$ and $tr_{ij} \subseteq \mathtt{TR}_i$. Note that for the collaboration Coll we do not distinguish between conductors and participants unless it is explicitly stated. If we talk about a conductor and participants directed by the conductor, we take, for the sake of simple notation, the conductor component as the first entry in the Cartesian product.

Due to vertical consistency and consistency rules, reflecting as such the horizontal consistency, the dynamics of separate components are consistently integrated in the collaboration behaviour. Thus, to determine which transitions are possible for the overall collaboration, Coll, current states and/or enabled transitions of the STDs of the relevant components need to be checked. This information can always be extracted from a current state of Coll.

As explained earlier, the vertical consistency is demonstrated in two ways. On the one hand, the current state of any role STD, being a phase, constrains the actions that can be taken by the underlying detailed STD. The same holds if two or more role STDs dynamically constrain their common, underlying STD. We illustrate the effect of the phase constraint on the CS–NDetServer collaboration involving only one Client. A system with only one client is simple but sufficient

enough to illustrate the consistency between the detailed and the global dynamics. (Behavioural influencing between more clients is part of the horizontal consistency discussed later.) The state space of CS collaboration with one Client is the set of all triplets $\langle s, cl, cg\rangle$ where s ranges over the states of NDetServer, cl ranges over the (local) states of the Client STD and cg ranges over the (global) states of the Client(CS) STD. Note that this is the total state space, and some states may not be reachable from the initial one $\langle$Idle, Out, Without$\rangle$. Now consider possible behaviour of the collaboration CS in state $\langle$Idle, Out, Without$\rangle$. The current phase Without allows the Client to take the enter transition, as enter belongs to Without. Other phase constraints are not imposed on enter, therefore

$$\langle \mathsf{Idle}, \mathsf{Out}, \mathsf{Without}\rangle \xrightarrow{\mathsf{enter}} \langle \mathsf{Idle}, \mathsf{Waiting}, \mathsf{Without}\rangle.$$

In state $\langle$Idle, Waiting, Without$\rangle$, however, the collaboration CS cannot take transition explain. Although this transition is enabled in the current local state Waiting of Client, the current phase Without imposes a constraint on this transition, explain does not belong to Without, hence

$$\langle \mathsf{Idle}, \mathsf{Waiting}, \mathsf{Without}\rangle \xrightarrow{\mathsf{explain}} .$$

On the other hand, again due to the vertical consistency, a phase transfer can occur only if the local dynamics of the component corresponding to that phase has progressed far enough, and the corresponding trap has been entered. Such checking has to be taken into account too, when defining the collaboration transitions. Consider again the CS collaboration with one Client. Assume state $\langle$NDHelping, Busy, With$\rangle$ to be current. According to the CS role of the client, Figure 4, global transition done might be taken in phase With. However, being currently in the local state Busy means that the client has not entered the trap done yet, Busy $\notin$ done. Thus in the current state the done transition cannot be taken yet: Client has not progressed far enough locally. Assume now state $\langle$NDChecking, Waiting, Interrupt$\rangle$ to be current. According to the CS role of the client, two global transitions might be taken: notYet and request. However, the current local state of Client imposes a constraint, and determines which transition is possible. Namely, being in the local state Waiting for Client means that in the phase Interrupt the trap request is entered but not the trap notYet. Thus, the phase transfer can happen only from Interrupt to With by taking the global transition request, but not notYet. Stated slightly differently, Waiting $\in$ request but Waiting $\notin$ notYet.

Note, the client in its current configuration of local state Waiting and global state Interrupt can only *contribute* transition request to a collaboration step of CS. "Contributing" a transition to a collaboration step does not straightforwardly mean that the transition, by default, is allowed to be taken by the collaboration, since, as we discuss below, further checks involving the other components (e.g. the server) are needed.

For vertical consistency we define two collaboration transitions. To ease the notation, we use as shorthand: if $\mathsf{s} = (x_1, \ldots, x_k)$ is a k-tuple, then $\mathsf{s}[x_i := y_i]$

denotes the k-tuple obtained from s by substituting the ith entry x_i by y_i and by keeping all other entries unchanged, $\mathsf{s}[x_i := y_i] = (x_1, \ldots, x_{i-1}, y_i, x_{i+1}, \ldots, x_k)$. The notation can be extended for more substitutions accordingly.

Let $\mathsf{s} = \langle s_i, \langle s_{ij} \rangle_{j=1}^{m_i} \rangle_{i=1}^{n}$ be a state in CollST.

- (Transitions – part 1) If for some $i, 1 \leq i \leq n$,
  1. $s_i \xrightarrow{a} s_i'$ for some state s_i' in Z_i, and
  2. for all $j = 1, \ldots, m_i$, $s_i \xrightarrow{a} s_i' \in tr_{ij}$

  then $\mathsf{s} \xrightarrow{a} \mathsf{s}[s_i := s_i']$ is a transition in collaboration Coll. This transition is called a *consistent detailed transition*.

- (Contributed transitions to collaboration) If for some $i, 1 \leq i \leq n$, and some $j, 1 \leq j \leq m_i$,
  1. $s_{ij} \xrightarrow{\mathsf{trap}} s_{ij}'$ is a global transition (a phase transfer via trap trap) of the role $Z(\pi_{ij})$ with the underlying detailed STD Z_i, and
  2. for state s_i of the detailed STD Z_i it holds: $s_i \in \mathsf{trap}$

  then the role $Z(\pi_{ij})$ *contributes transition* $s_{ij} \xrightarrow{\mathsf{trap}} s_{ij}'$ to the collaboration Coll in state s.

Note that the first aspect of the vertical consistency directly yields a transition of the collaboration. The notion of "contribution to a collaboration step", which reflects the second aspect of the vertical consistency may not lead to a collaboration step: the contribution check is required, as we see later, for all global steps that synchronize into a collaboration steps (specified in the Paradigm model via a consistency rule). The notion of a contributed transition enables us to reason not only about the overall collaboration but also about the behaviour of its subsystems. For instance, we may reason and derive the dynamics of a single component, as a new STD. This new STD combines the detailed STD and all global STDs of the considered component in a Cartesian product space with all consistent detailed transitions and all transitions that the component contributes to the collaboration. This construction will be exploited in Sections 4 and 5 to establish a relation between STDs and their translations in process algebra in a component-wise way, rather than for the overall collaboration.

The horizontal consistency imposes certainly additional constraints on the behaviour of Coll. As already mentioned, several phase transfers (for different global STDs) may occur simultaneously. In the Paradigm language this scenario is specified as a consistency rule that synchronizes all involved global transitions. This synchronization may be conducted by a local conductor step in which case we have an orchestration step, or may not be conducted by any local transition, in which case we have a choreography step.

Recall that the consistency rules 1–4 and 5–7 define orchestration steps for CS collaborations, the first with NDetServer and the second with RoRoServer.

To prepare for the formal definition, we consider again the dynamics of collaboration CS–NDetServer, this time taking the consistency rules into account. Moreover, as the ways components influence and restrict each other (via the consistency rules) are the main focus, we consider a CS collaboration of two clients. Now, assume that $\langle \mathsf{NDHelping}_2, \mathsf{Waiting}_1, \mathsf{Without}_1, \mathsf{AtDoor}_2, \mathsf{With}_2 \rangle$ is the current state of the CS collaboration. As discussed above, Client_1 can contribute transition $\mathsf{request}_1$ to a collaboration step in this state. According to the consistency rule 3 (page. 13) this step of Client_1 must be executed simultaneously with permit_1 step of NDetServer. However, NDetServer cannot perform permit_1 at its current state, therefore, this contribution of Client_1 does not yield a collaboration step. At the same state, on the other hand, the second client contributes done_2. The NDetServer in its current state $\mathsf{NDHelping}_2$ (Figure 8) can perform $\mathsf{continue}_2$. Note that no vertical constraint is imposed on this transition as NDetServer has no roles (partitions) in collaboration CS. Finally, we find that consistency rule 4 (page 13) binds these two transitions in one synchronized collaboration step. By putting all together, we obtain that CS collaboration can make the following step

$$\langle \mathsf{NDHelping}_2, \mathsf{Waiting}_1, \mathsf{Without}_1, \mathsf{AtDoor}_2, \mathsf{With}_2 \rangle \xrightarrow{\mathsf{cr4}_2} \langle \mathsf{Idle}, \mathsf{Waiting}_1, \mathsf{Without}_1, \mathsf{AtDoor}_2, \mathsf{Without} \rangle$$

where $\mathsf{cr4}_2$ is a newly introduced action name to denote the synchronization of done_1 and $\mathsf{continue}_1$, referring to consistency rule 4 applied on the second component. Formally,

- (Orchestration transitions – part 2) Let $\mathsf{s} = \langle c, \langle c_i \rangle_{i=1}^{k}, \langle s_i, \langle s_{ij} \rangle_{j=1}^{m_i} \rangle_{i=1}^{n} \rangle$ be a state in CollST. If p components $i_1, \ldots, i_p$ $(p \geq 1)$ synchronize on phase transfers via their roles $Z(\pi_{i_t j_t})$, $1 \leq t \leq p$, conducted by the detailed step of the conductor C, such that

  1. $c \xrightarrow{\mathsf{man}} c'$ is a consistent detailed transition of the conductor C in local state c
  2. the role $Z(\pi_{i_t j_t})$ of the i_t-th component, $1 \leq t \leq p$, contributes transition $s_{i_t j_t} \xrightarrow{\mathsf{trap}_{\mathsf{i_t}}} s'_{i_t j_t}$ to a collaboration in state s, and
  3. the following consistency rule is defined for the collaboration Coll:
     $$c \xrightarrow{\mathsf{man}} c' \quad * \quad s_{i_1 j_1} \xrightarrow{\mathsf{trap}_{\mathsf{i_1}}} s'_{i_1 j_1}, s_{i_2 j_2} \xrightarrow{\mathsf{trap}_{\mathsf{i_2}}} s'_{i_2 j_2}, \quad \ldots \quad , s_{i_p j_p} \xrightarrow{\mathsf{trap}_{\mathsf{i_p}}} s'_{i_p j_p}$$

  then $\mathsf{s} \xrightarrow{\mathsf{cr\_orch}} \mathsf{s}[c: = c', s_{i_t j_t}: = s'_{i_t j_t}]_{t=1}^{p}$ is a transition in collaboration Coll. Here cr_orch is a newly introduced action name that implicitly refers to the involved actions of the synchronizing components and the consistency rule.

Note that the vertical consistency on the detailed step of the conductor is implicit, although indicated by consistent in condition part 1.

To illustrate dependencies between components' dynamics that lead to a choreographic transition at the level of collaboration, we use the third critical section solution from Section 2.3, CS–RoRoChoreography corresponding to Figure 13. As intended, this solution does not make use of any server (conductor)

but the critical section is solved by a proper synchronization of the participants via choreography. Obviously, the choreography steps for the collaborations are simpler than the orchestration steps, from the aspect of the consistency checks. Consider the CS–RoRoChoreography collaboration involving two clients only. Following the same line of reasoning as in the earlier examples, we observe that Client_1 can contribute triv_1 and Client_2 can contribute done_2 to a collaboration step in state $\langle \mathsf{Waiting}_1, \mathsf{Without}_1, \mathsf{AtDoor}_2, \mathsf{With}_2 \rangle$. The consistency rule 9 (page 15) binds these two transitions into a synchronizing action. Therefore, we conclude the following transition can be taken

$\langle \mathsf{Waiting}_1, \mathsf{Without}_1, \mathsf{AtDoor}_2, \mathsf{With}_2 \rangle \overset{\mathsf{cr9_{21}}}{\rightarrow} \langle \mathsf{Waiting}_1, \mathsf{Interrupt}_1, \mathsf{AtDoor}_2, \mathsf{Without}_2 \rangle$,

where $\mathsf{cr9_{21}}$ is a newly introduced action name to denote the synchronization of done_2 and triv_1, referring to consistency rule 9 for $i = 2$. Formally,

- (Choreography transitions – part 3) Let $\mathsf{s} = \langle s_i, \langle s_{ij} \rangle_{j=1}^{m_i} \rangle_{i=1}^{n}$ be a state in CollST. If p components $i_1, \ldots, i_p$ $(p \geq 1)$ synchronize on phase transfers via their roles $Z(\pi_{i_t j_t})$, $1 \leq t \leq p$, such that

  1. the role $Z(\pi_{i_t j_t})$ of the i_tth component, $1 \leq t \leq p$, contributes transition $s_{i_t j_t} \overset{\mathsf{trap_{i_t}}}{\longrightarrow} s'_{i_t j_t}$ to the collaboration in state s, and
  2. the following consistency rule is defined for the collaboration Coll:

     $$* \qquad s_{i_1 j_1} \overset{\mathsf{trap_{i_1}}}{\rightarrow} s'_{i_1 j_1}, s_{i_2 j_2} \overset{\mathsf{trap_{i_2}}}{\rightarrow} s'_{i_2 j_2}, \quad \ldots \quad , s_{i_p j_p} \overset{\mathsf{trap_{i_p}}}{\rightarrow} s'_{i_p j_p}$$

  then $\mathsf{s} \overset{\mathsf{cr\_chor}}{\rightarrow} \mathsf{s}[s_{i_t j_t} := s'_{i_t j_t}]_{t=1}^{p}$ is a transition in collaboration Coll. Here cr_chor is a newly introduced action name that implicitly refers to the synchronizing components and the consistency rule.

The operational semantics presented in this section captures all dependencies between detailed component dynamics and the overall coordination dynamics. We opt for this approach as its advantages are twofold: first, it gives better understanding of Paradigm and reveals that its underlying semantics is heavily based on synchronization of components working in parallel, and second, it brings the Paradigm language and Paradigm models to a level close to process algebraic reasoning. As the underlying semantics of process algebra is also based on STDs (usually referred to as LTSs), establishing a relation between a Paradigm model and its process algebraic representation boils down to relating STDs. Thus, translating Paradigm into process algebra comes very naturally. Also note that no restriction has been assumed on the hierarchical structure of Paradigm models. The way the operational semantics is defined is general enough to capture the behaviour of a component that appear in the collaboration both as a conductor and as a participants, possibly in the same protocol.

3.2. Paradigm positioned as coordination modeling language

In Section 2 we have introduced Paradigm. In Subsection 3.1 we have formulated Paradigm's operational semantics, actually by shaping the constraining effect of both phases and traps in the more usual format of step dependencies between

all STDs involved. In this Subsection 3.2 we relate Paradigm to the framework for coordination models and languages as presented in [14] as well as to the architectural description language WRIGHT from [1]. As we shall point out, Paradigm fits well into both. Moreover, the well-fitting underlines the special character and key relevance of Paradigm's constraint notions of phases and of traps. These dynamic constraints and the dynamic compositions thereof are clarifying for Paradigm's position amidst other coordination languages.

According to [14] coordination models should be built from coordination entities, coordination media and coordination laws. A coordination language then should allow for describing coordination models orthogonally combined with sequential computation models. For Paradigm this is indeed the case, as can be argued as follows. Detailed STDs of Participants are the coordination entities, the thread-like units to be coordinated. By their step-wise behaviours these STD descriptions moreover provide the computational model for the intra-unit, algorithmic actions. Roles or global STDs are Paradigm's coordination media. A particular role then serves as the means of asynchronous communication for a coordination entity in view of coordination; communication sent in the form of traps committed to, as well as communication received in the form of phases imposed. The coordination laws are the consistency rules. They synchronize role steps of different roles, thus regulating, via their roles as coordination media, the coordination entities in the context of such roles only, exactly according to the coherence expressed by a consistency rule for a single synchronized step. In case of a choreography, this is all there is. In case of an orchestration step however, an additional detailed STD step from a Conductor is involved in synchronizing the role steps too: such an additional Conductor step then regulates, as it were more explicitly componentized, the more distributed, more anonymous regulating exerted by and among the synchronized roles steps together, i.e. witin the anonymous collaboration. But if such a detailed Conductor STD is to be coordinated too, this can happen only via the actual coordination medium of a role of it, subject to the coordination law of suitable consistency rules.

In view of the requirements in [14] for a coordination language, it is particularly clarifying to address how Paradigm combines coordination models and computation models. First of all, coordination entities and coordination media are STDs. Moreover, as has been clarified by the operational semantics, the coordination laws together induce STD-like behaviour through the product state space spanned all STDs together. So, in terms of the coordination model ingredients from the framework in [14], Paradigm models consist of STDs only. Second, and most characteristic for Paradigm, the vertical consistency, i.e. the step dependency between a coordination entity and all of its coordination media, is defined via special language constructs: the phases and the traps thereof, grouped into a partition. This is special syntax of the modeling language Paradigm, specifically geared towards semantically achieving vertical consistency in whatever Paradigm model: the semantical effect is the constraining character. In view of horizontal consistency, the language introduces nothing special at all, straightforward synchronization of steps from different coordination media, i.e. from role STDs; possibly, such synchronization between steps

is extended with a detailed Conductor step. But, in the context of a protocol, the semantical effect of a protocol step is a suitable overview of the resulting phase configuration for the protocol's participants. It is the quality of modeling phases, traps and roles in view of the coordination goals to be achieved, that determines the quality of the (consecutive) phase configurations for taking the protocol steps and that hence determines the quality of the coordination.

According to the approach in [1] and embodied in the architectural description language Wright, an architectural description consists of components and connectors. A component comprises the ports of that component and a behavioural component specification. The ports define the logical points of interaction the component is involved in. A connector comprises the roles it expects to connect and the glue according to which the expected roles are to be coordinated. When binding components to connectors, component ports and connector roles that should correspond, are attached to each other. For Paradigm this means the following. Participants are the components and their detailed STDs are the behavioural component specifications. A role or global STD of a Participant is Paradigm's (dynamic) interaction the Participant is involved in, which is to be conceived as being provided at a specific port serving as the scene of action for that interaction – cf. [24] where such ports are explicit parts of the UML collaboration diagrams as given for the Paradigm models discussed. The glue as occurring in a connector, consists of Paradigm's consistency rules. The connector itself then compounds to a protocol in Paradigm. This also means, the roles in a Wright connector are precisely the Paradigm roles involved in the Paradigm protocol: they have the same (global) STD description, be it that when comparing them in more detail, send actions in a port role (at the component) are mirrored as receive actions in the corresponding connector role (at the protocol) and similarly, receive actions in the port role are mirrored as send actions in the connector role. Wright-like attachments thus are superfluous in Paradigm. As mentioned above, the Paradigm language introduces nothing special for the glue: horizontal consistency is achieved via synchronization per protocol step. Contrarily, the Paradigm language introduces its characteristic behavioural constraint notions of phases and traps, per port of a component grouped into a partition. Per partition then a role is constructed, being per definition vertically, dynamically consistent with the component's internal behavioural specification. Moreover, if well chosen, via the phases and traps it is built from, such a role together with other such roles facilitates the step-wise construction of a good protocol, expressing coordination through which a given collaborative goal aimed at, is reached indeed. Thus, Paradigm's dynamic constraints, its phases and traps, serve as language structures guaranteeing vertical consistency. Choosing the right shape of these structures, then facilitates role construction and protocol construction and hence the coordination.

It is through its phases and traps, we want to clarify Paradigm's position among other coordination languages. Coordination languages can be divided into three main regions: data-based, flow-based and transition-based. Examples from the data-based region are Linda and other tuple-space languages [13]. Examples from the flow-based region are languages such as Reo and Focus [6, 12],

where components are connected as well as influenced through streams of data / triggers. Examples from the transition-based region are CSP and Manifold [5]. Paradigm belongs to this region too. The much older Petri nets [31] actually are a mix of these: markings represent data configurations, (some) tokens represent flows and firings represent step-like transitions, synchronized over many, not well-separated threads.

Since [5] in particular, the separation of computation and coordination has been seen as a valuable concept. Less well-separating in that respect are the languages from the data-based region, although via additional structuring through different tuple-types and via corresponding discrimination of processes manipulating the tuples, such separation is achievable.

For the transition-based languages the separation is rather diverse. Older languages in this region, like CSP [33] and team automata [8], are more geared towards interaction. Thus they are not so well-separating on their own, as their transition systems have no notions for discriminating easily between computation and coordination. But newer ones, like Manifold, additionally structure their behavioural units: special manager-like units for coordination tasks, other units for computation tasks.

Flow-based coordination languages strictly follow this distinction: components comprise computation, streams and manipulations thereof through channels comprise coordination. So their separation of computation and coordination is clear. Such clear separation however, causes a consistency gap difficult to bridge. It is not so clear how dynamics within a component, via the flows the component brings about, must lead to or cannot lead to certain dynamics within a different component. The new problem then becomes, how to guarantee that component dynamics and the flows between them are coupled rightly, consistently indeed. Nevertheless, both Reo and the variant of Focus discussed in [11] underline the temporarily restrictive effect a flow must have on a component. But an algorithm for coordination is not so straightforwardly expressed in terms of stream handling; moreover, the behavioral effect of a flow on a component, though being restrictive, is rather declarative instead of operational, as component behaviours are outside the scope of the flow-based languages.

The temporarily restrictive effect of a flow on a component actually attempts to provide a concrete solution to the consistency issue raised through separating coordination and computation: coordination and computation must influence each other consistently over time. Thus they mutually allow and disallow parts of their activity, i.e. they mutually vary their dynamic freedom through restricting the freedom temporarily by dynamically adjusting the restrictions.

The transition-based coordination language Paradigm has the separation of coordination and computation. Moreover, it solves the consistency issue raised by such separation as follows. As a language it offers the notions of phase and trap, to build roles from. In this manner the language guarantees vertical dynamic consistency between a component and any role of it. A well-designed Paradigm model then, like a well-designed program, achieves the purpose of the collaboration through coordination, specified as protocols in the model and, apart from possible conducting steps, in terms of roles only.

4. Translation to process algebra

In this section we translate the three CS examples into process algebra. The examples are used to ease the understanding of the general translation in the next section. The last subsection reports on the verification results established for the specific CS solutions.

4.1. Preliminaries: process algebra

Process Algebra (PA) is a formal framework used for modeling and analyzing systems of concurrent processes. Basic ingredients of any PA are a set of operators, a set of equations, also called axioms or laws capturing relationship between the operators, and equivalences of processes. A PA is typically parameterized by a set of atomic actions A, the atomic activities that a process can perform. In a PA, actions are usually represented by constants. Other basic operators common to most process algebras are action prefix or sequential composition '$\cdot$', non-deterministic choice '$+$' and parallel composition '$\|$'. The process algebra ACP [7, 9] we know well and use as vehicle of analysis, has encapsulation '∂_H' and abstraction 'τ_I' as additional operators. The encapsulation operator '∂_H' is used to block actions from the set $H \subseteq A$, generally to avoid unmatched synchronization actions. The abstraction operator 'τ_I' is used to hide the actions in the set $I \subseteq A$, i.e. any action in I is considered internal and is renamed into τ, the special action denoting internal activity. The internal unobservable action τ is usually treated differently than other observable actions for A.

Processes are described by algebraic expressions. Infinite processes can be expressed by recursive specifications. For instance, the process Client_i, $(1 \leq i \leq n)$, considered as a process in isolation, is described by the following recursive specification.

$$\begin{aligned} \mathsf{Client}_i &= \mathsf{Out}_i \\ \mathsf{Out}_i &= \mathsf{enter}_i \cdot \mathsf{Waiting}_i \\ \mathsf{Waiting}_i &= \mathsf{explain}_i \cdot \mathsf{Busy}_i \\ \mathsf{Busy}_i &= \mathsf{thank}_i \cdot \mathsf{AtDoor}_i \\ \mathsf{AtDoor}_i &= \mathsf{leave}_i \cdot \mathsf{Out}_i \end{aligned}$$

It reads as: process Client_i behaves like process Out_i; process Out_i executes action enter_i and continues to behave like process $\mathsf{Waiting}_i$ afterwards; etc. Note, process Client_i exhibits deterministic behavior without any branching; non-deterministic choice does not occur in the description. From the example it is obvious that process expressions and recursive specifications can be graphically represented as STDs. In fact, the same STD in Figure 1 is obtained from the algebraic specification of process Client_i above by means of the underlying operational semantics of ACP.

Once components are modeled, they can be composed into more complex specifications, specifying more complex processes. Composition is build by means of the operators. Concurrency and interaction of separate processes that run in parallel is captured by the parallel operator, communication function and communication operator. The behaviour of the parallel composition of two

processes is obtained by interleaving their separate behaviours. In addition, processes can synchronize on specific actions. In ACP, synchronization of actions is user-defined via the so-called communication function. The communication function may involve two or more arguments, enabling multi-party synchronization. As we will discuss in Section 4.2, via a well-chosen communication function Paradigm's consistency rules can be naturally expressed in ACP. This will not come as a surprise, as a both communication function and consistency rules are meant for specifying synchronization between behaviours.

Once systems are modeled algebraically, their behaviours can be compared. Comparison is typically done by means of equivalence relations, chosen appropriately to preserve certain properties. More precisely, given a notion of equivalence and its associated logic, two systems are equivalent iff they have the same logical properties. For example, strong bisimulation has the same distinguishing power as Hennesy-Milner logic [28], branching bisimulation identifies the same processes as the temporal logic CTL* without the next operator [15]. Hence, if two systems are strongly bisimilar than they satisfy or dissatisfy the same Hennesy-Milner formula. Conversely, if a property expressed in CTL^* without the next operator holds for one system, then it holds too for any other system that is branching bisimilar.

A wide range of equivalence relations have been studied [19, 18]. Some of them, e.g. strong bisimulation, are appropriate for concrete behaviour, when every action of the system is observable. However, these relations are often too fine when part of the behaviour is preferred to be abstracted away and considered unobservable. In view thereof, we choose for branching bisimulation [20] as the equivalence relation we apply. Indeed, branching bisimulation is the strongest in the spectrum of such equivalence relations, but yet weak enough to identify sufficiently many systems. Moreover, it possesses many other useful properties.

Branching bisimulation, as introduced in [20], is defined as a relation between the states of an STD and lifted to STDs by considering branching bisimilarity of their initial states.

- Given an STD $\langle \mathtt{ST}, \mathtt{AC}, \mathtt{TS} \rangle$, a symmetric relation $R \subseteq \mathtt{ST} \times \mathtt{ST}$ is called a branching bisimulation relation if it satisfies the following transfer property: for any two states $s, t \in \mathtt{ST}$ such that if $R(s,t)$ and $s \xrightarrow{a} s'$ either

  1. $a = \tau$ and $R(s',t)$, or
  2. for some $n \geq 0$, there exist $t_0, t_1, \ldots, t_n$ and t' in $\mathtt{ST}$ such that $R(s,t_i)$ for $0 \leq i \leq n$, $t \equiv t_0 \xrightarrow{\tau} t_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} t_n \xrightarrow{a} t'$ and $R(s',t')$.

  Two states s and t are called branching bisimilar, notation $s \leftrightarrow t$, if there exists a branching bisimulation relation relating s and t. Two STDs, with indicated initial states, are branching bisimilar if their initial states are branching bisimilar.

As elaborated below, process algebraic descriptions of Paradigm models help us to treat their dynamics algebraically and, as we will argue, make them amenable

to PA analysis techniques. In part, this is because the process algebraic treatment makes interaction of detailed and global processes precise. In Paradigm, for instance, the mechanism by which one phase imposes constraints on the detailed STD dynamics is implicit, unless the designer represent the model at a low abstract level, which is very unlikely and undesirable. In process algebra this is explicitly specified. Once all components that build the Paradigm model are translated into process algebra, in principle, the complete behaviour can be computed. Therefore, various verification techniques can be applied to the system obtained. A widely accepted technique for system verification, which we use as well below is model checking. In short, model checking provides automated means to assess the validity of a temporal or modal formula against a model of the system. For instance, one can verify if the property *"at any moment there is at most one client in the critical section"* holds. For our experiments, we have used the toolset `mCRL2` [26, 25].[1] Its formal specification language `mCRL2` is based on ACP, together with facilities for abstract datatypes. More specifically it employs the method of parameterized boolean equation systems (PBES) for symbolic model checking purposes.

4.2. The example models translated into ACP

In this section, the various STDs from the example Paradigm models introduced in Section 2, will be translated into ACP processes. In particular, each STD will be represented by a set of recursively defined processes, extended with specific communication details. The translation, first applied to the example, will help to understand the general translation and results presented in the next section.

Section 2 explains that in a Paradigm model several STDs may belong to the same component, describing the component's dynamics either at various levels of abstraction (detailed vs. global STDs) or describing different roles the component has in various collaborations. Thus the structure and hierarchy of the components is embedded in the Paradigm model. In process algebra, however, all STDs are considered as different processes. They may perform steps independently of each other, and must communicate to exchange information, even the information that they belong to the same component. In other words, the architectural appearance nicely present in Paradigm is completely flattened and more or less lost in the process algebra representation. Therefore, an additional mechanism needs to be used to "reconstruct" the structure of the original Paradigm model.

To realize the vertical consistency in the translation, two aspects must be taken into account. The information about: (1) a current state of a detailed STD, and (2) the current phases that global STDs reside in and their constraining effect, both need to be communicated between (involved) processes, even if they belong to the same component.

The first issue requires some transitions (some of them being loops) to be added to the corresponding process algebra specifications. We use the comple-

[1] Available from `http://www.mcrl2.org`.

mentary actions at!(.) and at?(.) that take detailed states as their arguments. The detailed process sends its current state, via at!(), while the global process after receiving this information, by synchronizing on at?(.), updates its trap information, if applicable.

For the second issue, we use actions ok!(.) and ok?(.) that take the labels of detailed steps as their argument. This reflects that transitions of the detailed STD should be consistent with the current phase of the global STD. The complementary actions synchronize if the step to be taken by the detailed STD is allowed by the current phase as constraint. Thus, actions ok!(·) and ok?(·) interchange this permission. The consistency rules defining the horizontal consistency need to be integrated in the translation too. They are embedded in the communication function defining the synchronization of processes. For the communication within the protocol captured by the consistency rules, here between the server and its clients, actions man(.) on the side of a conductor are meant to complement emp(.) actions on the side of the participants. Thus, the process algebraic representation of role Client(CS) keeps track of transitions (phase transfers) contributed to the collaboration. Synchronization leads to execution of the corresponding consistency rule: a local transition of the conductor, phase changes for the participants involved. In case of a choreography when no conductor is involved, as in the third example in Section 2.3, synchronization clearly includes only emp(.) actions.

For the concrete examples this amounts to the following. We adorn the n processes Client_i with the actions at!(.), conveying state information, and actions ok?(.), regarding transition eligibility.

$$\begin{aligned}
\widehat{\mathsf{Client}}_i &= \mathsf{Out}_i \\
\mathsf{Out}_i &= \mathtt{at!}(\mathsf{Out}_i) \cdot \mathsf{Out}_i + \mathtt{ok?}(\mathsf{enter}_i) \cdot \mathsf{Waiting}_i \\
\mathsf{Waiting}_i &= \mathtt{at!}(\mathsf{Waiting}_i) \cdot \mathsf{Waiting}_i + \mathtt{ok?}(\mathsf{explain}_i) \cdot \mathsf{Busy}_i \\
\mathsf{Busy}_i &= \mathtt{at!}(\mathsf{Busy}_i) \cdot \mathsf{Busy}_i + \mathtt{ok?}(\mathsf{thank}_i) \cdot \mathsf{AtDoor}_i \\
\mathsf{AtDoor}_i &= \mathtt{at!}(\mathsf{AtDoor}_i) \cdot \mathsf{AtDoor}_i + \mathtt{ok?}(\mathsf{leave}_i) \cdot \mathsf{Out}_i
\end{aligned}$$

The definition of process $\widehat{\mathsf{Client}}_i$ assures, the process really starts in close correspondence to starting state Out from Figure 1. The definition of process Out_i expresses: (1) upon being asked, it can exchange state information while keeping the process as-is; (2) it can ask for permission to take the analogue of transition enter from Figure 1, in view of continuing with process $\mathsf{Waiting}_i$ thereafter. Note, in the definition of process Busy_i the possibility for exchange of state information is specified, although asking for it does never occur (in Figure 3: state Busy does not belong to trap done). Therefore, action $\mathtt{at!}(\mathsf{Busy}_i)$ might be simply omitted.

In a similar manner, process $\widehat{\mathsf{Client}}_i(\mathsf{CS})$ is defined in close correspondence to $\mathsf{Without}_i[\mathsf{triv}]$. The processes $\widehat{\mathsf{Client}}_i(\mathsf{CS})$ is augmented with the actions at?(.) and ok!(.). The ok!(.)-actions provide the permission answers to requests from $\widehat{\mathsf{Client}}_i$ to take a detailed step. The at?(.)-actions ask for state information relevant for deciding a trap has been entered. As these global processes are participant roles in the protocol, the emp(.) actions have been put in place as well. The emp(.)-actions correspond to a phase change, so they synchronize with

a particular conductor step and/or with other clients. Observe that for every global state–phase in $\mathsf{Client}_i(\mathsf{CS})$ there are more processes defined, one for each trap of that phase. A phase is always entered via its triv trap. For instance, after leaving phase $\mathsf{Without}_i$ the global process ends as process $\mathsf{Interrupt}_i[\mathsf{triv}]$. While the process resides in triv trap, it exchanges state information with the detailed process and updates its behaviour accordingly. For instance, the summand $\mathtt{at}?(\mathsf{Out}_i) \cdot \mathsf{Interrupt}_i[\mathsf{notYet}]$ in the specification of process $\mathsf{Interrupt}_i[\mathsf{triv}]$.

$$
\begin{array}{rcl}
\widehat{\mathsf{Client}_i}(\mathsf{CS}) & = & \mathsf{Without}_i[\mathsf{triv}] \\
\mathsf{Without}_i[\mathsf{triv}] & = & \mathtt{ok}!(\mathsf{leave}_i) \cdot \mathsf{Without}_i[\mathsf{triv}] + \mathtt{ok}!(\mathsf{enter}_i) \cdot \mathsf{Without}_i[\mathsf{triv}] + \\
 & & \mathtt{emp}(\mathsf{triv}_i) \cdot \mathsf{Interrupt}_i[\mathsf{triv}] \\
\mathsf{Interrupt}_i[\mathsf{triv}] & = & \mathtt{at}?(\mathsf{AtDoor}_i) \cdot \mathsf{Interrupt}_i[\mathsf{notYet}] + \mathtt{at}?(\mathsf{Out}_i) \cdot \mathsf{Interrupt}_i[\mathsf{notYet}] + \\
 & & \mathtt{at}?(\mathsf{Waiting}_i) \cdot \mathsf{Interrupt}_i[\mathsf{request}] + \mathtt{ok}!(\mathsf{leave}_i) \cdot \mathsf{Interrupt}_i[\mathsf{triv}] \\
\mathsf{Interrupt}_i[\mathsf{notYet}] & = & \mathtt{ok}!(\mathsf{leave}_i) \cdot \mathsf{Interrupt}_i[\mathsf{notYet}] + \mathtt{emp}(\mathsf{notYet}_i) \cdot \mathsf{Without}_i[\mathsf{triv}] \\
\mathsf{Interrupt}_i[\mathsf{request}] & = & \mathtt{emp}(\mathsf{request}_i) \cdot \mathsf{With}_i[\mathsf{triv}] \\
\mathsf{With}_i[\mathsf{triv}] & = & \mathtt{at}?(\mathsf{AtDoor}_i) \cdot \mathsf{With}_i[\mathsf{done}] + \mathtt{ok}!(\mathsf{explain}_i) \cdot \mathsf{With}_i[\mathsf{triv}] + \\
 & & \mathtt{ok}!(\mathsf{thank}_i) \cdot \mathsf{With}_i[\mathsf{triv}] \\
\mathsf{With}_i[\mathsf{done}] & = & \mathtt{emp}(\mathsf{done}_i) \cdot \mathsf{Without}_i[\mathsf{triv}]
\end{array}
$$

As the STDs of the clients are the same for all three Paradigm example models, we use the same specifications of processes $\widehat{\mathsf{Client}_i}$ and $\widehat{\mathsf{Client}_i}(\mathsf{CS})$. In the sequel, we give the complete translation of the three models, by specifying the server processes $\widehat{\mathsf{NDetServer}}$ and $\widehat{\mathsf{RoRoServer}}$ for the first two examples, and by defining the corresponding communication functions '|' for all three examples. For the communication function in all three examples '|' we put $\mathtt{at}!(s) \mid \mathtt{at}?(s) = \tau$ and $\mathtt{ok}?(a) \mid \mathtt{ok}!(a) = \mathtt{ok}(a)$, for $s = \mathsf{Out}_i, \mathsf{Busy}_i, \mathsf{Waiting}_i, \mathsf{AtDoor}_i$, and $a = \mathsf{enter}_i, \mathsf{explain}_i, \mathsf{thank}_i, \mathsf{leave}_i$. Note, ACP allows for keeping the resulting action of a synchronization observable. We exploit this feature and define the synchronization actions $\mathtt{ok}(a)$ as observable, as they describe detailed steps taken by clients, e.g., observation of $\mathtt{ok}(\mathsf{enter}_i)$ specifies a service request made by Client_i. Detailed steps should be observable, if they are used later to express system properties. Contrarily, synchronization of $\mathtt{at}!(.)$ and $\mathtt{at}?(.)$ is only used to update the information of the current local state. The resulting synchronization actions are neither needed in any further specifications nor in computations. Therefore, we choose to turn them into unobservable actions.

The non-deterministic server process $\widehat{\mathsf{NDetServer}}$ is defined by the following specification. See also Figure 8.

$$
\begin{array}{rcl}
\widehat{\mathsf{NDetServer}} & = & \mathsf{Idle} \\
\mathsf{Idle} & = & \mathtt{man}(\mathsf{check}_1) \cdot \mathsf{NDChecking}_1 + \cdots + \mathtt{man}(\mathsf{check}_n) \cdot \mathsf{NDChecking}_n \\
\mathsf{NDChecking}_i & = & \mathtt{man}(\mathsf{permit}_i) \cdot \mathsf{NDHelping}_i + \mathtt{man}(\mathsf{refuse}_i) \cdot \mathsf{Idle} \\
\mathsf{NDHelping}_i & = & \mathtt{man}(\mathsf{continue}_i) \cdot \mathsf{Idle}
\end{array}
$$

Moreover, in the case of the protocol driven by $\widehat{\mathsf{NDetServer}}$, we assume

$$\begin{aligned}
\mathtt{man}(\mathsf{check}_i) \mid \mathtt{emp}(\mathsf{triv}_i) &= \mathsf{check}_i \\
\mathtt{man}(\mathsf{permit}_i) \mid \mathtt{emp}(\mathsf{request}_i) &= \mathsf{permit}_i \\
\mathtt{man}(\mathsf{refuse}_i) \mid \mathtt{emp}(\mathsf{notYet}_i) &= \mathsf{refuse}_i \\
\mathtt{man}(\mathsf{continue}_i) \mid \mathtt{emp}(\mathsf{done}_i) &= \mathsf{continue}_i
\end{aligned}$$

All actions in the set $H = \{\mathtt{man}, \mathtt{emp}, \mathtt{at}?, \mathtt{at}!, \mathtt{ok}?, \mathtt{ok}!\}$ will be blocked to enforce communication. Finally, the process for the collaboration of the nondeterministic server and the n clients is given by

$$\begin{aligned}
\mathsf{CSNDet} = \partial_H(\, &\widehat{\mathsf{Client}}_1 \parallel \widehat{\mathsf{Client}}_1(\mathsf{CS}) \parallel \ldots \\
&\ldots \parallel \widehat{\mathsf{Client}}_n \parallel \widehat{\mathsf{Client}}_n(\mathsf{CS}) \parallel \widehat{\mathsf{NDetServer}}\,)
\end{aligned} \tag{12}$$

The following lemma states that the process algebraic translation is indeed branching bisimilar to the Paradigm model, for the case of CS–NDetServer collaboration. Let STD(CS–NDetServer) denote the STD of the CS–NDetServer collaboration obtained by the operational semantics as defined in Section 3.1.

Lemma 1. *CSNDet* $\leftrightarrow\!\!\!\!\!\!\underline{}\;\; STD$(*CS–NDetServer*) *up to renaming* $\mathtt{ok}(t)$ *actions into* t *and* $\mathtt{emp}(p)$ *actions into* p.

Proof. Since branching bisimulation is a congruence with respect to parallel composition and by the definition of the operational semantics of Paradigm in Section 3.1, it is sufficient to show the two STDs are component-wise branching bisimilar. The relation to be investigated is between the client component in totality obtained from the detailed STD and the global STD of Client_i following the semantics of Paradigm, and the parallel composition of processes $\widehat{\mathsf{Client}}_i$ and $\widehat{\mathsf{Client}}_i(\mathsf{CS})$ with synchronization $\mathtt{at}!(.)|\mathtt{at}?(.)$ and $\mathtt{ok}!(.)|\mathtt{ok}?(.)$ only and with the actions $H = \{\mathtt{at}?, \mathtt{at}!, \mathtt{ok}?, \mathtt{ok}!\}$ blocked. Both STDs are given in Figure 15. It is easy to establish a branching bisimulation relation. □

For the round-robin case, the translations $\widehat{\mathsf{Client}}_i$ of the clients remain the same. The translation of the global STD, $\mathsf{Client}_i(\mathsf{CS})$, needs minor modification only. We simply adapt $\widehat{\mathsf{Client}}_1(\mathsf{CS})$. This is because this global STD will start in phase Interrupt. More specifically, its starting state is $\mathsf{Interrupt}[\mathsf{triv}]$. We put

$$\widehat{\mathsf{Client}}_1(\mathsf{CS}) = \mathsf{Interrupt}_1[\mathsf{triv}] \quad \text{and} \quad \widehat{\mathsf{Client}}_i(\mathsf{CS}) = \mathsf{Without}_i[\mathsf{triv}] \quad \text{for } i > 1$$

The RoRoServer itself is translated in the following specification (see Figure 11):

$$\begin{aligned}
\widehat{\mathsf{RoRoServer}} &= \mathsf{RRChecking}_1 \\
\mathsf{RRChecking}_i &= \mathtt{man}(\mathsf{grant}_i) \cdot \mathsf{RRHelping}_i + \mathtt{man}(\mathsf{pass}_i) \cdot \mathsf{RRChecking}_{i+1} \\
\mathsf{RRHelping}_i &= \mathtt{man}(\mathsf{proceed}_i) \cdot \mathsf{RRChecking}_{i+1}
\end{aligned}$$

The communication function for the round-robin protocol is defined as

$$\begin{aligned}
\mathtt{man}(\mathsf{grant}_i) \mid \mathtt{emp}(\mathsf{request}_i) &= \mathsf{grant}_i \\
\mathtt{man}(\mathsf{proceed}_i) \mid \mathtt{emp}(\mathsf{done}_i) \mid \mathtt{emp}(\mathsf{triv}_{i+1}) &= \mathsf{proceed}_i \\
\mathtt{man}(\mathsf{pass}_i) \mid \mathtt{emp}(\mathsf{notYet}_i) \mid \mathtt{emp}(\mathsf{triv}_{i+1}) &= \mathsf{pass}_i
\end{aligned}$$

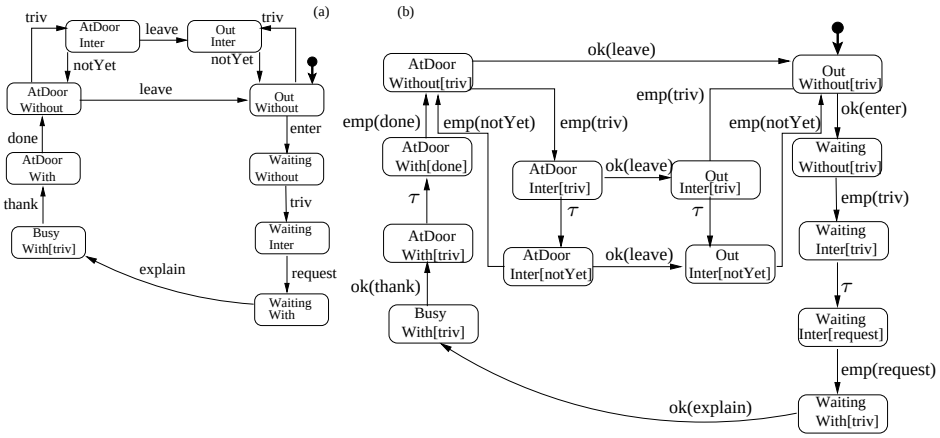

Figure 15: Branching bisimilar STDs a. Paradigm Client_i in totality; b. Process algebra Client_i in totality.

Again, for the set of blocked actions we have $H = \{\, \mathtt{man}, \mathtt{emp}, \mathtt{at}?, \mathtt{at}!, \mathtt{ok}?, \mathtt{ok}! \,\}$. The above results in the collaborative process for the round-robin protocol as defined by

$$\begin{aligned} \mathsf{CSRoRo} = \partial_H(\, &\widehat{\mathsf{Client}_1} \parallel \widehat{\mathsf{Client}_1}(\mathsf{CS}) \parallel \ldots \\ &\ldots \parallel \widehat{\mathsf{Client}_n} \parallel \widehat{\mathsf{Client}_n}(\mathsf{CS}) \parallel \widehat{\mathsf{RoRoServer}}\,) \end{aligned} \tag{13}$$

For the last choreography solution $\mathsf{CS-RoRoChoreography}$, only the communication is defined differently:

$$\begin{aligned} \mathtt{emp}(\mathsf{done}_i) \,|\, \mathtt{emp}(\mathsf{triv}_{i+1}) &= \mathsf{done}_i \\ \mathtt{emp}(\mathsf{notYet}_i) \,|\, \mathtt{emp}(\mathsf{triv}_{i+1}) &= \mathsf{notYet}_i \end{aligned}$$

The set of blocked actions is $H = \{\, \mathtt{emp}, \mathtt{at}?, \mathtt{at}!, \mathtt{ok}?, \mathtt{ok}! \,\}$. The above results in the collaborative process for the choreography protocol as defined by

$$\begin{aligned} \mathsf{CSRoRoChor} = \partial_H(\, &\widehat{\mathsf{Client}_1} \parallel \widehat{\mathsf{Client}_1}(\mathsf{CS}) \parallel \ldots \\ &\ldots \parallel \widehat{\mathsf{Client}_n} \parallel \widehat{\mathsf{Client}_n}(\mathsf{CS})\,) \end{aligned} \tag{14}$$

The following results can be easily proved. It states that STDs of the collaborations CS–RoRoServer, denoted $STD(\mathsf{CS\text{–}RoRoServer})$, obtained by the operational semantics as defined in Section 3.1 and its process algebraic translation CSRoRo are branching bisimilar. And similar for the $STD(\mathsf{CS\text{–}RoRoChoreography})$ collaboration and process CSRoRoChor.

Lemma 2.

(i) $\mathsf{CSRoRo} \leftrightarrows STD(\mathsf{CS\text{–}RoRoServer})$ *and*

(ii) $\mathsf{CSRoRoChor} \leftrightarrows STD(\mathsf{CS\text{–}RoRoChoreography})$

up to renaming $\mathtt{ok}(t)$ *actions into* t *and* $\mathtt{emp}(p)$ *actions into* p. □

4.3. Checking properties of the client-server systems

Having represented the three solutions to the critical section problem in ACP, the next step is to move to mCRL2 and to illustrate the model checking of a number of system properties expressed in the μ-calculus [10], the logical language for mCRL2. Translation of ACP-based specifications into the input language of the mCRL2 toolset of the n clients $\widehat{\mathsf{Client}}_i$, the global $\widehat{\mathsf{Client}}_i(\mathsf{CS})$ and the servers $\widehat{\mathsf{NDetServer}}$ and $\widehat{\mathsf{RoRoServer}}$ is straightforward[2] largely.

We consider the following properties for the three protocols, for clients $\widehat{\mathsf{Client}}_i$ and $\widehat{\mathsf{Client}}_j$ with $i \neq j$:

1. At any moment in time at most one client will be given service. In other words, never two (or more) clients, will be in the critical section at the same time. This is expressed as

   $$\texttt{[true*.ok(explain}_i\texttt{).(!ok(thank}_i\texttt{))*.ok(explain}_j\texttt{)] false}$$

   A sequence of actions in which $\mathsf{ok}(\mathsf{explain}_i)$ appears and at some later point $\mathsf{ok}(\mathsf{explain}_j)$ while no action $\mathsf{ok}(\mathsf{thank}_i)$ appears in between is impossible. Clearly, we use the detailed steps $\mathsf{ok}(\mathsf{explain}_i)$ and $\mathsf{ok}(\mathsf{thank}_i)$ to detect entering and leaving the critical section.

2. Two clients may request access to the critical section at the same time. In other words, more than one client can be in state Waiting in the detailed STD. Again, relying on the detailed STD, we can express this property by the following μ-calculus formula:

   $$\texttt{< true*.ok(enter}_i\texttt{).(!ok(thank}_i\texttt{))*.ok(enter}_j\texttt{) > true}$$

   There exists a sequence of actions in which an occurrence of $\mathsf{ok}(\mathsf{enter}_i)$ is followed, not necessarily immediately, by $\mathsf{ok}(\mathsf{enter}_j)$ before any occurrence of action $\mathsf{ok}(\mathsf{thank}_i)$.

3. Under the usual strong fairness assumption, every client who requested a service, eventually gets served, i.e. enters the critical section. The corresponding formula is:

   $$\texttt{[true*.ok(enter}_i\texttt{).(!ok(explain}_i\texttt{))*]< true*.ok(explain}_i\texttt{) > true}$$

   Once a client requires service, she will eventually be granted access, under the fairness assumption that any action that is enabled infinitely often will eventually be taken.

4. Every client who requested service will be eventually served, without further assumption. It is expressed by the recursive formula

   $$\texttt{[true* . ok(enter}_i\texttt{)] mu X . [!ok(explain}_i\texttt{)] X}$$

[2] Available on http://www.win.tue.nl/~andova

Note, the least fixed point of `mu X.[!ok(explain`$_i$`)] X` is the set of all states that have to make the `ok(explain`$_i$`)` step, possibly preceded by a finite number of steps other than `ok(explain`$_i$`)`. Thus, the formula reads as, once the `ok(enter`$_i$`)` step is made, eventually the `ok(explain`$_i$`)` step will be taken as well.

As expected, the first three properties are valid for the non-deterministic server, while the fourth one is not, because the $\widehat{\mathsf{NDetServer}}$ does not guarantee general access to the critical section. This is clear from the specification, as the $\widehat{\mathsf{NDetServer}}$ can always ignore a client, even if it has requested a service. On the other hand, all four properties are valid for the other two solutions based on the round-robin policy.

5. Specifying Paradigm models in Process Algebra

Based on the three example translations presented above, we proceed by formulating how to express a general Paradigm model in ACP. For clarity, we restrict to the hierarchical case where a component in a collaboration is either a conductor or a participant [22]. However, participants are allowed to have multiple roles addressing different conductors. Furthermore, for ease of presentation, we assume action-determinism, i.e. any two different transitions have different actions. This way, a transition is identified by its label.

Participants synchronize their detailed behavior with the global behavior, while the global behavior is governed by the consistency rules. The behavior of a conductor is connected to that of the participants by means of the consistency rules as well. The process algebraic translation of a participant, as seen in the example of Client in section 4.2, contains "informing" and "performing" elements. The "informing" part, informing about the detailed STD state towards one performing role, is modeled by the action `at`(.). It keeps the participant process unchanged. In addition, "performing" a local step by a participant is modeled by the action `ok`(.) with relevant argument.

After a role process has been informed about the current state of the detailed STD and has concluded that a new trap has been entered (without changing the current phase yet), it stores this information as it were, by making a step to the next corresponding global state: with the same phase as before, but with the new trap replacing the trap entered earlier.

A participant $Z = \langle \mathtt{ST}, \mathtt{AC}, \mathtt{TR} \rangle$ in Paradigm has a number of roles, R_1 to R_n say. For each detailed state s, we define a recursive equation in ACP as follows:

$$Z_s \quad = \quad \mathtt{at}!(s).Z_s + \sum_{s \xrightarrow{a} s' \in \mathtt{TR}} \mathtt{ok}?(a).Z_{s'}$$

Thus, Z_s can convey state information and can query the eligibility of any transition $s \xrightarrow{a} s'$ of the detailed STD for s and continue as the process $Z_{s'}$ corresponding to the target state s'. For a role $R_i = \langle \mathtt{ST}_i, \mathtt{AC}_i, \mathtt{TS}_i \rangle$ of the participant Z,

we define a system of recursive equations $p[t]$, for phase $p = \langle st_p, ac_p, tr_p \rangle \in \mathtt{ST}_i$, indexed by the traps t in the phase p, for $t \in \mathtt{AC}_i$.

$$\begin{aligned} p[t] \quad = \quad & \textstyle\sum_{s \in t} \sum_{s \in t' \neq t} \ \mathtt{at}?(s).p[t'] + \sum_{s \xrightarrow{a} s' \in tr_p} \ \sum_i \ \mathtt{ok}!_i(a).p[t] + \\ & \textstyle\sum \{ \ \mathsf{cr}_\gamma(\mathsf{man}, t_1, \ldots, t, \ldots, t_m).p'[\mathsf{triv}] \mid \\ & \qquad \gamma : \ c \xrightarrow{\mathsf{man}} c' \ * \ p_1 \xrightarrow{t_1} p'_1, \ \ldots, \ p \xrightarrow{t} p', \ \ldots, \ p_m \xrightarrow{t_m} p'_m \ \} \end{aligned}$$

The equation expresses that the process $p[t]$ is willing to update the trap information upon synchronizing its $\mathtt{at}?(s)$ action with the complementary $\mathtt{at}!(s)$ action of the detailed participant process, for any state s of the current trap t. The process $p[t]$ subsequently may evolve into any process $p[t']$, as they both share local state s. Furthermore, $p[t]$ allows, by offering synchronization on $\mathtt{ok}!_i(a)$, each transition $s \xrightarrow{a} s'$ present in phase p. The one-one correspondence of $\mathtt{ok}!_i(a)$ and $s \xrightarrow{a} s'$ relies on our assumption of action nondeterminism. The third group of synchronizations offered by $p[t]$ relates to the consistency rules. For any consistency rule γ involving a phase transition $p \xrightarrow{t} p'$ in role R_i from phase p in trap t, process $p[t]$ contributes transition $\mathsf{cr}_\gamma(\ldots, t, \ldots)$ as a synchronization option – in Section 3.1 called: contribution to collaboration. After synchronization $p[t]$ continues as $p'[\mathsf{triv}]$, representing phase p' in the trivial trap, as no specific state information is available (yet).

Thus, the detailed STD is willing to communicate its state information to every role; a role only synchronizes via $\mathtt{at}?(s)$ if s is a shared state of a trap t' and the current trap t. The communication function satisfies $\mathtt{at}!(s) \mid \mathtt{at}?(s) = \tau$ for $s \in \mathtt{ST}$. A transition $s \xrightarrow{a} s'$ in the detailed STD can only be made if allowed by all the roles. Therefore, the single $\mathtt{ok}?(a)$ of Z_s should be matched by all $\mathtt{ok}!_i(a)$ in any of the processes $p[t]$, t a trap of p, p a phase of R_i, $1 \leq i \leq n$. In this case, synchronization is amongst $n+1$ parties, the detailed STD and its n roles. We put $\mathtt{ok}?(a) \mid \mathtt{ok}!_1(a) \mid \cdots \mid \mathtt{ok}!_n(a) = \mathtt{ok}(a)$. The last part of the equation for the phase/trap process $p[t]$, chooses from all consistency rules γ that involve the trap t of the phase p. Synchronization and communication function for this are discussed below.

The process expression for a conductor Z is simpler, as we have assumed that no roles are defined in the Paradigm model for the conductor Z. However, all relevant transitions made by the conductor should match with a consistency rule for the particular collaboration. For a state $s \in \mathtt{ST}$ of Z, we now have the recursive equation

$$Z_s \quad = \quad \textstyle\sum \{ \ \mathsf{cr}_\gamma(a, t_1, \ldots, t_m).Z_{s'} \mid \gamma \colon s \xrightarrow{a} s' \ * \ p_1 \xrightarrow{t_1} p'_1, \ \ldots, \ p_m \xrightarrow{t_m} p'_m \ \}$$

In addition, for the irrelevant (non-conducting) conductor steps $s \xrightarrow{a} s'$ we have $Z_s = a.Z_{s'}$. So, in the collaboration, apart from the conductor Z, m participants are involved. For a consistency rule γ to apply, Z must have reached state s, while the participants must have reached the traps $t_1, \ldots, t_m$, respectively. Therefore, for the communication function we require that $m+1$ copies of the same communication action synchronize, one for the conductor and m for

the participants, by putting

$$\mathsf{cr}_\gamma(a,t_1,\ldots,t_m) \mid \mathsf{cr}_\gamma(a,t_1,\ldots,t_m) \mid \cdots \mid \mathsf{cr}_\gamma(a,t_1,\ldots,t_m) = \mathsf{cr}_\gamma(a).$$

If we are interested only in a component specification, and the way the vertical consistency restricts its dynamics, but the synchronization with other components is not considered (yet), the recursive specification of $p[t]$, being relevant for role R_i, is simplified by replacing the last summand of the specification in the following way:

$$\begin{array}{rcl} p[t] & = & \sum_{s\in t}\sum_{s\in t'\neq t} \ \mathtt{at}?(s).p[t'] + \sum_{s \xrightarrow{a} s'\in tr_p} \ \sum_i \ \mathtt{ok}!_i(a).p[t] + \\ & & \sum_{p':p \xrightarrow{t} p'\in \mathtt{TS}_i} t.p'[\mathsf{triv}] \end{array}$$

This allows us to establish a component-wise relation between a Paradigm model of a single component and its specification in process algebra, later to be easily extended to a relation between the Paradigm model of the overall collaboration and its process algebraic translation. The following lemma captures the equivalence between the Paradigm model and the process algebra specification of a single component.

Lemma 3. *Let $Z = \langle \mathtt{ST}, \mathtt{AC}, \mathtt{TR}\rangle$ be a participant with an initial state* init *and with roles R_1 to R_n. Role R_i, for $1 \leq i \leq n$, has phases p_{ij}, $1 \leq j \leq n_i$ such that $p_{ij} = \langle st_{ij}, ac_{ij}, tr_{ij}\rangle$ where $st_{ij} \subseteq ST_i$, $ac_i \subseteq AC_i$ and $tr_{ij} \subseteq TR_i$ of which p_{ij_0} is its initial state. Let $STD(Z, R_1, \ldots, R_n)$ be the STD of the participant component Z obtained by the operational semantics in Section 3.1. Let*

$$\widehat{Z} = \tau_{\mathtt{at}(s)} \circ \partial_H \ (\widehat{Z_{init}} \parallel \widehat{R_1} \parallel \ldots \parallel \widehat{R_n})$$

where $\widehat{R_i} = p_{ij_i}[\mathsf{triv}]$. Then $STD(Z, R_1, \ldots, R_n) \leftrightarrows \widehat{Z}$.

Proof. To simplify notation, by $\langle s, s_1[t_1], \ldots, s_n[t_n]\rangle$ we denote a state in the state space of $\widehat{Z}$ for s a state of Z, s_i a phase of R_i and t_i a trap of s_i. We establish the following relation $\mathcal{R}$ between the states of $STD(Z, R_1, \ldots, R_n)$ and the states of $\widehat{Z}$:

$$\langle s, s_1, \ldots, s_n\rangle \mathcal{R} \langle s', s_1[t_1], \ldots, s_n[t_n]\rangle \quad \textit{iff} \quad s = s' \wedge \forall i\colon s \in s_i[t_i]$$

It says that two states are in relation if their current local states are the same and this local state belongs to all current traps t_i of the current phases s_i of R_i. Next, we have to show that these two states (each of them in a separate STD) can mimic each other in making transition, in a way defined on the definition on page 26. To simplify the notation we reduce the number of roles to 1. We reason as follows: by first assuming that $\langle s, p\rangle \xrightarrow{a} \langle s', p'\rangle$, we study all cases how these transition could have occurred following the operation semantics, and we find a transition of $\langle s, p[t]\rangle$ that matches it. Then we prove the opposite direction, by matching every transition of $\langle s, p[t]\rangle$ to a transition of $\langle s, p\rangle$. Below, we assume that $\langle s, p\rangle \mathcal{R} \langle s, p[t]\rangle$ for $s \in \mathtt{ST}$, p a phase of a role R and t a trap of p. Note, according to the definition of $\mathcal{R}$, $s \in t$.
($\Rightarrow$): Assume that $\langle s, p\rangle \xrightarrow{a} \langle s', p'\rangle$ for some phases $p = \langle st, ac, tr\rangle$ and $p' = \langle st', ac', tr'\rangle$. According to the operational semantics there are two possible cases:

local transition: If a is a local transition, namely $a \in \mathtt{AC}$ of Z, then $s \xrightarrow{a} s' \in \mathtt{TR}$ of Z and moreover $s \xrightarrow{a} s' \in tr$. Then, according to the definition of Z_s and $p[t]$ we have

$Z_s \xrightarrow{\mathtt{ok?}(a)} Z_{s'}$ and $p[t] \xrightarrow{\mathtt{ok!}(a)} p[t]$. Therefore, $\langle s, p[t]\rangle \xrightarrow{\mathtt{ok}(a)} \langle s', p[t]\rangle$. As p is a phase that contain transition $s \xrightarrow{a} s'$, t is a trap of p such that $s \in t$ and therefore $s' \in t$. Thus, we conclude, $\langle s', p\rangle \mathcal{R} \langle s', p[t]\rangle$.

phase transfer: If a is a phase transfer, then $p \xrightarrow{a} p'$ is a global transition of role R, a is a connecting trap of phase p and phase p' and $s \in a$. Moreover, $s = s'$ and $s \in p'$. Consider the state $\langle s, p[t]\rangle$, for which we know that $s \in t$. There are two possibilities: Case $t = a$. In this case we obtain $\langle s, p[t]\rangle \xrightarrow{a} \langle s, p'[\mathsf{triv}]\rangle$ directly from the definition of $p[t]$. Furthermore, as $s \in p'$ it also holds $s \in p'[\mathsf{triv}]$, and therefore, $\langle s, p'\rangle \mathcal{R} \langle s, p'[\mathsf{triv}]\rangle$. Case $t \neq a$. In this case we observe the following sequence of transitions is possible from the state $\langle s, p[t]\rangle$: $\langle s, p[t]\rangle \xrightarrow{\tau} \langle s, p[a]\rangle \xrightarrow{a} \langle s, p'[\mathsf{triv}]\rangle$ where the τ action is due to hiding $\mathtt{at}(s)$ action. We conclude that: $\langle s, p\rangle \mathcal{R} \langle s, p[a]\rangle$ (since $s \in a$), and $\langle s, p'\rangle \mathcal{R} \langle s, p'[\mathsf{triv}]\rangle$, for the same reasons as in the previous case.
($\Leftarrow$): Assume that $\langle s, p[t]\rangle \xrightarrow{a} \langle s', p'[t']\rangle$. According to the definition of $\widehat{Z}$, this transition is part of, we distinguish three cases.

$a = \tau$: In this case τ is the result of synchronization on $\mathtt{at?}(s)$ and $\mathtt{at!}(s)$ actions. Then, $s' = s$, $p = p'$ and t' is again a trap of p which contains s. Therefore, $\langle s, p\rangle \mathcal{R} \langle s, p[t']\rangle$ as well.

local transition: In this case, $a = \mathtt{ok}(e)$, thus a is the result of synchronization of $\mathtt{ok?}(e)$ and $\mathtt{ok!}(e)$ actions, for some local transition $e \in \mathtt{AC}$. According to the specifications of Z_s and $p[t]$ we have that $Z_s \xrightarrow{\mathtt{ok?}(e)} Z_{s'}$ and $p[t] \xrightarrow{\mathtt{ok!}(e)} p[t]$, and therefore $p = p'$ and $t = t'$. Moreover, this implies that $s \xrightarrow{e} s' \in \mathtt{TR}$ and also $s \xrightarrow{e} s' \in tr_p$ (meaning phase p allows this local transition). Hence, by the operational semantics, $\langle s, p\rangle \xrightarrow{e} \langle s', p\rangle$. We finally conclude that $\langle s', p\rangle \mathcal{R} \langle s', p[t]\rangle$ since t is a trap of phase p which contains s and therefore it must contain s' as well.

phase transfer: In this case $t = a$ and $\langle s, p[t]\rangle \xrightarrow{t} \langle s, p'[\mathsf{triv}]\rangle$. Hence, t is a connecting trap from p to p', thus $s \in p$ and $s \in p'$. By the operational semantics we obtain $\langle s, p\rangle \xrightarrow{t} \langle s, p'\rangle$, and $\langle s, p'\rangle \mathcal{R} \langle s, p'[\mathsf{triv}]\rangle$ since $s \in \mathsf{triv}$ in phase p'.
The overall conclusion is that relation $\mathcal{R}$ is a branching bisimulation relating the initial states of $STD(Z, R_1, \ldots, R_n)$ and $\widehat{Z}$. Therefore these two systems are branching bisimilar. □

The following theorem extends the result of the previous lemma to the case of overall collaboration. We show that indeed for a collaboration Coll, its two representations, one being $STD(\mathsf{Coll})$, the STD of Coll induced by the operational semantics as defined in Section 3.1, and the other one being $\widehat{\mathsf{Coll}}$, the process algebra translation of Coll, are branching bisimilar.

Theorem 4. *For any Paradigm model Coll of a collaboration:* $STD(\mathsf{Coll}) \leftrightarrow \widehat{\mathsf{Coll}}$.

Proof. By Lemma 3 we establish branching bisimilarity of the Paradigm model and the process algebra translation of each component of the collaboration. Next, we take the consistency rules and the corresponding synchronization function into account. The congruence properties of branching bisimulation with respect to the parallel composition operator of ACP allows us to lift the equivalence from the components to their composition. □

From the above outline of the general translation one can see how more complicated Paradigm models can be dealt with. In case of more than one conductor per protocol there is always at most one conductor per protocol step. The cr action selected is synchronized with the relevant combination of phase transfers from different processes for different roles, possibly driven differently.

6. Conclusion

The paper addresses the following issues. It starts by introducing Paradigm based on formal descriptions. It is pointed out, Paradigm is a language for specifying coordination models for collaborations between components. Most characteristic for Paradigm are its dynamic constraints, phases and traps. They can be dynamically composed into roles and, via the roles, into protocols. The language Paradigm is transition-based: any Paradigm model consists of STDs, both for components and for the roles they have (as participant); protocols are specified step-wise too, per protocol step possibly conducted by a component (as conductor). Within a Paradigm model, vertical dynamic consistency is maintained syntactically; horizontal dynamic consistency is modeled through protocols, constituting glue.

The paper also presents complete operational semantics for Paradigm, carefully redefining constraining effects of phases and traps in terms of step dependencies across dimensions of a Cartesian product space. Given the operational semantics, the paper positions Paradigm, relating it to the framework [14], to Wright [1] and to different types of coordination languages. In addition, on the basis of the operational semantics, a systematic ACP translation of a large class of Paradigm models is given. The models covered are those, where components are either participants or conductors, not both. Participants may have multiple roles, though. Not yet covered by the systematic translation, are the Paradigm models where at least one component not only has a role in some protocol, but also is conductor of a protocol. For models covered, the paper establishes branching bisimilarity between the Cartesian product STD of the Paradigm model and the process algebra STD of the translated model (Theorem 4).

The paper moreover presents some example models, thus illustrating language and translation. Some verification results established through the `mCRL2` toolset have been given too. We could have added other, more interesting example models or example models with more flexibility, but that would have made the long paper even longer.

As future work we want to address the general translation of any Paradigm model into ACP. Some results in that direction have been achieved already [4, 3], but translations are as yet case-driven. Future work also is going to address integrated tooling for graphical editing, visual animation, ACP translation and `mCRL2` verification of general Paradigm models. See [35, 34] for first results.

In the introduction we referred to our translation of Paradigm into ACP, as a first step towards a formal underpinning of originally unforeseen changes of systems. Thus, certainly not the least part of our future work is going to address such changes, particularly changes established through self-adaptation. As has been recently explained in [3], see also [23, 2, 24], a Paradigm model (for a regular, foreseen coordination solution) which moreover contains a special component McPal, can coordinate its own on-the-fly migration towards an originally unforeseen way of working. In general, migration coordination should account for a variety of migration trajectories, even per component. It is for such forms of self-adaptation, controlled by the model, we want to integrate

formal analysis of coordinating migration trajectories into our investigations. Topics of interest are for instance, migration patterns, temporary and gradual relaxation of consistency requirements during migration, cooperation between several McPals, alignment and co-evolution of systems. In this manner we expect to gain substantial insight into quite different forms of change and of flexible control thereof.

Acknowledgement. We would like to thank Rob Nederpelt as well as the three anonymous reviewers for their most valuable comments.

References

[1] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6:213–249, 1997.

[2] S. Andova, L. Groenewegen, and E. de Vink. System evolution by migration coordination. In A. Serebrenik, editor, *Proc. BENEVOL 2008*, pages 18–22, Eindhoven, 2008.

[3] S. Andova, L.P.J. Groenewegen, J. Stafleu, and E.P. de Vink. Formalizing adaptation on-the-fly. In G. Salaün and M. Sirjani, editors, *Proc. FOCLASA'09*. ENTCS, to appear. 20pp.

[4] S. Andova, L.P.J. Groenewegen, J.H.S. Verschuren, and E.P. de Vink. Architecting security with Paradigm. In R. de Lemos, J.-C. Fabre, C. Gacek, F. Gadducci, and M.H. ter Beek, editors, *Architecting Dependable Systems VI*, pages 255–283. LNCS 5835, 2009.

[5] F. Arbab. The IWIM model for coordination of concurrent activities. In P. Ciancarini and C. Hankin, editors, *Proc. COORDINATION 1996*, pages 34–56. LNCS 1061, 1996.

[6] F. Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14:329–366, 2004.

[7] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Number 18 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

[8] M.H. ter Beek, C.A. Ellis, J. Kleijn, and G. Rozenberg. Team automata for spatial access control. In W. Prinz, M. Jarke, Y. Rogers, K. Schmidt, and V. Wulf, editors, *Proc. ECSCW'01*, pages 59–77. Kluwer, 2001.

[9] J.A. Bergstra, A. Ponse, and S.A. Smolka. *Handbook of Process Algebra*. Elsevier, 2001.

[10] J. Bradfield and C. Stirling. Modal mu-calculi. In *Handbook of Modal Logic*, pages 721–756. Elsevier, 2007.

[11] M. Broy, I.H. Krüger, and M. Meisinger. A formal model of services. *ACM Transactions on Software Engineering and Methodology*, 16, 2007.

[12] M. Broy and K. Stoelen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer, 2001.

[13] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32:444–458, 1989.

[14] P. Ciancarini. Coordination models and languages as software integrators. *ACM Computing Surveys*, 28:300–302, 1996.

[15] R. De Nicola and F.W. Vaandrager. Three logics for branching bisimulation. *Journal of the ACM*, 42:458–487, 1995.

[16] 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 (Chapter 4)*, pages 71–102. Research Study Press, Taunton, 1994.

[17] 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 (Chapter 12)*, pages 307–331. MIT Press, 2000.

[18] R.J. van Glabbeek. The linear time–branching time spectrum II: the semantics of sequential systems with silent moves. In E. Best, editor, *Proc. CONCUR'93*, pages 66–81. LNCS 715, 1993.

[19] R.J. van Glabbeek. The linear time – branching time spectrum I; the semantics of concrete, sequential processes. In *Handbook of Process Algebra*, chapter 1, pages 3–99. Elsevier, 2001.

[20] R.J. van Glabbeek and P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43:555–600, 1996.

[21] L. Groenewegen, N. van Kampenhout, and E. de Vink. Delegation modeling with Paradigm. In J.-M. Jacquet and G.P. Picco, editors, *Proc. COORDINATION 2005*, pages 94–108. LNCS 3454, 2005.

[22] 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.

[23] L. Groenewegen and E. de Vink. Evolution-on-the-fly with Paradigm. In P. Ciancarini and H. Wiklicky, editors, *Proc. COORDINATION 2006*, pages 97–112. LNCS 4038, 2006.

[24] L.P.J. Groenewegen and E.P. de Vink. Dynamic system adaptation by constraint orchestration. Technical Report CSR 08/29, Technische Universiteit Eindhoven, 2008. 20pp, arXiv:0811.3492v1.

[25] J.F. Groote, A.H.J. Mathijssen, M.A. Reniers, Y.S. Usenko, and M.J. van Weerdenburg. The formal specification language mCRL2. In E. Brinksma, D. Harel, A. Mader, P. Stevens, and R. Wieringa, editors, *Methods for Modelling Software Systems*. IBFI, Schloss Dagstuhl, 2007. 34 pages.

[26] J.F. Groote and M.A. Reniers. *Handbook of Process Algebra*, chapter 17, Algebraic Process Verification, pages 1151–1208. Elsevier, 2001.

[27] N. Kokash, C. Koehler, and E.P. de Vink. Data-aware design and verification of service composition with Reo and mCRL2. In *Proc. SAC 2010, Sierre, March 21–26, 2010*. ACM, 2010. To appear.

[28] R. Milner. Operational and algebraic semantics of concurrent processes. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 1201–1242. Elsevier/The MIT Press, 1990.

[29] M. Möller, E.-R. Olderog, H. Rasch, and H. Wehrheim. Integrating a formal method into a software engineering process with UML and Java. *Formal Aspects of Computing*, 20:161–204, 2008.

[30] M.Á. Pérez-Toledano, A.N. Martínez, J.M. Murillo, and C. Canal. A safe dynamic adaptation framework for aspect-oriented software development. *J. UCS*, 14(13):2212–2238, 2008.

[31] W. Reisig. *Petri Nets: an Introduction*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985.

[32] N.F. Rodrigues and L.S. Barbosa. Architectural prototyping: From CCS to .Net. In A. Mota and A.V. Moura, editors, *Proc. SBMF 2004*, pages 151–167. ENTCS 130, 2005.

[33] A.W. Roscoe. *The theory and practice of concurrency*. Prentice Hall, 1998.

[34] J. Stafleu. UML description of Paradigm. Master's thesis, LIACS, Leiden University, 2009. Technical Report 09–16.

[35] A.W. Stam. *Interaction Protocols in PARADIGM*. PhD thesis, LIACS, Leiden University, 2009. Forthcoming.

[36] M.R. Steen, L.P.J. Groenewegen, and G. Oosting. Parallel control processes: Modular parallelism and communication. In L.O. Hertzberger, editor, *Proc. Intelligent Autonomous Systems*, pages 562–579. North-Holland, 1987.