Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 229 (2009) 3–20
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Dynamic Consistency in Process Algebra: From Paradigm to ACP

S. Andova[1],[2]

Formal Methods Group
Technical University of Eindhoven, The Netherlands

L.P.J. Groenewegen

FaST Group, LIACS
Leiden University, The Netherlands

E.P. de Vink

Formal Methods Group
Technical University of Eindhoven, 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. To enable automated verification of Paradigm models, a translation of Paradigm into process algebra is provided. Examples are given and guidelines for a systematic translation into the process algebra ACP are discussed. Verification results building on the mCRL2 toolset are presented as well.

Keywords: automated verification, 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. [1,16,14,13]. In this paper, we link the coordination modeling language Paradigm via the process algebra ACP with the `mCRL2` toolset. This way, the flexibility of

[1] Corresponding author

[2] Email: `s.andova@tue.nl`

doi:10.1016/j.entcs.2009.06.026

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 a recursive specification of a system of parallel processes is presented. Central to Paradigm is the decomposition of dynamic constraints along two axes: (i) vertically, restrictions of a component with respect to the roles it fulfills in all collaborations it is engaged in, (ii) horizontally, coordination and synchronization of subbehaviour enforced upon participants in a collaboration.

The coordination modeling language Paradigm [8,9] specifies roles and interactions within collaborations between components. Interactions are 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 within a protocol of a component, 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 modelling evolution and self-adaptation [9].

Processes algebras (PA for short), such as CCS, CSP, LOTOS, ACP, 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 define the synchronization and hiding of actions (by renaming them into the so-called silent action τ). Characteristic for the process algebra ACP [2,4] is the user-defined communication function '$|$' of multi-actions, such that e.g. $\partial_{\{a,b,c\}}(a.P \parallel b.Q \parallel c.R)$ yields $d.(P \parallel Q \parallel R)$ if we have $a \mid b \mid c = d$. Various equivalences can be built upon process calculi. For ACP, strong bisimulation and branching bisimulation are the prominent ones.

In the present paper, the translation of Paradigm models into PA is first introduced by means of two examples. In both cases the system consists of n clients who try to get service from one server exclusively, a critical section problem. The difference is, that in one example the server is supposed to choose the next client in a non-deterministic manner, while in the other example this is done in a round-robin manner. For both examples, a translation into PA is given and a subsequent analysis with the mCRL2 toolset discussed.

While the translation of the Paradigm models into PA for both examples is done manually, the computation of the state spaces of the entire systems (including all interleaving and interaction) and their analysis exceed human capabilities as the number of clients increases. (As discussed in Section 4, for an n-client system this is $(5{\cdot}2^{n-2}-1){\cdot}n + 1$.) Exploiting of the toolset mCRL2, unleashes push-button techniques to generate the complete, symbolic state spaces, on which further analysis can be done. Several properties of functional correctness of the two examples have been 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 server 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. As a consequence, to be discussed in outline, a counterexample obtained by model-checking can be traced back in the original Paradigm model.

Related work. Our work on consistency in a horizontal and vertical dimension has been influenced by the work of Küster [6,12]. The Wright language [1] based on CSP provides FDR support to check static consistency properties. Next to consistency, which is built-in for Paradigm, the focus in Paradigm modelling is on dynamics and change of constraints. As for the Korrigan ADL [15], the formalism should be more than just a box-and-line notation. Managers of Paradigm collaborations are reminiscent to the orchestration connector of [3]. However, in Paradigm there is an explicit separation of the manager and its coordinating of the collaboration, yielding different 'slices' of abstraction.

Other bridges from software architecture to automated verification include the pipeline from UML via Rebeca and Promela to the SPIN model-checker and from UML via Object-Z and CSP to the FDR model-checker [17,13]. Process algebra driven prototyping as coordination from CCS is proposed in [16]. The skeletons generated from CCS-specifications overlap with Paradigm collaborations. In the TITAN framework [14], CCS is playing a unifying role in a heteorgeneous environment for aspect-oriented software engineering.

Structure of the paper. In Section 2 the basics of Paradigm is given together with two running examples. Section 3 briefly introduces process algebra. In Section 4, a detailed translation of the Paradigm models of the two protocols is given, followed by a general translation of Paradigm models in Section 5. Section 6 concludes the paper.

2 Paradigm and two critical section models

This section briefly describes the main notions of Paradigm. By means of two examples, that will be used as running examples through out the paper, the main aspects of the Paradigm approach are explained. We also introduce an UML-style architectural diagram for collaborating components, that sets the stage for various notions of dynamics. We point out where the Paradigm notions are relevant and how consistency of dynamics can be guaranteed.

The following notions constitute Paradigm's basic concepts: state-transition diagram, phase, (connecting) trap, partition and global process.

- A *state-transition diagram* or *STD* is a triple $\langle \mathtt{ST}, \mathtt{AC}, \mathtt{TS} \rangle$. Here, $\mathtt{ST}$ is the set of states, $\mathtt{AC}$ the set of actions and $\mathtt{TS} \subseteq \mathtt{ST} \times \mathtt{AC} \times \mathtt{ST}$ is the set of transitions. A transition $(x, a, x') \in \mathtt{TS}$ is said to be from state x to state x' and is denoted by $x \xrightarrow{a} x'$.
- A *subprocess* or *phase* of an STD $\langle \mathtt{ST}, \mathtt{AC}, \mathtt{TS} \rangle$ is an STD $\langle \mathtt{st}, \mathtt{ac}, \mathtt{ts} \rangle$ such that $\mathtt{st} \subseteq \mathtt{ST}$, $\mathtt{ac} \subseteq \mathtt{AC}$ and $\mathtt{ts} \subseteq \{ (x, a, x') \in \mathtt{TS} \mid x, x' \in \mathtt{st}, a \in \mathtt{ac} \}$.
- A *trap* t of a phase $S = \langle \mathtt{st}, \mathtt{ac}, \mathtt{ts} \rangle$ is a non-empty set of states $t \subseteq \mathtt{st}$ such that $x \in t$ and $x \xrightarrow{a} x' \in \mathtt{ts}$ imply that $x' \in t$. A trap *connects* the phase S it belongs to with another phase $S' = \langle \mathtt{st}', \mathtt{ac}', \mathtt{ts}' \rangle$ if $t \subseteq \mathtt{st}'$, notation $S \xrightarrow{t} S'$.
- A *partition* $\{ (S_i, T_i) \mid i \in I \}$ of an STD $\langle \mathtt{ST}, \mathtt{AC}, \mathtt{TS} \rangle$, I a finite index set, is a set of phases $S_i = \langle \mathtt{st}_i, \mathtt{ac}_i, \mathtt{ts}_i \rangle$, each with a set T_i of its traps.
- A *global process* or *global STD* at the level of a partition $\pi = \{ (S_i, T_i) \mid i \in I \}$ of an STD $Z = \langle \mathtt{ST}, \mathtt{AC}, \mathtt{TS} \rangle$ 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 changes. Z is called the detailed STD of the global STD $Z(\pi)$.

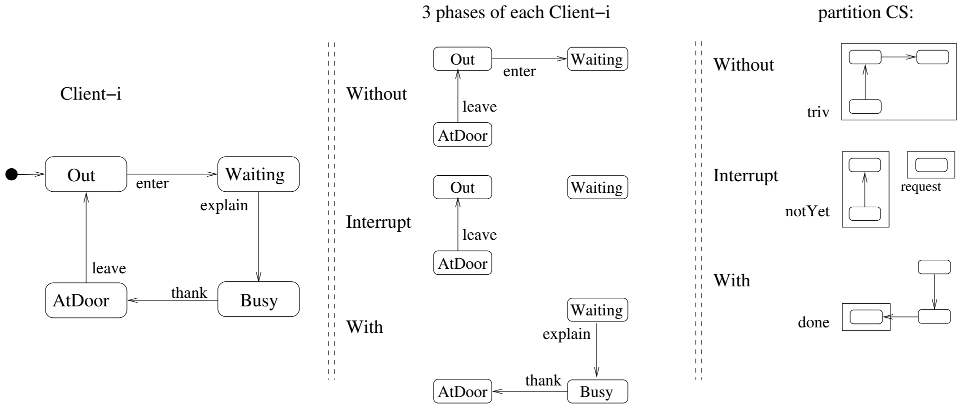

Fig. 1. (a) STD, (b) phase and (c) trap constraints and partition.

An STD is a step-wise description, from state to state, of the dynamics belonging to some component. Often it is visualized as a directed graph, where nodes are states and action-labeled edges are transitions. The initial state of an STD is graphically indicated by a black dot-and-arrow. Figure 1a gives an example STD Client_i in and around a shop. The entire system we consider later, contains n such clients with the same dynamics each (as well as other components).

The meaning of states and actions should be self-explanatory. The overall requirement is that only one client at a time, out of all n clients, is allowed to be in its state Busy. So, being in state Busy is a Critical Section problem (CS for later use). To solve it, Client_i dynamics is constrained by the phase prescribed. Figure 1b visualizes the phases Without, Interrupt and With. Phase Without excludes being in state Busy by prohibiting to take the (steps labeled with) actions explain and thank. Contrarily, phase With allows both, going to and leaving state Busy. Finally, phase Interrupt is an interrupted form of Without, as action enter cannot be taken, but being in state Waiting is allowed, though.

In view of a change of the current phase into a next phase to take place, enough progress within the current phase must have been made; a connecting trap has to be entered first. Figure 1c pictures relevant connecting traps for the above three phases, drawn as polygons around the states the trap consists of. In particular we need, trap triv to be connecting from Without to Interrupt, trap notYet to be connecting from Interrupt back to Without, trap request to be connecting from Interrupt forward to With and finally, trap done to be connecting from With back to Without. In this manner, Figure 1c gives all ingredients needed for the dynamics of a Client_i STD at the higher, more global level of these phases and their traps. The ingredients constitute a partition as well as a global STD $\mathsf{Client}_i(\mathsf{CS})$ – see Figure 2a. The two additional trivial traps, triv of Interrupt as well as triv of With, are not really needed for the higher level dynamics and are not drawn.

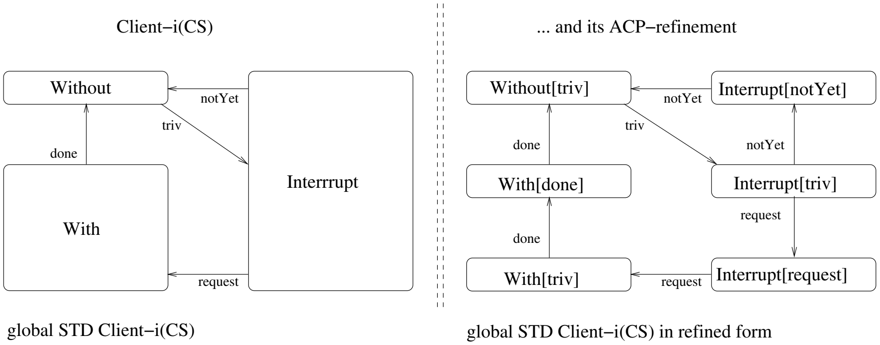

Fig. 2. (a) Global process and (b) refined global process.

Figure 2b presents a slightly refined diagram of the 'official' global STD. Here the state names keep track of the most recently entered trap within a current phase –as if it could be taken as a smaller phase committed to within the one currently imposed; action names similarly refer to the corresponding trap entering. As it will be pointed out below, the refined form of a global process serves its purpose for our PA translation.

So far, we have discussed the sequential composition of constraints: imposed phases alternated via traps committed to. Any current phase constrains the actions that can be selected to those belonging to the phase. As a consequence, at any moment, a current detailed state should belong to the current phase too. From this, consistent dynamics of the detailed STD and of the global STD follows.

Paradigm's consistency rules are to the essence of synchronizing composition. They specify the combination of detailed behaviour of a so-called *manager* and global behaviours of arbitrarily many so-called *employees*. Any consistency rule describes a simultaneous execution of a step of the manager, and a global step of each of the employees.

Figure 3 presents an overview of a Paradigm model as a liberal mixture of UML's object, composite structure and collaboration diagrams: a general collaboration at the left, the CS collaboration at the right. The elements called Client_i are employee

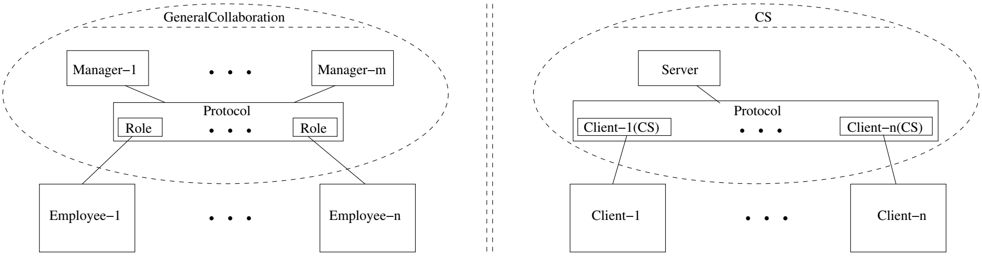

Fig. 3. Impression of Paradigm's architecture: (a) general, (b) CS example.

components of collaboration CS. As employees, they contribute their dynamics to the collaboration, not directly but via the global STDs $\mathsf{Client}_i(\mathsf{CS})$ serving as roles. The suitably synchronized roles together are the protocol, whose dynamics is driven by one or more managers, exactly one manager for each step. The consistency rules specify the protocol dynamics in a consistent manner: Paradigm's constraints express how single role dynamics and correspondingly single, detailed employee dynamics stay consistent. Syntactically, a consistency rule couples global steps from different global STDs, appearing at the right-hand side of the rule, as a list of phase changes, with the one detailed manager step at the left-hand side.

For our running examples, of n clients getting service, one at a time, we will consider two instantiations of the collaboration of a server with n clients, a non-deterministic and a round-robin one. The non-deterministic server checks the clients in an arbitrary order. If the client that is checked is ready to be served, it is served, i.e. permitted to the critical section. If not, the client is refused. Only after finishing the serving of the particular client, the server returns into the idle position, where it can select any client for inspection again.

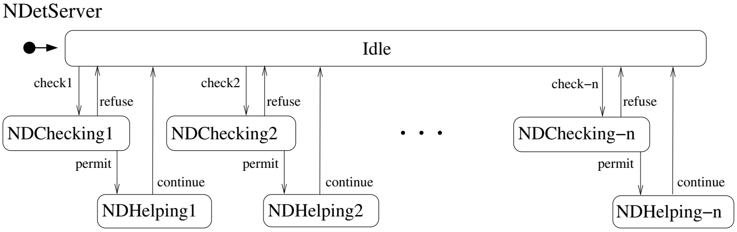

Fig. 4. STD non-deterministic manager NDetServer.

The STD of the nondeterministic server NDetServer comprises Figure 4. Being the manager, the detailed steps of NDetServer need to be coupled to phases of the Clients. This can be done as follows

$$\begin{array}{lrclclrcl}
\mathsf{NDetServer}: & \mathsf{Idle} & \overset{check_i}{\rightarrow} & \mathsf{NDChecking}_i & * & \mathsf{Client}_i(\mathsf{CS}): & \mathsf{Without} & \overset{triv}{\rightarrow} & \mathsf{Interrupt} \\
\mathsf{NDetServer}: & \mathsf{NDChecking}_i & \overset{refuse}{\rightarrow} & \mathsf{Idle} & * & \mathsf{Client}_i(\mathsf{CS}): & \mathsf{Interrupt} & \overset{notYet}{\rightarrow} & \mathsf{Without} \\
\mathsf{NDetServer}: & \mathsf{NDChecking}_i & \overset{permit}{\rightarrow} & \mathsf{NDHelping}_i & * & \mathsf{Client}_i(\mathsf{CS}): & \mathsf{Interrupt} & \overset{request}{\rightarrow} & \mathsf{With} \\
\mathsf{NDetServer}: & \mathsf{NDHelping}_i & \overset{continue}{\rightarrow} & \mathsf{Idle} & * & \mathsf{Client}_i(\mathsf{CS}): & \mathsf{With} & \overset{done}{\rightarrow} & \mathsf{Without}
\end{array}$$

where '$*$' separates a manager step from the global steps and $1 \leq i \leq n$. Note, for this protocol, a manager step of the server corresponds to a phase change of exactly one client. E.g., the server moves from the state Idle to $\mathsf{NDChecking}_i$ iff the global client process $\mathsf{Client}_i(\mathsf{CS})$ changes from the phase Without to the phase Interrupt. The server then makes a $\mathtt{check}_i$ transition. In general, there is a precondition, however. Within the phase Without sufficient progress should have been made, such that the trap, in this case the trivial trap triv, has been reached. Here, the requirement is vacuous as the trivial trap, consisting of all states of the detailed STD, is trivially reached. For the actual checking, the two middle consistency rules above, dependent on the trap, viz. notYet vs. request, the target of the manager transition and the next employee phase is decided, viz. state Idle and phase Without or state $\mathsf{NDHelping}_i$ and phase With, respectively.

The consistency rules maintain horizontal consistency, in the present example between server and the clients. The constraint that a detailed transition of a client needs to be allowed by the current phase and the relevant trap, enforces vertical consistency of the employee, on the one hand, and its abstraction within the collaboration, on the other hand.

As a variation to the above example, we next consider round-robin access to service. Now, the server checks the clients in turn for a request for being served. Clients wait, if necessary, for being checked. The STD of the server RoRoServer is given in Figure 5.

The server acts, in our set-up, as the manager of the partition CS of which each client is an employee via their global process $\mathsf{Client}_i(\mathsf{CS})$. The consistency rules that synchronize the detailed steps of the server and the global phases of the clients are the following (implicitly counting modulo n and $1 \leq i \leq n$):

$$\mathsf{RoRoServer}: \mathsf{RRChecking}_i \overset{\mathit{grant}}{\to} \mathsf{RRHelping}_i \ * \ \mathsf{Client}_i(\mathsf{CS}) : \mathsf{Interrupt} \overset{\mathit{request}}{\to} \mathsf{With}$$

$$\mathsf{RoRoServer}: \mathsf{RRHelping}_i \overset{\mathit{proceed}}{\to} \mathsf{RRChecking}_{i+1} \ *$$
$$\mathsf{Client}_i(\mathsf{CS}): \mathsf{With} \overset{\mathit{done}}{\to} \mathsf{Without}, \quad \mathsf{Client}_{i+1}(\mathsf{CS}) : \mathsf{Without} \overset{\mathit{triv}}{\to} \mathsf{Interrupt}$$

$$\mathsf{RoRoServer}: \mathsf{RRChecking}_i \overset{\mathit{pass}}{\to} \mathsf{RRChecking}_{i+1} \ *$$
$$\mathsf{Client}_i(\mathsf{CS}): \mathsf{Interrupt} \overset{\mathit{notYet}}{\to} \mathsf{Without}, \quad \mathsf{Client}_{i+1}(\mathsf{CS}): \mathsf{Without} \overset{\mathit{triv}}{\to} \mathsf{Interrupt}$$

Note the difference with the non-deterministic protocol. For instance, in the second consistency rule here, the synchronization between the server 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 leaving the critical section and Client_{i+1} being interrupted to be checked. In contrast, in the previous non-deterministic case, analogous coordination is splitted in two consistency rules, viz. first a return of the server to idling after helping client i, followed by a check of the next client.

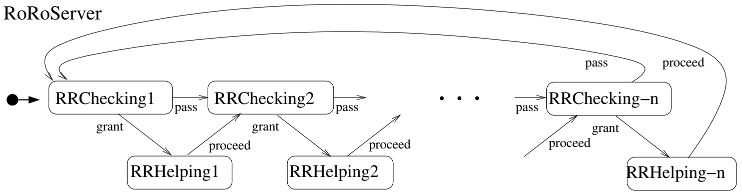

Fig. 5. STD round-robin manager RoRoServer.

3 Process algebra

Process Algebra (PA) is a formal method used for modeling and analyzing systems of concurrent processes. Basic ingredients of any PA are a set of operators, including constants, and a set of equations, also called axioms or laws, capturing dependencies between the operators. A PA is typically parameterized by a set of atomic actions A, the smallest activities that a process can perform. In a PA, the actions are considered as constants. Other basic operators common to most process algebras are sequential composition '$\cdot$', non-deterministic choice '$+$' and parallel composition '$\|$'. The process algebra ACP [2,4] we will consider here, has as additional operators, encapsulation '∂_H' and abstraction 'τ_I'. The encapsulation operator '∂_H' can be used to block actions from the set $H \subseteq A$, generally to avoid unmatched synchronization. The abstraction operator 'τ_I' is used to hide observable actions that are part of the set of actions I, i.e. any action in I is considered internal and is renamed into τ, the special action denoting internal activity.

Processes are described by algebraic expressions, and relations between processes by equalities. Infinite processes can be expressed by recursive specifications. For instance, a process Client_i, considered in isolation, is described by the following recursive specification:

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

In the specification, the index i appears as a parameter that ranges over $\{1, \ldots, n\}$. Note, process Client_i has sequential behaviour without any branching, therefore, non-deterministic choice does not occur in the description.

The behaviour of the parallel composition of two processes is obtained by interleaving the two separate behaviours. In addition, processes can communicate by synchronizing on certain actions. In ACP, action synchronization is user-defined by a so-called *communication function.* The communication function may involve two or more arguments, enabling multi-party synchronization. As we will discuss in Section 4, the synchronization by communication very naturally expresses the consistency rules in Paradigm. This will not come as a surprise, as they are both meant to express synchronization between some behaviours.

Higher level abstract behaviour is obtained by hiding activities, that are considered internal. These actions will subsequently be ignored in the further computations (to a large extent). For instance, the more abstract behaviour of process Client shown in Figure 2a. can easily be obtained from the more detailed behaviour of the process in Figure 2b, by abstracting away the following events: action done between states $\mathsf{With[triv]}$ and $\mathsf{With[done]}$, action notYet between $\mathsf{Interrupt[triv]}$ and $\mathsf{Interrupt[notYet]}$ and similar for the action $\mathsf{request}$. The abstract behaviour is obtained, first, by renaming these actions into the special action τ, and then manipulating the specification by means of some algebraic laws that allow removing of some τ events from the specification.

As elaborated below, process algebraic descriptions of Paradigm models help us to ‘observe’ the dynamics at a more detailed level. While in Paradigm, for instance, the mechanism by which one phase imposes constraints on the detailed STD dynamics is implicit, in process algebra this has to be explicitly specified. In fact it is defined as a synchronization of the two behaviours by which certain information is exchanged, in this particular case, an allowance or disallowance to make certain steps.

Once all components that build the Paradigm model are translated into process algebra, in principle, the complete behaviour can be computed. Furthermore, various verification techniques can be applied to the system obtained. A widely accepted technique for verifying properties of a system, which we use as well below, is model-checking. In short, once the model is obtained in the form of an STD, a modal formula can be checked against the 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 the system. Usually, the model that describes the complete system is very large and it is beyond human capabilities to reason about it without computer assistance. Fortunately, many efficient tools have been developed to support the analysis, in particular automated tools providing model-checking. For our experiments, we have used the toolset `mCRL2` [10,11]. [3] Its formal specification language `mCRL2` is based on ACP, together with facilities for abstract datatypes and the method of parameterized boolean equation systems (PBES) for symbolic model-checking purposes.

4 Two example models translated into ACP

In this section, the various STDs from the example Paradigm models introduced earlier, will be translated into ACP processes. In particular, each STD will be interpreted as a set of recursively defined processes, extended with specific communication details. For the communication of a detailed STD and the global STD, we use actions `ok`?(.) and `ok`!(.) that take the labels of detailed steps as their argument. The complementary actions synchronize if the step to be taken by the detailed STD is allowed by the current phase as constraint. In addition, we use the actions `at`?(.) and `at`!(.) to signal the current state from detailed STD to global STD. Upon synchronization of these actions, the global process will update its trap information. For the communication within the protocol, e.g. between the server and its clients, actions `man`(.) on the side of a manager are meant to complement `emp`(.) actions on the side of the employees. Synchronization leads to execution of the corresponding consistency rule: a local transition of the manager, phase changes for the employees involved.

For the two concrete examples this yields the following. We adorn the n processes

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

Client_i with state actions $\mathtt{at!}$ and transition actions $\mathtt{ok?}$.

$$\begin{array}{rcrcl}
\mathsf{Out}_i & = & \mathtt{at!}(\mathsf{Out}_i)\ .\ \mathsf{Out}_i & + & \mathtt{ok?}(\mathsf{enter}_i)\ .\ \mathsf{Waiting}_i \\
\mathsf{Waiting}_i & = & \mathtt{at!}(\mathsf{Waiting}_i)\ .\ \mathsf{Waiting}_i & + & \mathtt{ok?}(\mathsf{explain}_i)\ .\ \mathsf{Busy}_i \\
\mathsf{Busy}_i & = & \mathtt{ok?}(\mathsf{thank}_i)\ .\ \mathsf{AtDoor}_i & & \\
\mathsf{AtDoor}_i & = & \mathtt{at!}(\mathsf{AtDoor}_i)\ .\ \mathsf{AtDoor}_i & + & \mathtt{ok?}(\mathsf{leave}_i)\ .\ \mathsf{Out}_i\,.
\end{array}$$

In a similar manner, the n processes $\mathsf{Client}_i(\mathsf{CS})$ are augmented with the actions $\mathtt{at?}$ and $\mathtt{ok!}$. As these global processes are involved as employees in the protocol, the $\mathtt{emp}$ actions have been put in place as well. See Figure 1.

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

The nondeterministic $\mathsf{NDetServer}$ STD is defined by the following specification:

$$\begin{array}{rcrl}
\mathsf{Idle} & = & \sum_{i=1}^{n} \mathtt{man}(\mathsf{check}_i) & .\ \mathsf{NDChecking}_i \\
\mathsf{NDChecking}_i & = & \mathtt{man}(\mathsf{permit}_i) & .\ \mathsf{NDHelping}_i\quad + \mathtt{man}(\mathsf{refuse}_i)\ .\ \mathsf{Idle} \\
\mathsf{NDHelping}_i & = & \mathtt{man}(\mathsf{continue}_i) & .\ \mathsf{Idle}\,.
\end{array}$$

See also Figure 4. For the communication function '|' we put

$$\mathtt{at!}(s) \mid \mathtt{at?}(s) = \mathtt{at}(s) \quad \text{and} \quad \mathtt{ok?}(a) \mid \mathtt{ok!}(a) = \mathtt{ok}(a),$$

for $s = \mathsf{Out}_i, \mathsf{Waiting}_i, \mathsf{AtDoor}_i$, 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, useful for verification. Moreover, in the case of the protocol driven by $\mathsf{NDetServer}$, we assume

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

All actions from the set $H = \{\,\mathtt{man}, \mathtt{emp}, \mathtt{at?}, \mathtt{at!}, \mathtt{ok?}, \mathtt{ok!}\,\}$ need to be blocked to enforce communication. Finally, the collaboration process is given as

$$\mathsf{CSNDet} = \partial_H(\mathsf{Client}_1 \parallel \mathsf{Client}_1(\mathsf{CS}) \parallel \ldots \parallel \mathsf{Client}_n \parallel \mathsf{Client}_n(\mathsf{CS}) \parallel \mathsf{NDetServer})\,.$$

For the round-robin case, the translations of the Client_i STDs remain the same. The translation of the global STD $\mathsf{Client}_i(\mathsf{CS})$ changes slightly only. We simply adapt $\mathsf{Client}_1(\mathsf{CS})$, the global process of the first client. This is because this global STD will start in phase $\mathsf{Interrupt}$. More specifically, its starting state is $\mathsf{Interrupt}[\mathsf{triv}]$.

$$\mathsf{Client}_1(\mathsf{CS}) = \mathsf{Interrupt}_i[\mathsf{triv}] \quad \text{and} \quad \mathsf{Client}_k(\mathsf{CS}) = \mathsf{Without}_i[\mathsf{triv}] \quad \text{for } k \geq 1$$

The STD of $\mathsf{RoRoServer}$ does have a rather different translation however, reflecting the different character of the protocol. See Figure 5.

$$\begin{aligned}\mathsf{RRChecking}_j &= \quad \mathtt{man}(\mathsf{grant}_j)\ .\ \mathsf{RRHelping}_j + \mathtt{man}(\mathsf{pass}_j)\,.\,\mathsf{RRChecking}_{j+1}\\ \mathsf{RRHelping}_j &= \mathtt{man}(\mathsf{proceed}_j)\ .\ \mathsf{RRChecking}_{j+1}\,.\end{aligned}$$

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

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

The set of blocked actions is now defined as $H' = \{\,\mathtt{man}, \mathtt{emp}, \mathtt{at?}, \mathtt{at!}, \mathtt{ok?}, \mathtt{ok!}\}$. The complete collaborative process for the round-robin protocol is defined by

$$\mathsf{CSRoRo} = \partial_{H'}(\mathsf{Client}_1 \parallel \mathsf{Client}_1(\mathsf{CS}) \parallel \ldots \parallel \mathsf{Client}_n \parallel \mathsf{Client}_n(\mathsf{CS}) \parallel \mathsf{RoRoServer}).$$

Translation of ACP-based specifications into the input language of the mCRL2 toolset of the n clients Client_i, the global $\mathsf{Client}_i(\mathsf{CS})$ and the servers NDetServer and RoRoServer is largely straightforward. In the appendix, the mCRL2 specifications of the collaborative processes for three clients are given. Similarities with the ACP specifications are obvious. From these specifications the tool generates corresponding transition systems. For four clients the entire state space for the round-robin protocol consists of 1080 states and 3456 transitions. However, if we abstract from internal communications and focus on entrance and exit of the critical section, it can be reduced to 77 states with 204 transitions only using branching bisimulation. More generally, we found in the round-robin case, for n clients ($n \geq 2$) the number of states can be reduced to $(5{\cdot}2^{n-2}{-}1){\cdot}n + 1$.

We consider the following properties for the two protocols, where we assume clients Client_i and Client_j for $i \neq j$:

1. At any moment of 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. In μ-calculus, the property specification language of mCRL2, it 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 $\mathsf{ok}(\mathsf{explain}_j)$ and no action $\mathsf{ok}(\mathsf{thank}_i)$ appears in between is impossible. Clearly, we use $\mathsf{ok}(\mathsf{explain}_i)$ and $\mathsf{ok}(\mathsf{thank}_i)$ to detect entering and leaving the critical section. For further details we refer to [5].

2. At any moment of time the server will not permit two different clients to be served. In other words, two or more clients will be not allowed to access the critical section at the same time. Obviously, this corresponds to the previous property, only seen from the server perspective. In μ-calculus, for RoRoServer (and similar for NDetServer) it is expressed as:

   $$\texttt{[true*.grant}_i\texttt{.(!proceed}_i\texttt{)*.grant}_j\texttt{] false}$$

3. 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,

using the action transition of the underlying 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 is a sequence of actions in which an occurrence of action $\mathsf{ok}(\mathsf{enter}_i)$ is followed (not necessarily in a consecutive way) by $\mathsf{ok}(\mathsf{enter}_j)$ before any occurrence of action $\mathsf{ok}(\mathsf{thank}_i)$.

4. 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 be allowed access, under the assumption that any loop that does not contain this action will be eventually left.

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

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

As expected, the first four properties are valid for the non-deterministic server, while the fifth one is not, because the NDetServer does not guarantee general access to the critical section. This is clear from the specification, as the NDetServer can always ignore a client, even if it has requested a service. On the other hand, all five properties are valid for the round-robin protocol.

5 Embedding Paradigm in ACP

Based on the two example translations presented above, we proceed by formulating how to express a general Paradigm model in ACP. For clarity, we restrict to the hierarchal case where a component in a collaboration is either a manager or an employee [8]. However, employees are allowed to have multiple roles addressing different managers. Furthermore, for ease of presentation, we assume action-determinism, i.e. that any two different transitions have different actions. This way, a transition is identified by its label.

Employees synchronize their detailed behaviour with the global behaviour, while the global behaviour is governed by the consistency rules. The behaviour of a manager is connected to that of the employees by means of the consistency rules as well. The process algebraic translation of an employee contains 'informing' and 'performing' elements. To clarify this dual nature, in Figure 6 we repeat essential fragments from the architectural impression from Figure 3, additionally decorated with communication actions. The 'informing' in the process for one employee is modeled by the action `at`. It keeps the process unchanged, thus informing about the original STD state towards one performing role. In addition, 'performing' by an employee is modeled by the action `ok`, with relevant argument.

We need a refined form of the original global STDs, as in Figure 2b. After such a role process has been informed about the current state of the detailed STD and has

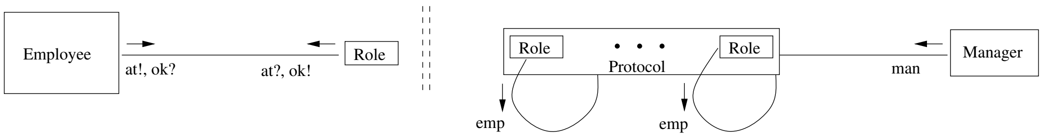

Fig. 6. Process interaction: (a) sequential, (b) synchronized.

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 refined global state: with the same phase as before, but with the inner trap replacing the trap entered so far.

An employee Z in Paradigm has a number of roles, R_1 to R_n say. In each role we distinguish phases p_{ij}, $1 \leq i \leq n$, $1 \leq j \leq n_i$. For each state s, we define a recursive equation in ACP as follows:

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

For a phase p in a role R_i of the employee Z, we define a system of recursive equations $p[t]$, indexed by the traps t in the phase p.

$$\begin{aligned} p[t] = {} & \textstyle\sum_{s \in t' \subseteq t} \mathtt{at?}(s).p[t'] + \sum_{s \xrightarrow{a} s' \in p} \mathtt{ok}_i(a).p[t] + \\ & \textstyle\sum\{\ \mathsf{cr}_\gamma(a, t_1, \ldots, t, \ldots, t_m).p'[\mathsf{triv}] \mid \\ & \qquad \gamma \colon s \xrightarrow{a} s' \ * \ p_1 \xrightarrow{t_1} p_1',\ \ldots,\ p \xrightarrow{t} p',\ \ldots,\ p_m \xrightarrow{t_m} p_m'\ \}. \end{aligned}$$

Thus, the detailed STD is willing to communicate its state information to every partition; a partition only synchronizes via $\mathtt{at?}(s)$ if s is in an inner trap t' of the current trap t. The communication function satisfies $\mathtt{at!}(s) \mid \mathtt{at?}(s) = \tau$ for $s \in Z$. A transition $s \xrightarrow{a} s'$ in the detailed STD can only be made if allowed by all the partitions. Therefore, the single $\mathtt{ok?}(a)$ of A_s should be matched by all $\mathtt{ok}_i(a)$ in any of the phases $p[t]$, t a trap of p, p a phase of P_i, $1 \leq i \leq n$. In this case, synchronization is amongst $n+1$ parties, the detailed STD and its n partitions. We put $\mathtt{ok?}(a) \mid \mathtt{ok}_1(a) \mid \cdots \mid \mathtt{ok}_n(a) = \tau$. 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. If the consistency rule is fired, the phase p' is entered in the partition P_i. As there is no specific state information, the trivial trap triv is assumed. Synchronization and communication function are to be discussed below.

The process expression for a manager Z is simpler, as there is no interaction with its possible roles. However, all transitions made by the manager should match a consistency rule for the particular collaboration. For a state $s \in Z$, we now have the recursive equation

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

So, in the collaboration, apart from the manager Z, m employees are involved. For a consistency rule γ to apply, Z must have reached state s, while the employees 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,

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

From the above general translation outline one can see, that and how more complicated Paradigm models can be dealt with. More than one manager per protocol just has, at the time, at most one cr action of one of the managers, synchronized with the relevant combination of cr actions from different processes for different roles, possibly driven differently.

6 Conclusions

In this paper we have linked the coordination modeling language Paradigm via the process algebra ACP with the mCRL2 toolset, enabling systematic translation of Paradigm models into ACP and subsequent tool supported verification of their correctness. Each Paradigm STD, detailed or global, is described as a recursive specification. Thus, separate specifications exist for each global and detailed STD, managers included. The composed pair of a detailed and corresponding global representation (an employee and one role of it) has specific interaction, guaranteeing consistent progress of combined dynamics. The interaction of global dynamics of the coordinated components, driven by a manager process, is described in Paradigm by the consistency rules. Therefore, in the translation, the consistency rules are captured by the communication function. Thus, all interaction between Paradigm STDs, either explicitly described (consistency rules) or implicitly, is flattened: represented, after the process algebraic translation, by a single communication pattern indeed, loosing its architectural level characteristics.

As various verification techniques can be applied to a process algebraic specification, via the translation, correctness of Paradigm models can now be analyzed. For our experiments, we have used the mCRL2 toolset, which is closely related to the ACP process algebra. The ACP specifications for both examples of the non-deterministic and round-robin protocols, have been analyzed with the tool. mCRL2 has generated automatically the complete state spaces for further inspection. We have run the model-checker and checked several properties of interest. The results confirmed correctness of the Paradigm models. It is noted, the translation into ACP, although introducing explicit exchange of state and step information, provides a semantical definition by itself.

In addition to confirming the correctness of a Paradigm model by means of model checking its PA translation, counterexamples found by the model checker can be translated back into the Paradigm setting too. The generated trace refuting the property under consideration, can be reconstructed at the level of Paradigm. We present an outline of such inverse translation here: When mapping to PA, each Paradigm model is eventually translated into a collaboration process $\partial_H(\ldots \parallel \ldots \parallel \ldots)$, a parallel composition of constituent processes specified as explained above. In fact, each such constituent process uniquely corresponds to a Paradigm STD. Moreover, the dynamics of one such STD –sequences of transitions– correspond to rather similar sequences of atomic actions taken according to that process' specification, interspersed additionally with communication actions `at`?(), `at`!(), `ok`!(). The similarity of both types of sequences lies in the property of preserving the original step order –the STD dynamics– per constituent process from the PA translation; that is, adorned with a communication indicator `ok`? in case of an employee step, adorned with a communication indicator `emp` in case of a non-refined global STD step, or adorned with a communication indicator `man` in case of a manager step. Note, the latter three communication actions `ok`?(), `emp`(), `man`() do correspond one-to-one to a Paradigm STD transition; the former three communication actions `at`?(), `at`!(), `ok`!() are extra, needed for technical PA reasons.

So, a particular counterexample can be traced back to the particular composition of similar sequences of steps per STD, to clarify where in the original Paradigm model the problem occurs. On the basis of Paradigm's semantics, formulated in the product space spanned by the STDs of the various Paradigm processes (cf. the appendix in [7]), the reason of the problem occurring then is visible. As is to be expected, understanding why the Paradigm model didn't satisfy a particular property, while it was designed to do so, remains the task of the modeler. In the Paradigm domain, faulty or insufficient communication is being mirrored in terms of wrongly chosen phases (`at`?, `at`! communication) or traps (`ok`?, `ok`! communication) or consistency rules (`emp`, `man` communication).

Last but not least, we did some verification based on equivalence checking, as supported by the `mCRL2` toolset. Experiments with branching bisimulation reduction yielded substantial reductions of the state spaces. In particular, for the nondeterministic protocol, we obtain a very simple structure after reduction. However, due to interleaving of internal activity, for the round-robin protocol a minimal model remains rather complex. In our future work we will investigate how further simplification can be achieved. To that aim, we want to formulate suitable process equivalences, possibly depending on Paradigm's architectural structure. Other future work is to extend the translation presented here, to Paradigm's approach towards self-adaptation, based on lazily defined coordination of migration. Verification of such self-adaptive systems then should be straightforward.

References

[1] R.J. Allen. *A Formal Approach to Software Architectures.* PhD thesis, Carnegie Mellon University, 1997.

[2] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* CUP, 1990.

[3] M.A. Barbosa and L.S. Barbosa. An orchestrator for dynamic interconnection of software components. In G. Boella et al., editor, *Proc. CoOrg 2006 and MTCoord 2006*, pages 49–61. ENTCS 181, 2007.

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

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

[6] G. Engels, R. Heckel, J.M. Küster, and L. Groenewegen. Consistency-preserving model evolution through transformations. In J.-M. Jézéquel, H. Hußmann, and S. Cook, editors, *Proc. UML 2002*, pages 212–226. LNCS 2460, 2002.

[7] L. Groenewegen, A. Stam, P. Toussaint, and E. de Vink. Paradigm as organization-oriented coordination language. In L. van de Torre and G. Boella, editors, *Proc. CoOrg 2005*, pages 93–113. ENTCS 150(3), 2005.

[8] 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.

[9] 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.

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

[11] J.F. Groote et al. The formal specification language mCRL2. In E. Brinksma et al., editor, *Methods for Modelling Software Systems*. IBFI, Schloss Dagstuhl, 2007. 34 pages.

[12] J.M. Küster. *Consistency Management of Object-Oriented Behavioral Models*. PhD thesis, University of Paderborn, 2004.

[13] 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*, pages 161–204, 2008.

[14] M.A. Pérez-Toledano, A. Navasa, J.M. Murillo, and C. Canal. TITAN: a framework for aspect-oriented system evolution. In *Proc. ICSEA 2007, Cap Esterel*. IEEE, 2007. 8 pages.

[15] P. Poizat and J.-C. Royer. A formal architectural description language based on symbolic transition systems and modal logic. *Journal of Universal Computer Science*, 12:1741–1782, 2006.

[16] 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.

[17] M. Sirjani, A. Movaghar, A. Shali, and F.S. de Boer. Modeling and verification of reactive systems using Rebeca. *Fundamenta Informaticae*, 63:385–410, 2004.

A mCRL2 code of the NDet-server example

```
%% model with non-deterministic server for 3 clients

sort
  STran = struct check | permit | refuse | continue ;

  CName = struct A | B | C ;
  CStat = struct out | waiting | busy | atdoor ;
  CTran = struct enter | explain | thank | leave ;
  CTrap = struct triv | notyet | request | done ;

act
  ati, ato, at : CName # CStat ;
  oki, oko, ok : CName # CTran ;
  man, emp, sync : STran # CName # CTrap ;

proc
  %% detailed server process
  ServerIdle = sum i:CName .
    man(check,i,triv) . ServerChecking(i);
  ServerChecking(i:CName) =
    man(refuse,i,notyet).ServerIdle +
    man(permit,i,request) . ServerHelping(i);
  ServerHelping(i:CName) =
    man(continue,i,done) . ServerIdle;

  %% detailed client process
  ClientOut(i:CName) =
    ato(i,out).ClientOut(i) +
    oki(i,enter).ClientWaiting(i) ;
  ClientWaiting(i:CName) =
    ato(i,waiting).ClientWaiting(i) +
    oki(i,explain).ClientBusy(i) ;
  ClientBusy(i:CName) =
    ato(i,busy).ClientBusy(i) +
    oki(i,thank).ClientAtDoor(i) ;
  ClientAtDoor(i:CName) =
    ato(i,atdoor).ClientAtDoor(i) +
    oki(i,leave).ClientOut(i) ;

  %% global client process for partition CSM
  ClientCSMWithoutTriv(i:CName) =
    oko(i,leave).ClientCSMWithoutTriv(i) +
    oko(i,enter).ClientCSMWithoutTriv(i) +
    emp(check,i,triv).ClientCSMInterruptTriv(i);
  ClientCSMInterruptTriv(i:CName) =
    ati(i,out).ClientCSMInterruptNotYet(i) +
    ati(i,atdoor).ClientCSMInterruptNotYet(i) +
    ati(i,waiting).ClientCSMInterruptRequest(i);
  ClientCSMInterruptNotYet(i:CName) =
    oko(i,leave).ClientCSMInterruptNotYet(i) +
    emp(refuse,i,notyet).ClientCSMWithoutTriv(i);
  ClientCSMInterruptRequest(i:CName) =
    emp(permit,i,request).ClientCSMWithTriv(i);
  ClientCSMWithTriv(i:CName) =
    ati(i,atdoor).ClientCSMWithDone(i) +
    oko(i,explain).ClientCSMWithTriv(i) +
    oko(i,thank).ClientCSMWithTriv(i);
  ClientCSMWithDone(i:CName) =
    emp(continue,i,done).ClientCSMWithoutTriv(i);

init
  allow(
    { at, ok, sync },
  comm(
    { ati|ato -> at,
      oki|oko -> ok,
      man|emp -> sync },
    ServerIdle ||
    ClientOut(A) || ClientCSMWithoutTriv(A) ||
    ClientOut(B) || ClientCSMWithoutTriv(B) ||
    ClientOut(C) || ClientCSMWithoutTriv(C)
  ));
```

B mCRL2 code of the RoRo-server example

```
%% model with round-robin server for 3 clients

sort
  CStat = struct Out | Waiting | Busy | AtDoor ;
  CTran = struct enter | explain | thank | leave ;
  CTrap = struct triv | notyet | request | done ;

sort
  CN = struct A | B | C ;

map next : CN -> CN ;
eqn
  next(A) = B ; next(B) = C ; next(C) = A ;

sort
  STran = struct grant | proceed | pass;

act
  ati, ato, at : CN # CStat ;
  oki, oko, ok : CN # CTran ;
  man3, emp3, sync3 : STran # CN # CTrap ;
  man5, emp5, sync5 : STran # CN # CTrap  # CN # CTrap ;

proc
  %% detailed server process
  RRServerChecking(i:CN) =
    man3(grant,i,request) . RRServerHelping(i) +
    man5(pass,i,notyet,next(i),triv) . RRServerChecking(next(i)) ;
  RRServerHelping(i:CN) =
    man5(proceed,i,done,next(i),triv) . RRServerChecking(next(i)) ;

  %% detailed client process
  ClientOut(i:CN) =
    ato(i,Out) . ClientOut(i) +
    oki(i,enter) . ClientWaiting(i) ;
  ClientWaiting(i:CN) =
    ato(i,Waiting) . ClientWaiting(i) +
    oki(i,explain) . ClientBusy(i) ;
  ClientBusy(i:CN) =
    ato(i,Busy) . ClientBusy(i) +
    oki(i,thank) . ClientAtDoor(i) ;
  ClientAtDoor(i:CN) =
    ato(i,AtDoor) . ClientAtDoor(i) +
    oki(i,leave) . ClientOut(i) ;
  %% global client process for partition RR
  ClientRRWithoutTriv(i:CN) =
    oko(i,leave) . ClientRRWithoutTriv(i) +
    oko(i,enter) . ClientRRWithoutTriv(i) +
    sum i':CN,t':CTrap .
      emp5(proceed,i',t',i,triv) . ClientRRInterruptTriv(i);
  ClientRRInterruptTriv(i:CN) =
    oko(i,leave) . ClientRRInterruptTriv(i) +
    ati(i,Out) . ClientRRInterruptNotYet(i) +
    ati(i,AtDoor) . ClientRRInterruptNotYet(i) +
    ati(i,Waiting) . ClientRRInterruptRequest(i);
  ClientRRInterruptNotYet(i:CN) =
    oko(i,leave) . ClientRRInterruptNotYet(i) +
    sum i':CN,t':CTrap .
      emp5(pass,i,notyet,i',t') . ClientRRWithoutTriv(i);
  ClientRRInterruptRequest(i:CN) =
    emp3(grant,i,request) . ClientRRWithTriv(i);
  ClientRRWithTriv(i:CN) =
    ati(i,AtDoor) . ClientRRWithDone(i) +
    oko(i,explain) . ClientRRWithTriv(i) +
    oko(i,thank).ClientRRWithTriv(i);
  ClientRRWithDone(i:CN) =
    sum i':CN,t':CTrap .
      emp5(proceed,i,done,i',t') . ClientRRWithoutTriv(i);

init
  allow(
    { at, ok, sync3, sync5 },
  comm(
    { ati|ato -> at,
      oki|oko -> ok,
      man3|emp3 -> sync3,
      man5|emp5|emp5 -> sync5 },
    RRServerChecking(A) ||
    ClientOut(A) || ClientRRInterruptTriv(A) ||
    ClientOut(B) || ClientRRWithoutTriv(B) ||
    ClientOut(C) || ClientRRWithoutTriv(C)
  )));
```