Extending Paradigm with Data

L.P.J. Groenewegen[1], J.H.S. Verschuren[1], and E.P. de Vink[2,3,⋆]

[1] Leiden Institute of Advanced Computer Science
Leiden University, The Netherlands
[2] Department of Mathematics and Computer Science
Eindhoven University of Technology, The Netherlands
[3] Centrum Wiskunde & Informatica, Amsterdam, The Netherlands

Abstract. We discuss an extension of the coordination modeling language Paradigm. The extension is geared towards data-dependent interaction among components, where the coordination is influenced by possibly distributed data. The approach is illustrated by the well-known example of a bakery where tickets are issued to serve clients in order. Also, it is described how to encode Paradigm models with data in the process language of the `mCRL2` toolset for further analysis of the coordination.

1 Introduction

The so-called IWIM model for the coordination of concurrent components as proposed by Farhad Arbab and co-workers [3, 6] distinguishes ideal workers and ideal managers. Among others, IWIM forms the conceptual framework for the coordination language Manifold [4, 7]. The central ideas of IWIM evolved into the theory of Reo connectors [5], which exploits constraint automata for its semantics and whose distributed implementation approach separates coordination from parallelism [11].

Rather than considering hierarchies of components with atomic workers at the bottom layer and one overall manager at the top as for IWIM, the coordination modeling language Paradigm [9] takes networks of components as starting point, where each component exhibits both worker and manager activity. The worker activity is the internal behavior of the component that executes as local transitions asynchronously from other components; the manager activity consists of the synchronous interaction with other (groups of) components governed by so-called consistency rules. In terms of constraint automata, consistency rules comprise the atomic dataflow among synchronizing components. However, via a mechanism of phases and traps it is guaranteed that the local behavior, the worker level of a component, remains aligned with the global behavior, the manager level of the component.

In this paper an extension to Paradigm including data is proposed. In this extension, consistency rules incorporate the local variables of the components and

⋆ Corresponding author, email `evink@win.tue.nl`.

expressions thereof, in particular to compare or communicate their value. So, our data extension will be geared towards interaction and coordination thereof. Cast in terms of Reo, the data constraints are enriched with data and values. For Paradigm with data, the local memory of components can be accessed (via their ports) at the coordination level. Consequently, the communicated data itself can be stored too. However, this requires that the phases-and-trap mechanism of Paradigm needs to be adapted, somewhat complicating the semantics. An encoding scheme for Paradigm, without data, into the model checking toolsuit `mCRL2`[4], as proposed in [1], brings the advantage of formal analysis of the coordination among components. For a concrete coordination problem, we will describe a Paradigm model with data of a bakery, describe its encoding in `mCRL2`'s specification language.

Outline Section 2 provides a formal definition of Paradigm with data and provides its operational semantics. Section 3 illustrates and further explains the underlying concepts for the case of a bakery where clients need to be served in order of arrival. Section 4 discusses how formal analysis of Paradigm using the `mCRL2` toolset can be obtained. Section 5 wraps up the paper.

2 Formal definitions

We subsequently introduce components with variables, Paradigm models and consistency rules with data, and configurations with local transitions and global transfers among them. An example illustrating the above notions is presented in the next section.

Definition 1. *Let, for some index set I, a number of local variables v_i of type D_i, respectively, be given. Put $E = \prod_{i\in I} D_i$. Furthermore, fix a set of actions A. For E and A, a Paradigm component C is a tuple $C = (\Sigma, T, \Psi)$ where*

(i) for some set S, the elements of which are called states, $\Sigma = S \times E$ is the set of extended states of C
(ii) $T \subseteq \Sigma \times A \times \Sigma$ is the transition relation of C
(iii) $\Psi = (\Phi_1, \ldots, \Phi_n)$, for some $n \geqslant 0$, is a tuple of partial functions, called the roles of C, where each $\Phi : \mathcal{P}(T) \hookrightarrow \mathcal{P}(\mathcal{P}(\Sigma))$ is such that if $\sigma \in \theta$, $\theta \in \Phi(\varphi)$, and $\langle \sigma, a, \sigma' \rangle \in \varphi$ then also $\sigma' \in \theta$.

By definition, an extended state $\sigma \in \Sigma$ is a pair $\sigma = (s, e)$ of a state $s \in S$ and a tuple of 'current' values of the variables. We write $\sigma \xrightarrow{a} \sigma'$ for a transition in T, rather than $\langle \sigma, a, \sigma' \rangle \in T$. For a role Φ, i.e. a coordinate of Ψ, an element φ of $\mathrm{dom}(\Phi)$ is called a phase of Φ. An element θ of $\Phi(\varphi)$ is called a trap of φ. The idea is, a transition, $\sigma \xrightarrow{a} \sigma'$ starting from an extended state σ in a trap θ of a phase φ say, does not move outside of the trap. Hence, it is required for such a transition $\sigma \xrightarrow{a} \sigma'$ that the extended state σ' lies in θ too. So, in phase φ, once having entered θ, control is trapped in θ. The phases constituting $\mathrm{dom}(\Phi)$ of a

[4] See `www.mcrl2.org`.

role Φ will typically partially overlap, their overlaps being traps. One may think of a trap as a final stage within a phase. Reaching a trap of a phase indicates that a transfer to another phase is about to happen.

Suppose $\varphi_i \in \Phi_i$, for $i = 1, \ldots, n$, for the roles $\Phi_1, \ldots, \Phi_n$ of component C, and suppose $\sigma \xrightarrow{a} \sigma'$ is a transition of C, i.e. an element of T, such that the transition $\sigma \xrightarrow{a} \sigma'$ is an element of each φ_i too. Then the transition $\sigma \xrightarrow{a} \sigma'$ is called an admitted transition with respect to the phases $\varphi_1, \ldots, \varphi_n$.

Definition 2.

(a) A Paradigm model with data consists, for some index set H, of a tuple $(C_h)_{h\in H}$ of Paradigm components

$$C_h = (\, \Sigma_h, T_h, \Psi_h \,)$$

with their own local variables and actions, as well as extended states in Σ_h, transition relations T_h, and roles $\Psi_h = (\Phi_{h,1}, \ldots, \Phi_{h,n_h})$, for $h \in H$.

(b) A consistency rule γ for $(C_h)_{h\in H}$ consists, for an index set R, of a tuple $(\, C_r(\Phi_r) : \varphi_r(e_r) \xrightarrow{\theta_r} \varphi'_r(e'_r) \,)_{r\in R}$ where Φ_r is a role of component C_r, φ_r and φ'_r are phases of Φ_r, e_r and e'_r are values for the variables of C_r, and θ_r is a trap of φ_r.

(c) A set of consistency rules Γ is called closed if for each rule $(\, C_r(\Phi_r) : \varphi_r(e_r) \xrightarrow{\theta_r} \varphi'_r(e'_r) \,)_{r\in R}$ of Γ, if there exists, for some $r \in R$, a state s_r of C_r for which both $(s_r, e_r), (s_r, \bar{e}_r) \in \theta_r$, then Γ contains, for some $\bar{e}'_r$, a rule $\bar{\gamma}$ of the form $(\, C_r(\Phi_r) : \varphi_r(\bar{e}_r) \xrightarrow{\theta_r} \varphi'_r(\bar{e}'_r) \,)_{r\in R}$ as well.

For clarity we assume that different components have distinct names for states, variables, and actions, and hence distinct roles, phases, and traps. However, in a consistency rule, a component may have multiple occurrences, viz. in different roles. Also, a component may not be involved in a consistency rule at all. The rules are called *consistency* rules in Paradigm because the requirement of each θ_r to be a trap of phase φ_r guarantees that the 'coarse-grained' rule can only be applied if *consistent* with the current 'fine-grained' local state of each component involved. The closedness condition on sets of rules will guarantee that global behavior, to be defined in a minute, cannot be essentially influenced by local behavior respecting the traps mentioned in a rule.

Next, we define the behavior of a Paradigm model, with intra-component behavior (a so-called local transition) affecting the extended state of a single component vs. inter-component behavior (a global transfer) exchanging values and changing phases based on a trap in some of the roles of a number of components.

Definition 3. *Let $\Pi = (C_h)_{h\in H}$ be a Paradigm model and let Γ be a closed set of consistency rules for Π.*

(a) A configuration of Π is a tuple $\langle s_h, e_h, \psi_h \rangle_{h\in H}$, where for each index $h \in H$, (s_h, e_h) is an extended state of component C_h, and $\psi_h = (\varphi_{h,1}, \ldots, \varphi_{h,n_h})$ is a tuple of phases such that $\varphi_{h,i} \in \Phi_{h,i}$, for $i = 1, \ldots, n_h$.

(b) *A local transition* $\langle s_h, e_h, \psi_h\rangle_{h\in H} \xrightarrow{a} \langle s'_h, e'_h, \psi_h\rangle_{h\in H}$ *of* Π *is an admitted transition of one of the components of* Π*, i.e. for some* $h_0 \in H$ *it holds that (i)* $\langle s_{h_0}, e_{h_0}\rangle \xrightarrow{a} \langle s'_{h_0}, e'_{h_0}\rangle$ *is an admitted transition for component* C_{h_o} *with respect to the phases* $\psi_{h_0} = (\varphi_{h_0,1}, \ldots, \varphi_{h_0,n_{h_0}})$*, and (ii)* $s_h = s'_h$ *and* $e_h = e'_h$ *for each index* $h \neq h_0$ *in* H*.*

(c) *A global transfer* $\langle s_h, e_h, \psi_h\rangle_{h\in H} \xrightarrow{\gamma} \langle s_h, e'_h, \psi'_h\rangle_{h\in H}$ *of* Π *based on a consistency rule* $\gamma = (\,\hat{C}_r(\Phi_r) : \varphi_r(\hat{e}_r) \xrightarrow{\theta_r} \varphi'_r(\hat{e}_h)\,)_{r\in R}$ *updates phases and values as prescribed by* γ*, i.e. (i) if, for* $h \in H$*,* $i = 1, \ldots, n_h$*, it holds that* $C_h = \hat{C}_r$ *and* $\Phi_{h,i} = \Phi_r$*, for some index* $r \in R$*, then* $(s_h, e_h) \in \theta_r$*,* $e_h = \hat{e}_r$ *and* $e'_h = \hat{e}_r$*,* $\varphi_{h,i} = \varphi_r$ *and* $\varphi'_{h,i} = \varphi'_r$*, and (ii) if, for* $h \in H$*,* $C_h \neq \hat{C}_r$ *for each* $r \in R$ *then* $e_h = e'_h$*, and, for* $h \in H$*,* $i = 1, \ldots, n_h$*,* $\Phi_{h,i} \neq \Phi_r$ *for each* $r \in R$ *then* $\varphi_{h,i} = \varphi'_{h,i}$*.*

A configuration $\langle s_h, e_h, \psi_h\rangle_{h\in H}$ of a Paradigm model $\Pi = (C_h)_{h\in H}$ holds for each component C_h the current extended state (s_h, e_h) as well as the current phase $\varphi_{h,i}$ for each role $\Phi_{h,i}$ of C_h.

Note, in a local transition $\langle s_h, e_h, \psi_h\rangle_{h\in H} \xrightarrow{a} \langle s'_h, e'_h, \psi_h\rangle_{h\in H}$, say for component C_{h_0}, component C_{h_0} nor any of the other components changes phase; the tuple of phases ψ_h is for each component the same in the source configuration $\langle s_h, e_h, \psi_h\rangle_{h\in H}$ and the target configuration $\langle s'_h, e'_h, \psi_h\rangle_{h\in H}$ of the transition. However, the transition must be admitted for C_{h_0}, i.e. it must be present in all of the phases $\varphi_{h_0,i}$ of ψ_{h_0} for component C_{h_0}.

For a global transfer based on a consistency rule γ to apply, the current phases $\varphi_{h,i}$ of role $\Phi_{h,i}$ must match the phases of Φ_r, if $C_h = \hat{C}_r$ and $\Phi_{h,i} = \Phi_r$. Also, the extended states of the components $\hat{C}_r$ involved must lie in the traps θ_r, for all $r \in R$. States remain unaffected, but values of variables may change for the components mentioned in the rule, presumably because of the interaction. Phases may change too for the components mentioned, from $\varphi_{h,i} = \varphi_r$ to $\varphi_{h,i} = \varphi'_r$, which are both phases within the role $\Phi_{h,i} = \Phi_r$. Components C_h and phases $\varphi_{h,i}$ not mentioned by consistency rule γ remain the same.

We have the following result.

Theorem 1. *Let* $\Pi = (C_h)_{h\in H}$ *be a Paradigm model, and let* Γ *be a closed set of consistency rules for* Π*. Suppose*

$$\langle s_h, e_h, \psi_h\rangle_{h\in H} \xrightarrow{\gamma} \langle s_h, e'_h, \psi'_h\rangle_{h\in H} \quad \textit{and} \quad \langle s_h, e_h, \psi_h\rangle_{h\in H} \xrightarrow{a} \langle s'_h, e''_h, \psi_h\rangle_{h\in H}$$

for configurations $\langle s_h, e_h, \psi_h\rangle_{h\in H}$*,* $\langle s_h, e'_h, \psi'_h\rangle_{h\in H}$*, and* $\langle s'_h, e''_h, \psi_h\rangle_{h\in H}$ *of* Π*, a consistency rule* γ *in* Γ*, and a local transition for* a*. Then also*

$$\langle s_h, e_h, \psi_h\rangle_{h\in H} \xrightarrow{\gamma} \langle s_h, \bar{e}'_h, \psi'_h\rangle_{h\in H}$$

for suitable values $\bar{e}'_h$*, for* $h \in H$*.*

The theorem is a direct consequence of the closedness condition for the set of consistency rules. It states that the execution of a local transition cannot disable the execution of a consistency rule. This is the loose coupling in Paradigm

between the interaction between components and actions of the components of their own. The reverse obviously doesn't hold. A local transition that was admitted before, may be forbidden by one of the phases put in place by the execution of a consistency rule. Care has to be taken to deal with variables that are set by local transitions as well as by global transfer. To ensure non-interference of the global (manager) and local (worker) level, one may want to restrict reading or updating of variables to happen outside of the traps involved in consistency rules that may change the value of the different variables.

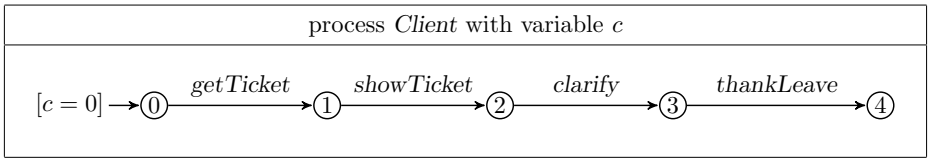
3 An example Paradigm model

We illustrate the formal definitions of the previous section by modeling in Paradigm with data the handling of clients in a busy bakery. Clients entering the shop take a ticket from a ticket dispenser and wait for their turn. The client having the ticket displayed is being served. The baker increments the display after having handled a client and next serves the client holding the ticket with the new number.

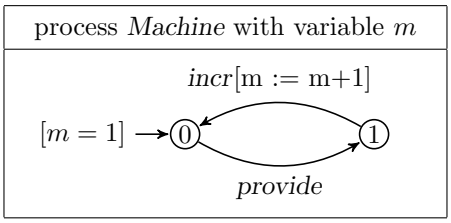
3.1 STDs for the components

We first model the basic behavior of the components by means of state-transition diagrams (STD).

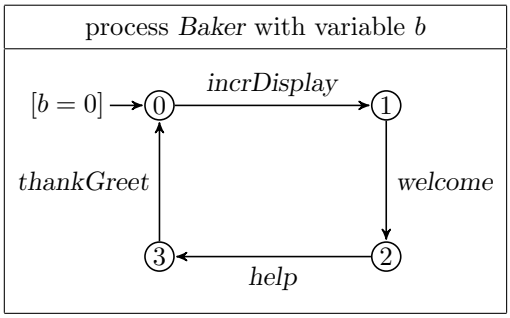
Client processes are introduced by the state-transition diagram below. Each client carries an integer variable c to hold a ticket number. Initially c is set to 0. A client subsequently obtains a ticket from the ticket machine, action *getTicket*, shows the ticket to the baker (action *showTicket*), clarifies his or her wishes (action *clarify*), and finally thanks the baker and leaves (action *thankLeave*). Note, apart from initialization, there is no explicit assignment to variable c in the STD. Also, we don't bother to distinguish multiple instances of the *Client* process. Incorporating another variable, *id* say, holding the identity of a client would cater for this.



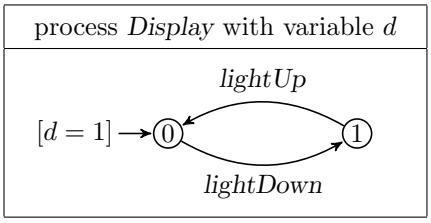
The ticket machine is modeled by the process *Machine* which has an integer variable m for the number of the ticket it dispenses. Starting from initial value 1 for m, the machine may provide a ticket with the current value of m while moving to state 1 (action *provide*) and returns to state 0 on an increment of the value of m (action *incr*). The *incr*-action is decorated with an assignment, viz. the increment $m := m+1$ of variable m.



The *Baker* process models the workflow for the baker. Starting from the initial state 0, with initial value 0 for the integer variable b of the process, the process cycles through its four states. First the baker aims to increment the display (action *incrDisplay*), next the baker welcomes the client holding the number displayed (action *welcome*) and helps the client (action *help*). The baker closes the cycle by some thanks and greetings (action *thankGreet*). Note, also here no explicit assignments to the variable b are present; changes to b will come from the interaction with the *Display* process described below.



The *Display* process is similar to the *Machine* process. It switches between two states. The *Display* process holds an integer variable d, initially set to 1. However, here we have chosen not to have an update of the variable in the STD as we have for the machine. As variation, the display gets incremented in the interaction with the baker. This is captured by the consistency rules modeling the interplay of these two processes.



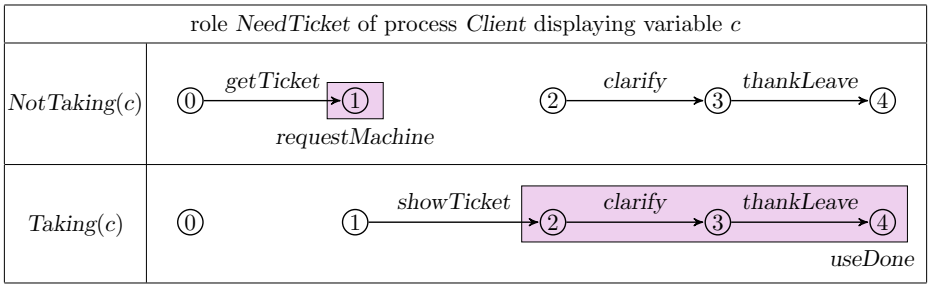
3.2 Roles of the components

As discussed above, in Paradigm a component can have multiple roles. At the level of the roles the interaction with other components takes place. The phase-

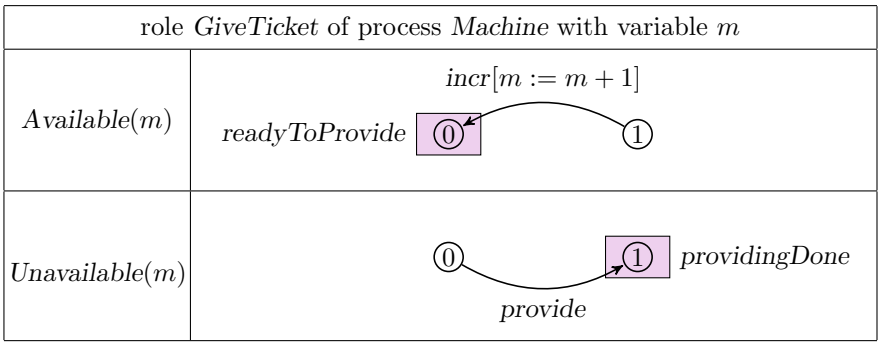
and-trap discipline of Paradigm ensures that STD and roles remain aligned during execution: a local transition cannot change the current phase or move out of a trap.

A *Client* process has two roles, *NeedTicket* and *NeedService*, in which it interacts with the *Machine* process and *Baker* process, respectively. The variable c of the *Client* process may be read and/or written during this interaction and is therefore displayed as parameter of the phases involved.

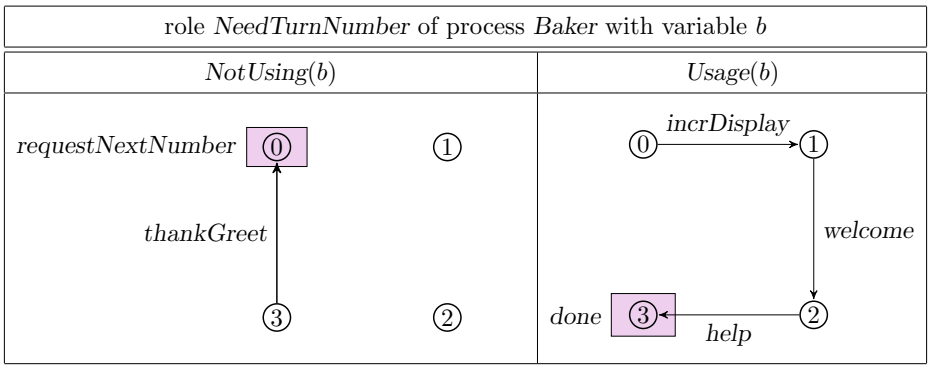
The role *NeedTicket* has two phases, *NotTaking* and *Taking*. The phase *NotTaking* only allows the action *getTicket* modeling that a ticket needs to be obtained first. When state 1 is reached in the STD the trap *requestMachine* has been entered, signaling that in the role *NeedTicket* the component is prepared to leave phase *NotTaking* (and ready to enter phase *Taking*, as we shall see). Phase *Taking* models a client in possession of a ticket. When state 2 has been reached, the trap *useDone* is entered. As required for a trap, the transitions for actions *clarify* and *thankLeave* do not leave trap *useDone*.



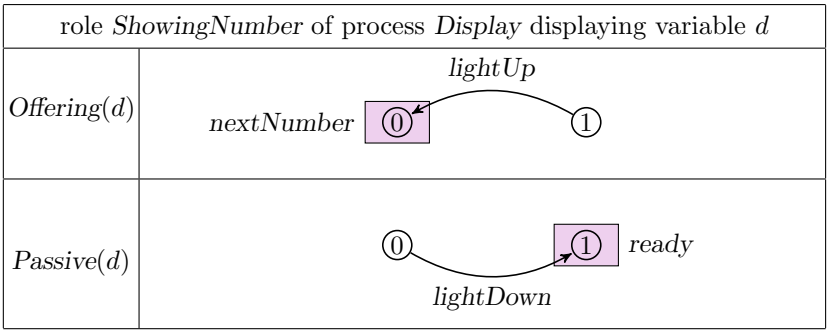
The *Machine* process in its single role *GiveTicket* interacts with the *Client* processes in their roles *NeedTicket*. The role *GiveTicket* has two phases, *Available* and *Unavailable*, that manage the variable m, as indicated. Both phases have a single-state trap, trap *readyToProvide* for phase *Available*, which indicates that a new ticket is available for issue when the trap is reached, and trap *providingDone* of phase *Unavailable*, that indicates that the current ticket number has been issued and the variable m needs to be adapted (as it actually will be in phase *Available*). Note, since variable m is updated when transition *incr*$[m := m + 1]$ is taken, the consistency rule (CM3) presented below doesn't have an increment of its parameter.



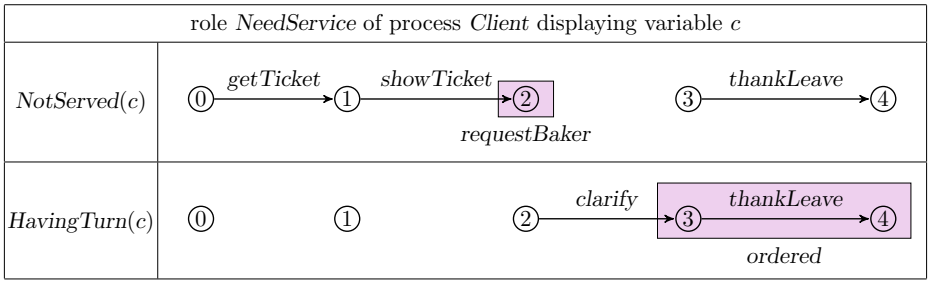
The *Baker* process has two roles, role *NeedTurnNumber* for interaction with the *Display* process, and role *NeedNextClient* for interaction with all *Client* processes. Role *NeedTurnNumber* distinguishes the phases *NotUsing* and *Usage*, that are connected by trap *requestNextNumber* from phase *NotUsing* to phase *Usage* and by trap *done* the other way around. The number of the client at turn is kept in the variable b of process *Baker*.



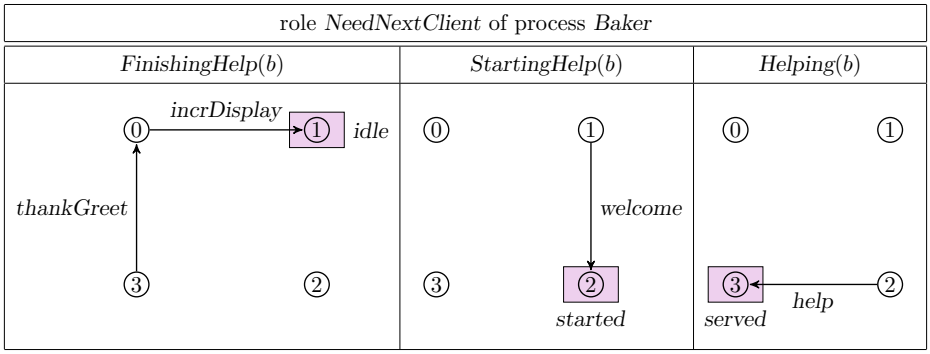
When the *Baker* process needs to know the next ticket number to store it in its variable b, this is provided by the *Display* process, in its single role *ShowingNumber*. In phase *Offering* the value of the variable d of process *Display* is guaranteed to be updated upon reaching trap *nextNumber*. To enforce such an update, phase *Offering* is switched to phase *Passive*, which will move control of *Display* to state 1 from which a next increment is possible once, and which is, via trap *ready*, switched back to phase *Offering*. Note, when changing phase from phase *Passive* to phase *Offering* as prescribed by consistency rule (BD3), given in the next subsection, the variable d will be incremented. Different from the modeling of role *GiveTicket* of the *Machine* process presented above, the role *ShowingNumber* of the *Display* process doesn't update the variable d itself.



The role *NeedService* of the *Client* process deals with the client-side in the interaction with the *Baker* process. The role has two phases, *NotServed* and *HavingTurn*, each making use of the variable c of the *Client* process during interaction, viz. to match the ticket number announced by the baker. Only in case of a match, the *Client* process will change phase to *HavingTurn*, based on the trap *requestBaker*. To highlight, be it a bit sketchy, that traps aren't necessarily comprised of a single state, the trap *ordered* of phase *HavingTurn* allows a transfer back to the phase *NotServed* again.



The role *NeedNextClient* of process *Baker* takes care of the baker's part in the interaction with a client. When having reached trap *idle* in phase *FinishingHelp* (finishing helping a previous client), a transfer will take place (by consistency rule (BC1) discussed below) to phase *StartingHelp*. Similarly, in phase *StartingHelp* on reaching trap *started* a transfer will take place (now by consistency rule (BC2)) to phase *Helping*, in which the client is actually served. After trap *ready* has been reached in phase *Helping*, the *Baker* process will switch to phase *FinishingHelp* in role *NeedNextClient*.



3.3 Interactions among components

The interaction between the *Client* processes, in their roles *NeedTicket*, and the *Machine* process, in its role *GiveTicket*, arranges that every client entering the bakery is provided with a uniquely numbered ticket. The three consistency rules (CM1), (CM2), and (CM3) below describe how phases change once proper traps have been entered.

$$\otimes\left\{\begin{array}{rrcll} \mathit{Client}(\mathit{NeedTicket}): & \mathit{NotTaking}(0) & \xrightarrow{\mathit{requestMachine}} & \mathit{Taking}(n) & \\ \mathit{Machine}(\mathit{GiveTicket}): & \mathit{Available}(n) & \xrightarrow{\mathit{readyToProvide}} & \mathit{Unavailable}(n) & \end{array}\right. \quad \text{(CM1)}$$

$$\mathit{Client}(\mathit{NeedTicket}): \quad \mathit{Taking}(n) \xrightarrow{\mathit{useDone}} \mathit{NotTaking}(n) \quad \text{(CM2)}$$

$$\mathit{Machine}(\mathit{GiveTicket}): \mathit{Unavailable}(n) \xrightarrow{\mathit{providingDone}} \mathit{Available}(n) \quad \text{(CM3)}$$

The first consistency rule, rule (CM1), is a synchronous transfer of phases, as indicated by the $\otimes$-symbol and enclosing braces. Given that (i) the *Client* process, in role *NeedTicket*, has reached trap *requestMachine* of phase *NotTaking*, while (ii) the *Machine* process, in role *GiveTicket* resides in trap *readyToProvide* of phase *Available*, then (i) the *Client* process switches to phase *Taking* of role *NeedTicket*, while (ii) the *Machine* process simultaneously changes to phase *Unavailable* of role *GiveTicket*. Moreover, (i) the *Client* process is assumed to (still) hold the initial value 0, while (ii) the *Machine* process has with a (presumably fresh) ticket number n, then the value n is copied from the *Machine* process to the *Client* process. For consistency rules (CM2) and (CM3) the *Client* and *Machine* process act independently. Based on (CM2), the *Client* process can change phase, from *Taking* to *NotTaking*, via trap *useDone*. Based on (CM3), the *Machine* process can change phase, from *Unavailable* to *Available*, via trap *providingDone*.

The interaction between the *Baker* and *Display* process is governed by the three consistency rules (BD1), (BD2), and (BD3). A similar effect is achieved as for rules (CM1) through (CM3). However, the actual update of variable m is done for the *Machine* process at the STD-level by the transition from state 1 to state 0 executing action $\mathit{incr}[m := m{+}1]$. Here, for process *Display* the update is

accomplished at the level of role *ShowingNumber* by rule (BD3), which passes the parameter value m for phase *Passive* to phase *Offering* as parameter value $m{+}1$.

$$\otimes \left\{ \begin{array}{lll} \mathit{Baker}(\mathit{NeedTurnNumber}) : \mathit{NotUsing}(n) & \xrightarrow{\mathit{requestNextNumber}} & \mathit{Usage}(m) \\ \mathit{Display}(\mathit{ShowingNumber}) : \mathit{Offering}(m) & \xrightarrow{\mathit{nextNumber}} & \mathit{Passive}(m) \end{array} \right. \quad \text{(BD1)}$$

$$\mathit{Baker}(\mathit{NeedTurnNumber}) : \mathit{Usage}(n) \xrightarrow{\mathit{done}} \mathit{NotUsing}(n) \quad \text{(BD2)}$$

$$\mathit{Display}(\mathit{ShowingNumber}) : \mathit{Passive}(m) \xrightarrow{\mathit{ready}} \mathit{Offering}(m{+}1) \quad \text{(BD3)}$$

The interaction of the *Baker* and *Client* processes in their respective roles *NeedNextClient* and *NeedService* is more tied up compared to the interactions described above. All three consistency rules (BC1), (BC2), and (BC3) prescribe simultaneous phase transfer for the two processes. Moreover, the value of the variable b of the *Baker* process must be equal to the variable c of the *Client* process; they must both have the value n.

$$\otimes \left\{ \begin{array}{lll} \mathit{Baker}(\mathit{NeedNextClient}) : \mathit{FinishingHelp}(n) & \xrightarrow{\mathit{idle}} & \mathit{StartingHelp}(n) \\ \mathit{Client}(\mathit{NeedService}) : \mathit{NotServed}(n) & \xrightarrow{\mathit{requestBaker}} & \mathit{NotServed}(n) \end{array} \right. \quad \text{(BC1)}$$

$$\otimes \left\{ \begin{array}{lll} \mathit{Baker}(\mathit{NeedNextClient}) : \mathit{StartingHelp}(n) & \xrightarrow{\mathit{started}} & \mathit{Helping}(n) \\ \mathit{Client}(\mathit{NeedService}) : \mathit{NotServed}(n) & \xrightarrow{\mathit{requestBaker}} & \mathit{HavingTurn}(n) \end{array} \right. \quad \text{(BC2)}$$

$$\otimes \left\{ \begin{array}{lll} \mathit{Baker}(\mathit{NeedNextClient}) : \mathit{Helping}(n) & \xrightarrow{\mathit{served}} & \mathit{FinishingHelp}(n) \\ \mathit{Client}(\mathit{NeedService}) : \mathit{HavingTurn}(n) & \xrightarrow{\mathit{ordered}} & \mathit{NotServed}(n) \end{array} \right. \quad \text{(BC3)}$$

4 Model checking Paradigm models with data

As for many modeling languages, formal analysis of Paradigm models supports the modeling itself. In [1, 2] it is discussed how to expressing Paradigm models without data in the process language of the `mCRL2` toolset [8, 10]. In short, for each component the local behavior is modeled as a state machine. For a transition to fire, it is checked if the current phase allows so. For the global behavior of a component a communication intent is issued for each consistency rule that mentions the component. However, the current state and phase should match the relevant trap. Correct interaction can subsequently be enforced by the allow and communication operators of `mCRL2`, that block single-sided communication intents and synchronize consistent ones, respectively. In this section, we describe by example how the approach extends to deal with data.

Figure 1 provides the `mCRL2` version of the process *Client* of the bakery example of the previous section (the complete code can be found in the appendix). Here, the `Client` process has four parameters, viz. the natural number `st` to hold the state of the underlying STD, the natural number `c` to hold the ticket number of the client, and the parameters `nt_ph` and `ns_ph` to keep track of the

```
1  proc Client(st:Nat,c:Nat,nt_ph:NeedTicketPh,ns_ph:NeedServicePh) =
2
3  ((st==0) && (nt_ph in [NotTaking]) && (ns_ph in [NotServed])) ->
4        getTicket . Client(1,c,nt_ph,ns_ph) +
5  ((st==1) && (nt_ph in [Taking]) && (ns_ph in [NotServed])) ->
6      showTicket(c) . Client(2,c,nt_ph,ns_ph) +
7  ((st==2) && (nt_ph in [NotTaking,Taking]) && (ns_ph in [HavingTurn])) ->
8       clarify(c) . Client(3,c,nt_ph,ns_ph) +
9  ((st==3) && (nt_ph in [NotTaking,Taking]) && (ns_ph in [HavingTurn])) ->
10      thankLeave(c) . Client(4,c,nt_ph,ns_ph) +
11
12 %% rule (CM1)
13    ((st in [1]) && (nt_ph==NotTaking)) ->
14      sum m:Nat . requestMachine(m) . Client(st,m,Taking,ns_ph) +
15 %% rule (CM2)
16    ((st in [2,3,4]) && (nt_ph==Taking)) ->
17      useDone . Client(st,c,NotTaking,ns_ph) +
18
19 %% rule (BC1)
20    ((st in [2]) && (ns_ph==NotServed)) ->
21      requestBaker1(c) . Client(st,c,nt_ph,NotServed) +
22 %% rule (BC2)
23    ((st in [2]) && (ns_ph==NotServed)) ->
24      requestBaker2(c) . Client(st,c,nt_ph,HavingTurn) +
25 %% rule (BC3)
26    ((st in [3,4]) && (ns_ph==HavingTurn)) ->
27      ordered(c) . Client(st,c,nt_ph,NotServed) ;
```

Fig. 1. mCRL2 code for the *Client* process

phase of the process with respect to their roles *NeedTicket* and *NeedService*, respectively. For this, the definition of the two enumerated types

```
NeedTicketPh = struct NotTaking | Taking ;
NeedServicePh = struct NotServed| HavingTurn ;
```

are included at the beginning of the specification. The specification of the `Client` process falls into three parts: (i) lines 3–10, specifying the local transitions, (ii) lines 12–17, describing *Client*'s part for the consistency rules (CM1) – (CM3), and similarly (iii) lines 19–27 for the consistency rules (BC1) – (BC3). Each part consists of a number of alternative branches, separated by the non-deterministic choice operation '`+`', of the form

```
<condition> -> <action> . <continuation process>
```

(with a variation for rule (CM1) to be discussed in a minute). For example, lines 3–4 express that the `Client` process in state `0`, in phase `NotTaking` for its *NeedTicket* role, as well as in phase `NotServed` for its *NeedService* role, can perform the action `getTicket` and continues as `Client(1,c,nt_ph,ns_ph)` with control now in state `1`, but leaving the parameters `c`, `nt_ph`, and `ns_ph` unchanged. The element-of-list construction `nt_ph in [NotTaking ,Taking]` shows profitable in line 7, for example. Note, the transition ② $\xrightarrow{clarify}$ ③ is admitted both by phase *NotTaking* and by phase *Taking*.

Lines 12–14 embody the contribution of a client process to execution of the (CM2)-rule. If the process resides in trap *useDone* consisting of states 2, 3,

and 4 (for all values of variable c), and in phase *Taking* regarding its *NeedTicket* role, then the process is willing to execute the action *useDone* and to continue with its *NeedTicket* phase changed from *Taking* to *NotTaking* as consistency rule (CM2) prescribes. Note, no for (CM2) the client process doesn't depend on other processes.

Consistency rule (CM1) in which both a client process and the machine process are involved requires their interaction. Assuming `Client` is in state `1`, hence in trap *requestMachine* as the value of c doesn't matter for this, as well as in phase *NotTaking*, then `Client` is willing to input the value `m` of the ticket as offered by the machine. But, a priori this value is not known to the client. Therefore, the value is abstracted away by the summation `sum m:Nat` over all possible values for `m`. Upon synchronization with the machine process the actual value for `m` will be handed over. However, this can only happen if the interacting *Machine* process has reached the proper trap in the proper phase regarding the corresponding role.

To enforce synchronization of processes, `mCRL2` provides the allow-and-communicate mechanism. Once all processes have been specified (`Client`, `Machine`, `Baker`, and `Display`) the so-called initial process is given. We have chosen to analyze a typical situation of three clients in combination with one machine, one baker, and one display:

```
allow( {
  getTicket, showTicket, clarify, thankLeave,
    ...
  CM1, useDone, providingDone,
    ... },
comm( {
  requestMachine | readyToProvide -> CM1 ,
    ... },
Client(0,0,NotTaking,NotServed) ||
Client(0,0,NotTaking,NotServed) ||
Client(0,0,NotTaking,NotServed) ||
Mach(0,1,Available) ||
Baker(0,0,NotUsing,FinishingHelp) ||
Display(0,1,Offering) ) )
```

The crucial point is, the synchronized execution of the actions `requestMachine(n)` by `Client` and `readyToProvide(n)` by `Machine`, for the same value `n`, will be represented by the execution of the action `CM1(n)` of the overall system. On top of this, for all `n`, the action `CM1` is allowed to be executed, as mentioned in the list of allowed actions, but the action `requestMachine` and the action `readyToProvide` on their own are not, since they are deliberately missing from the list of allowed actions. Thus, a `requestMachine` or `readyToProvide` cannot happen alone, but combined into the action `CM1` only, provided the actions carry the same value for their parameter. Since, by the sum construction the client is willing to perform `requestMachine(m)` for each value of `m`, it can match the specific value for `m` offered by the machine in `readyToProvide(m)`. This way,

for this basic case, passing of parameter values from one process to another is achieved.

In general, a Paradigm model $(C_h)_{h\in H}$ will be encoded in mCRL2 as the parallel composition of $\#H$ processes, with $\#H$ the number of elements of the index set H. For a process C_h we have in its encoding, on the one hand, a parameter st of type Nat enumerating the set of states S_h and parameters $\mathtt{d}_1, \ldots, \mathtt{d}_{n_h}$ of properly chosen built-in or user-defined type to represent the extended state of C_h, and, on the other hand, parameters $\mathtt{ph}_1, \ldots, \mathtt{ph}_{n_h}$ for each of the roles, each of type specifically introduced for the roles. The actions of the processes are either local actions, from the respective action sets A, together with action corresponding to traps of the various roles of the components. With the combined use of allow and communication operators synchronization of traps can be enforced.

A local transition $\langle s_h, e_h, \psi_h\rangle_{h\in H} \xrightarrow{a} \langle s'_h, e'_h, \psi_h\rangle_{h\in H}$, say with $e_h = (e_{h,j})_{j=1}^{m_h}$, is easy to handle. We only need to verify that the transition $\langle s_{h_0}, e_{h_0}\rangle \xrightarrow{a} \langle s'_{h_0}, e'_{h_0}\rangle$ for the active component h_0 is admitted by the phases in ψ_{h_0} (see Definition 3). That other processes remain unchanged is implied by the interleaving of the processes. Thus, for each transition $\langle s_{h_0}, e_{h_0}\rangle \xrightarrow{a} \langle s'_{h_0}, e'_{h_0}\rangle$ in T_h we incorporate for mCRL2 process C_h the non-deterministic branch

```
( ( st == s_h ) &&
  ( d_1 == e_h,1 ) && ... && ( d_m_h == e_h,m(h) ) &&
    ( st in [ list of admitting phases role 1] ) && ... &&
      ( st in [ list of admitting phases role n(h) ] ) ) ->
        a . C_h(s'_h,e'_h,1,...,e'_h,m_h,ph_1,...,ph_n_h)
```

and add the action a to the set of actions of the allow operator enclosing the parallel composition of components.

A consistency rule $\gamma = (\, C_r(\Phi_r) : \varphi_r(e_r) \xrightarrow{\theta_r} \varphi'_r(e'_r)\,)_{r\in R}$, say with $e_r = (e_{r,j})_{j=1}^{m_r}$ and $e'_r = (e'_{r,j})_{j=1}^{m_r}$, is distributed over all components and roles involved. For each index r in R, we include a non-deterministic branch for process C_r and role Φ_r in the mCRL2 process C_r.

```
( ( st in [ states for trap theta_r ] ) &&
  ( d_1 == e_h,1 ) && ... && ( d_m_h == e_h,m_h ) &&
    ( ph_i(r) == phase_phi_r_of_Phi_r ) ) ->
      sum w_1:W_1 . ... . sum w_n:W_n .
        theta_r(w_1,...,w_n,expr_1,...,expr_n) .
          C_r(st,e'_r,1,...,e'_r,m_r,...,phase_phi'_r_of_Phi_r,...)
```

The summations sum w_1:W_1 to sum w_n:W_n abstract away the n-1 groups of variables of the components other the component C_r itself (although this cannot be read off from the notation above). Thus, of the variable groups w_1 to w_n only w_r is not bound by a summation. The expressions expr_1 to expr_n, built-up from standard constructs and possibly all of the variables in the n groups w_1 to w_n, are the expressions as occurring in the $\#R$ righthand-sides of the consistency rule. For a successful interaction it is required that all parties agree

on the values of the parameters and expressions involved. By taking the sum over all possible (potentially infinitely many) values the process `C_r` leaves it totally to the other components to decide on the values of their variables, if occurring at all. Moreover, if $\#R > 1$ we add the communication `theta_1 | ... | theta_#R -> gamma` to the communication operator `com` enclosing the parallel composition of components, but do not include any of the trap actions `theta_1,...,theta_#R`. In case $\#R = 1$, no communication is introduced, but the trap action will be allowed instead.

Some further caution needs to be put in place though, to deal with summations over infinite data types as possibly occurring in the encoding of the consistency rules. In the various analysis steps with the `mCRL2` toolset, in particular statespace generation, the tools may hang because of infinite branching. For Paradigm with data, in concrete situations, simplifications to the coding are applied for variables whose actual value is not used. There are two flavors of this: comparison of an expression involving the variable to an expression involving another (as for the ticket number of the client and the baker in lines 21, 24, and 27), or when the variable doesn't occur at all in the expressions at the righthand-side of the the consistency rule. As illustration of the latter situation, the encoding for the *Machine* process for the consistency rule (CM1) reads

```
( ( st in [ 0 ] ) && ( gt_ph == Available ) ) ->
  readyToProvide(m) . Machine(st,m,Unavailable)
```

where no abstraction of the variable `c` of `Client` is needed. The machine just provides a ticket number, in our modeling, independent of the actual value of `c`.

5 Concluding remarks

We have shown, with the IWIM model in mind, how the coordination modeling language Paradigm can be extended to deal with data. The present set-up is relatively liberal in the use of variables, although in concrete modeling situations a relatively small number of patterns of data flow among interacting components seem to suffice. Further investigation needs to reveal if this allows for a simplification of the consistency rule format and the associated closedness requirements both for sets of consistency rules as well as for the restriction on updates of variables within a trap.

Currently, for formal analysis using the `mCRL2` toolset, the encoding needs to be tailored to avoid infinite branching during statespace generation. The toolset provides a number of tools, e.g. `lpssumelm` and `lpsconstelm`, that manipulate intermediate artifacts (in so-called linear process specification or lps format) to reduce the specification, leading to smaller and hence more amenable verification problems. It is a topic of further research to develop sum elimination techniques specifically targeting the encoding of consistency rules discussed in this paper.

Application areas for Paradigm with data include the modeling of services, where both coordination and data play a prominent role, as well as the analysis of security protocols. Dissertational work by the second author is underway dealing with the modeling with Paradigm of anonymous networking and internet voting.

Acknowledgments We are grateful to Farhad Arbab for the many conversations on topics in coordination, computer science, research, and life in general and are looking forward to the undoubtedly many discussions still to come. We thank the anonymous reviewers for their helpful comments.

References

1. S. Andova, L.P.J. Groenewegen, and E.P. de Vink. Dynamic consistency in process algebra: From Paradigm to ACP. *Science of Computer Programming*, 76(8):711–735, 2011.
2. S. Andova, L.P.J. Groenewegen, and E.P. de Vink. Dynamic adaptation with distributed control in Paradigm. *Science of Computer Programming*, 94:333–361, 2014.
3. F. Arbab. The IWIM model for coordination of concurrent activities. In P. Ciancarini and C. Hankin, editors, *Proc. COORDINATION*, pages 34–56. LNCS 1061, 1996.
4. F. Arbab, I. Herman, and P. Spilling. An overview of Manifold and its implementation. *Concurrency – Practice and Experience*, 5(1):23–70, 1993.
5. C. Baier, M. Sirjani, F. Arbab, and J.J.M.M. Rutten. Modeling component connectors in Reo by constraint automata. *Science of Computer Programming*, 61(2):75–113, 2006.
6. R. Banach, F. Arbab, G.A. Papadopoulos, and J.R W. Glauert. A multiply hierarchical automaton semantics for the IWIM coordination model. *Journal of Univeral Computer Science*, 9(1):2–33, 2003.
7. M.M. Bonsangue, F. Arbab, J.W. de Bakker, J.J.M.M. Rutten, A. Secutella, and G. Zavattaro. A transition system semantics for the control-driven coordination language Manifold. *Theoretrical Computer Science*, 240(1):3–47, 2000.
8. S. Cranen, J.F. Groote, J.J.A. Keiren, F.P.M. Stappers, E.P. de Vink, W. Wesselink, and T.A.C. Willemse. An overview of the mCRL2 toolset and its recent advances. In N. Piterman and S.A. Smolka, editors, *Proc. TACAS 2013*, pages 199–213. LNCS 7795, 2013.
9. L. Groenewegen and E. de Vink. Operational semantics for coordination in Paradigm. In F. Arbab and C. Talcott, editors, *Proceedings Coordination 2002*, pages 191–206. Lecture Notes in Computer Science 2315, 2002.
10. J.F. Groote and M.R. Mousavi. *Modeling and Analysis of Communicating Systems*. MIT Press, 2014.
11. S.-S.T.Q. Jongmans and F. Arbab. Global consensus through local synchronization: A formal basis for partially-distributed coordination. *Science of Computer Programming*, 115-116:199–224, 2016.

A Complete mCRL2 specification of the bakery example

This appendix contains the mCRL2 code of the bakery example of Section 3.

```
sort

  %% role NeedTicket of Client
     NeedTicketPhase = struct NotTaking | Taking ;
  %% role NeedService of Client
     NeedServicePhase = struct NotServed| HavingTurn ;

  %% role GiveTicket of Machine
     GiveTicketPhase = struct Available | Unavailable ;

  %% role ShowNumber of Display
     ShowNumberPhase = struct Offering | Passive ;

  %% role NeedTurnNumber of Baker
     NeedTurnNumberPhase = struct NotUsing | Usage ;
  %% role NeedNextClient of Baker
     NeedNextClientPhase = struct FinishingHelp | StartingHelp | Helping ;

act

  %% Client actions
     getTicket ; showTicket, clarify, thankLeave : Nat ;
  %% Client traps
     requestMachine : Nat ; useDone ;
     requestBaker1, requestBaker2 : Nat ; ordered : Nat ;

  %% Machine actions
     incr, provide ;
  %% Machine traps
     readyToProvide : Nat ; providingDone ;

  %% Display actions
     lightUp, lightDown ;
  %% Display traps
     nextNumber : Nat ; ready ;

  %% Baker actions
     incrDisplay ; welcome, help, thankGreet : Nat ;
  %% Baker traps
     requestNextNumber : Nat ; done ;
     idle, started : Nat ; served : Nat ;


  %% interactions
     CM1 : Nat ;
     BD1 : Nat ;
     BC1, BC2, BC3 : Nat ;

proc
  Client(st:Nat, c:Nat, nt_ph:NeedTicketPhase, ns_ph:NeedServicePhase) =

  %% local STD

    %% 0 -> 1
    ( ( st == 0 ) && ( nt_ph in [ NotTaking ] ) && ( ns_ph in [ NotServed ] ) ) ->
      getTicket . Client(1,c,nt_ph,ns_ph) +

    %% 1 -> 2
    ( ( st == 1 ) && ( nt_ph in [ Taking ] ) && ( ns_ph in [ NotServed ] ) ) ->
      showTicket(c) . Client(2,c,nt_ph,ns_ph) +

    %% 2 -> 3
    ( ( st == 2 ) && ( nt_ph in [ NotTaking, Taking ] ) && ( ns_ph in [ HavingTurn ] ) ) ->
```

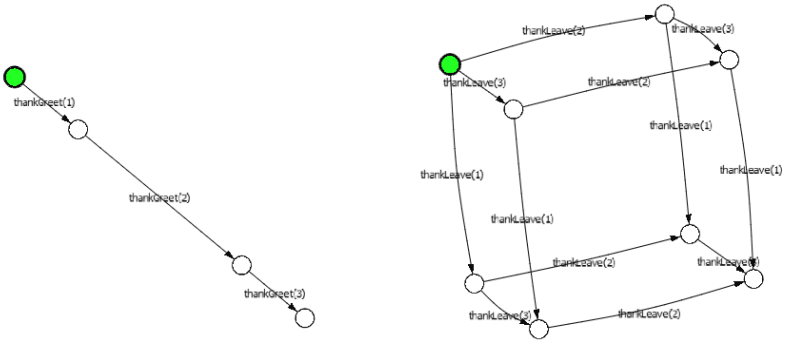
```
        clarify(c) . Client(3,c,nt_ph,ns_ph) +

    %% 3 -> 4
    ( ( st == 3 ) && ( nt_ph in [ NotTaking, Taking ] ) && (ns_ph in [ HavingTurn ] ) ) ->
      thankLeave(c) . Client(4,c,nt_ph,ns_ph) +


  %% role NeedTicket

    %% rule (CM1)
       ( ( st in [ 1 ] ) && ( nt_ph == NotTaking ) ) ->
         sum m:Nat . requestMachine(m) . Client(st,m,Taking,ns_ph) +

    %% rule (CM2)
       ( ( st in [ 2, 3, 4 ] ) && ( nt_ph == Taking ) ) ->
         useDone . Client(st,c,NotTaking,ns_ph) +


  %% role NeedService

    %% rule (BC1)
       ( ( st in [ 2 ] ) && ( ns_ph == NotServed ) ) ->
         requestBaker1(c) . Client(st,c,nt_ph,NotServed) +

    %% rule (BC2)
       ( ( st in [ 2 ] ) && ( ns_ph == NotServed ) ) ->
         requestBaker2(c) . Client(st,c,nt_ph,HavingTurn) +

    %% rule (BC3)
       ( ( st in [ 3, 4 ] ) && ( ns_ph == HavingTurn ) ) ->
         ordered(c) . Client(st,c,nt_ph,NotServed) ;

proc
  Machine( st:Nat, m:Nat, gt_ph:GiveTicketPhase ) =

  %% local STD

    %% 0 -> 1
    ( ( st in [ 0 ] ) && ( gt_ph in [ Unavailable ] ) ) ->
      provide . Machine(1,m,gt_ph) +

    %% 1 -> 0
    ( ( st == 1 ) && ( gt_ph in [ Available ] ) ) ->
      incr . Machine(0,m+1,gt_ph) +


  %% role GiveTicket

    %% rule (CM1)
       ( ( st in [ 0 ] ) && ( gt_ph == Available ) ) ->
         readyToProvide(m) . Machine(st,m,Unavailable) +

    %% rule (CM3)
       ( ( st in [ 1 ] ) && ( gt_ph == Unavailable ) ) ->
         providingDone . Machine(st,m,Available) ;

proc
  Baker(st:Nat, b:Nat, ntn_ph:NeedTurnNumberPhase, nnc_ph:NeedNextClientPhase) =

  %% local STD

    %% 0 -> 1
    ( ( st == 0 ) && ( ntn_ph in [ Usage ] ) && (nnc_ph in [ FinishingHelp ] ) ) ->
      incrDisplay . Baker(1,b,ntn_ph,nnc_ph) +

    %% 1 -> 2
    ( ( st == 1 ) && ( ntn_ph in [ Usage ] ) && (nnc_ph in [ StartingHelp ] ) ) ->
      welcome(b) . Baker(2,b,ntn_ph,nnc_ph) +
```

```
    %% 2 -> 3
    ( ( st == 2 ) && ( ntn_ph in [ Usage ] ) && (nnc_ph in [ Helping ] ) ) ->
      help(b) . Baker(3,b,ntn_ph,nnc_ph) +

    %% 3 -> 0
    ( ( st == 3 ) && ( ntn_ph in [ NotUsing ] ) && (nnc_ph in [ FinishingHelp ] ) ) ->
      thankGreet(b) . Baker(0,b,ntn_ph,nnc_ph) +

  %% role NeedTurnNumber

    %% rule (BD1)
       ( ( st in [ 0 ] ) && ( ntn_ph == NotUsing ) ) ->
         sum d:Nat . requestNextNumber(d) . Baker(st,d,Usage,nnc_ph) +

    %% rule (BD2)
       ( ( st in [ 3 ] ) && ( ntn_ph == Usage ) ) ->
         done . Baker(st,b,NotUsing,nnc_ph) +

  %% role NeedNextClient

    %% rule (BC1)
       ( ( st in [ 1 ] ) && ( nnc_ph == FinishingHelp ) ) ->
         idle(b) . Baker(st,b,ntn_ph,StartingHelp) +

    %% rule (BC2)
       ( ( st in [ 2 ] ) && ( nnc_ph == StartingHelp ) ) ->
         started(b) . Baker(st,b,ntn_ph,Helping) +

    %% rule (BC3)
       ( ( st in [ 3 ] ) && ( nnc_ph == Helping ) ) ->
         served(b) . Baker(st,b,ntn_ph,FinishingHelp) ;

proc
  Display( st:Nat, d:Nat, sh_ph:ShowNumberPhase ) =

  %% local STD

    %% 0 -> 1
    ( ( st in [ 0 ] ) && ( sh_ph in [ Passive ] ) ) ->
      lightDown . Display(1,d,sh_ph) +

    %% 1 -> 0
    ( ( st == 1 ) && ( sh_ph in [ Offering ] ) ) ->
      lightUp . Display(0,d,sh_ph) +

  %% role ShowingNumber

    %% rule (BD1)
       ( ( st in [ 0 ] ) && ( sh_ph == Offering ) ) ->
         nextNumber(d) . Display(st,d,Passive) +

    %% rule (BD3)
       ( ( st in [ 1 ] ) && ( sh_ph == Passive ) ) ->
         ready . Display(st,d+1,Offering) ;

init
  hide( {
    %% Client actions
       getTicket, showTicket, clarify, thankLeave,

    %% Machine actions
       incr, provide,

    %% Baker actions
       incrDisplay, welcome, help,
       %% thankGreet,
```

```
  %% Display actions
     lightUp, lightDown,

  %% interactions
     CM1, useDone, providingDone,
     BD1, done, ready,
     BC1, BC2, BC3 },

allow( {
  %% Client actions
     getTicket, showTicket, clarify, thankLeave,

  %% Machine actions
     incr, provide,

  %% Baker actions
     incrDisplay, welcome, help, thankGreet,

  %% Display actions
     lightUp, lightDown,


  %% interactions
     CM1, useDone, providingDone,
     BD1, done, ready,
     BC1, BC2, BC3 },

comm( {
  %% Client-Machine interaction
     requestMachine | readyToProvide -> CM1 ,
  %% Baker-Display interaction
     requestNextNumber | nextNumber -> BD1 ,
  %% Baker-Client interaction
     idle | requestBaker1 -> BC1 ,
     started | requestBaker2 -> BC2 ,
     served | ordered -> BC3 },

Client(0,0,NotTaking,NotServed) ||
Client(0,0,NotTaking,NotServed) ||
Client(0,0,NotTaking,NotServed) ||
Machine(0,1,Available) ||
Baker(0,0,NotUsing,FinishingHelp) ||
Display(0,1,Offering)
  ))) ;
```

B Reduced LTS

Labeled transition system for the specification of Appendix A with only actions `thankGreet` by the `Baker` process shown on the left. It validates that all three clients are served, assuming an atomic `welcome-help-thankGreet` sequences of actions, and are served in order of their tickets. On the right, the labeled transition system for the specification of Appendix A with only actions `thankLeave` by the `Client` processes shown. It validates that all three clients are served, but they can (and will) leave in any order.



Labeled transition system for the specification of Appendix A with only actions `showTicket` and `clarify` by the `Client` processes shown. It validates that all three clients can raise their ticket independently, since admitted local behavior can be executed asynchronously from other component behavior, but are served in order of their ticket.