Queue Merge: a Binary Operator for Modeling Queueing Behavior*

P.J.L. Cuijpers F.A.J. Koenders M.G.P. Pustjens
B.A.G. Senders P.J.A. van Tilburg P. Verduin

March 11, 2009

Abstract

We propose a process algebra QA, in which it is possible to describe a *queue* process. This process models a queue data structure in the same way as it is possible to model a bag data structure and a stack data structure using other process algebras. Furthermore we give a proof sketch that every process in this algebra is branching bisimilar to a regular process communicating with this queue. We try to establish a link between processes in QA and languages generated by queue grammars, but fail to map either of those to the other, conjecturing that no algebraic operator can exist which directly models the class of grammars that use a queue.

Keywords: process algebra, automata theory, queue data structure, queue automata, Chomsky hierarchy

1 Introduction

In the field of automata theory, different classes of languages are studied which are generated by communication between a finite automaton and some data structure. For example, a finite automaton communicating with a *tape* is known as a *Turing machine*, a finite automaton communicating with a *stack* is known as a *pushdown automaton*, and a finite automaton communicating with a *bag* (or *multiset*) is known as a *parallel pushdown automaton* [16].

In the field of process algebra, different classes of processes are studied which are generated by studying different combinations of operators in the algebra. The minimal algebra is often taken to consist of action prefix, alternative composition, and a zero and unit element for alternative composition. Then, sequential process algebra is the minimal algebra extended with sequential composition, basic parallel process algebra is the minimal algebra extended with parallel composition, and communicating process algebra is the minimal algebra extended with parallel composition and the possibility to model synchronization. We build upon the minimal and communicating process algebras in this paper.

In linking the field of process algebra and that of automata theory, several results have recently been established. For example, it has been shown that a stack can be modeled as a finite recursive specification in the sequential process algebra, and that every finite recursive specification in the sequential process

*This report was written as part of the Formal Methods Seminar 2008/2009, course 2IF95.

algebra is contrasimilar to a finite recursive specification over basic process algebra (a *regular process*) communicating with a stack (i.e. a *pushdown process*) [6]. Similarly, a bag or multiset can be modeled as a finite recursive specification in basic parallel process algebra, and every finite recursive specification in the basic parallel process algebra is branching bisimilar to regular process communicating with a bag [7].

In this report, inspired by these results, we research the question which process algebraic operator would yield a similar connection with the *queue* data structure. It is a well-known result by Post [17] that a finite automaton communicating with a queue can produce the same languages as a Turing Machine, and we are curious to see what the expressiveness would be of an operator that is able to model a queue. From a result by Bergstra and Tiuryn [9] we know that the standard operators of process algebra (alternative-, sequential-, and parallel composition without synchronization) are not sufficient to model a queue, but this paper also provides a unary operator that exploits additional structure on the set of actions that is able to model a queue. Taking this unary operator as a starting point, we construct a binary operator called the *queue merge* operator, that produces a queue in a similar way as that stack and bag were produced in [7, 6]. Furthermore, we prove that every *basic queueing process* is branching bisimilar to some regular process communicating with a queue. Although not very surprising, this result does not trivially follow from Post's result, since branching bisimulation is a stronger equivalence than language equivalence.

Sadly, after establishing this link between the queue merge operator and a *queue automaton*, the analogy seems to end. In particular, when studying languages generated by data structures, it is also common to study grammars that make use of a data structure (see for example [6] and [7]). The link between a grammar that uses a stack (a context free grammar) and sequential process algebra, is quite trivial (see [6]), and the same goes for the link between a grammar that uses a bag and basic parallel process algebra (see [7]). However, the link between a grammar that uses a queue [11] and basic queueing processes, does not exist. We have examples of a process that can be described using the queue merge operator that cannot be described using the queue grammar, and we have a conjecture that the catastrophe process, which is describable as a queue grammar, cannot be described using a basic queueing process.

The structure of this report is as follows: we start with a formal introduction to regular processes and communicating processes in Section 2, and continue in Section 3 by adding the queue merge operator. In Section 4 we show that the queue merge is able to model a queue using the same basic formula that was used to model a stack in [6] and a bag in [7]. In Section 5 we give a sketch of the proof that every basic queueing process is branching bisimilar to some regular process communicating with a queue. In Section 6 we discuss the creation of languages using grammars and we give the counterexamples that show that the link between the queue merge and the queue grammar is broken. Finally, in Section 7 we conclude this report and give recommendations for future work.

2 Regular and Communicating Processes

Before we introduce basic queueing processes we first consider the notion of a regular process and its relation to regular language theory. We start with the

definition of transition systems from process theory. A finite transition system can be thought of as a non-deterministic finite automaton. In order to have a complete analogy, the transition systems we consider have a subset of states marked as a final state.

Definition 1. *A* transition system *T is a quintuple $(\mathcal{S}, \mathcal{A}, \rightarrow, \uparrow, \downarrow)$ where:*

- *$\mathcal{S}$ is a set of states,*
- *$\mathcal{A}$ is an alphabet,*
- *$\rightarrow \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ is the set of transitions or steps,*
- *$\uparrow \in \mathcal{S}$ is the initial state,*
- *$\downarrow \subseteq \mathcal{S}$ is a set of final states.*

For a transition $(s, a, t) \in \rightarrow$ we write $s \xrightarrow{a} t$. For a final state $s \in \downarrow$ we write $s\downarrow$. A transition system is called *finite* when both sets $\mathcal{S}$ and $\mathcal{A}$ are finite.

In automata theory a regular language can be given by a right-linear grammar. Similarly, a grammar in process theory is called a recursive specification; a set of recursive equations over a set variables $\mathcal{V}$. Then a right-linear grammar coincides with a recursive specification over a finite set of variables in the Minimal Algebra MA. In this paper we adopt the process algebra notation as in [2, 5].

Definition 2. *The signature of Minimal Algebra MA is as follows:*

- *The constant* **0***; denoting inaction, a deadlock state; other names are δ or stop.*
- *The constant* **1***; termination, a final state; other names are ε, skip or the empty process.*
- *For each element of the alphabet $\mathcal{A}$ there is a unary operator $a\_$ called action-prefix; a term $a.x$ will execute the elementary action a and then proceed as x.*
- *The binary operator $+$ called alternative composition; a term $x + y$ will either execute x or execute y, a choice will be made between the alternatives.*

The constants **0** and **1** are needed to denote transition systems with a single state and no transitions. The constant **0** denotes a single state that is not a final state, while **1** denotes a single state that is also a final state.

Definition 3. *Let $\mathcal{V}$ be a set of variables. A* recursive specification *over $\mathcal{V}$ with initial variable S, with $S \in \mathcal{V}$ is a set of equations of the form $X = t_X$, exactly one for each $X \in \mathcal{V}$, where each right-hand side t_X is a term over some signature, possibly containing elements of $\mathcal{V}$. A recursive specification is called* finite *if $\mathcal{V}$ has finitely many elements.*

Definition 4. *A recursive specification is called* guarded *if each equation is guarded. An equation is guarded if each occurrence of a variable $V \in \mathcal{V}$ in t_X is guarded. A variable $V \in \mathcal{V}$ is guarded if it occurs as the operand of a action-prefix operator. E.g. in $a.X$, where $a \in \mathcal{A}$ and $X \in \mathcal{V}$, X is guarded.*

A finite guarded recursive specification over MA can be seen as a right-linear grammar. Therefore each finite transition system corresponds directly to a finite recursive specification over MA, using a variable for every state. To go from a term over MA to a transition system, we use *structural operational semantics* [1], with rules given in Table 1.

Table 1: Operational rules for MA and recursion ($a \in \mathcal{A}, X \in \mathcal{V}$)

$$1\frac{}{\mathbf{1}\downarrow} \qquad 2\frac{}{a.x \xrightarrow{a} x}$$

$$3\frac{x \xrightarrow{a} x'}{x + y \xrightarrow{a} x'} \qquad 4\frac{y \xrightarrow{a} y'}{x + y \xrightarrow{a} y'} \qquad 5\frac{x\downarrow}{x + y\downarrow} \qquad 6\frac{y\downarrow}{x + y\downarrow}$$

$$7\frac{t_X \xrightarrow{a} x \quad X = t_X}{X \xrightarrow{a} x} \qquad 8\frac{t_X\downarrow \quad X = t_X}{X\downarrow}$$

Because we model in this paper the interaction between a regular process and a queue, we need to introduce communication by synchronization. Therefore we introduce the *Communication Algebra* CA, which extends MA with the *parallel composition* operator $\|$, and has a notion of communication. From the viewpoint of a process there are two types of communication, a *receive action* and a *send action*; a receiving action is denoted as $?d$ and a sending action as $!d$. In this paper we use a particular communication function, that only synchronizes $!d$ and $?d$ (for the same $d \in D$). The result of such synchronization is denoted as $⁉d$. CA introduces two new operators: the *encapsulation operator* $\partial_*(\_)$, which blocks actions $!d$ and $?d$, and the abstraction operator $\tau_*(\_)$, which turns all actions $⁉d$ into the internal action τ. The operational rules for CA are given in Table 2.

A finite axiomatization of transition systems of CA modulo rooted branching bisimulation uses the auxiliary operators $\_ \between \_$ and $\_ \mid \_$ [10, 15]. The axioms can be found in Table 3, for the explanation we refer to the literature [5]. The given equational theory is sound and ground-complete for the model of transition systems modulo rooted branching bisimulation [14].

Now that we have introduced MA and CA, it is possible to define the queueing algebra QA, which is built upon MA; this is described in the next section.

3 Basic Queueing Processes

In [6] it was established that context-free processes can be given by guarded recursive specifications over the Sequential Algebra SA, which extends MA with the sequential composition operator $\_ \cdot \_$. In [7] parallel processes were constructed, a superset of the basic parallel processes, by guarded recursive specifications over the Communication Algebra CA, which extends MA with the parallel composition operator $\_ || \_$. Within SA the stack was considered the most basic sequential process, within CA the corresponding data structure is

Table 2: Operational rules for CA ($a \in \mathcal{A}$)

$$9\ \frac{x \xrightarrow{a} x'}{x \parallel y \xrightarrow{a} x' \parallel y} \qquad 10\ \frac{y \xrightarrow{a} y'}{x \parallel y \xrightarrow{a} x \parallel y'} \qquad 11\ \frac{x\!\downarrow \quad y\!\downarrow}{x \parallel y\!\downarrow}$$

$$12\ \frac{x \xrightarrow{?d} x' \quad y \xrightarrow{!d} y'}{x \parallel y \xrightarrow{⁉d} x' \parallel y'} \qquad 13\ \frac{x \xrightarrow{!d} x' \quad y \xrightarrow{?d} y'}{x \parallel y \xrightarrow{⁉d} x' \parallel y'}$$

$$14\ \frac{x \xrightarrow{a} x' \quad a \neq !d, ?d}{\partial_*(x) \xrightarrow{a} \partial_*(x')} \qquad 15\ \frac{x\!\downarrow}{\partial_*(x)\!\downarrow}$$

$$16\ \frac{x \xrightarrow{⁉d} x'}{\tau_*(x) \xrightarrow{\tau} \tau_*(x')} \qquad 17\ \frac{x \xrightarrow{a} x' \quad a \neq ⁉d}{\tau_*(x) \xrightarrow{\tau} \tau_*(x')} \qquad 18\ \frac{x\!\downarrow}{\tau_*(x)\!\downarrow}$$

Table 3: Equational theory of CA ($a \in \mathcal{A}$)

$x \parallel y$	$= x \between y + y \between x + x \mid y$	$a.(\tau.(x+y)+x)$	$= a.(x+y)$
$\mathbf{0} \between x$	$= \mathbf{0}$	$x \mid y$	$= y \mid x$
$\mathbf{1} \between x$	$= \mathbf{0}$	$x \parallel \mathbf{1}$	$= \mathbf{1}$
$a.x \between y$	$= a.(x \parallel y)$	$1 \mid x + \mathbf{1}$	$= \mathbf{1}$
$(x+y) \between z$	$= x \between z + y \between z$	$(x \parallel y) \parallel z$	$= x \parallel (y \parallel z)$
$\mathbf{0} \mid x$	$= \mathbf{0}$	$(x \mid y) \mid z$	$= x \mid (y \mid z)$
$(x+y) \mid z$	$= x \mid z + y \mid z$	$(x \between y) \between z$	$= x \between (y \parallel z)$
$\mathbf{1} \mid \mathbf{1}$	$= \mathbf{1}$	$(x \mid y) \between z$	$= x \mid (y \between z)$
$a.x \mid \mathbf{1}$	$= \mathbf{0}$	$x \between \tau.y$	$= x \between y$
$!d.x \mid ?d.y$	$= ⁉d.(x \parallel y)$	$x \mid \tau.y$	$= \mathbf{0}$
$a.x \mid b.y$	$= \mathbf{0}$ if $\{a,b\} \neq \{!d, ?d\}$		
$\partial_*(\mathbf{0})$	$= \mathbf{0}$	$\tau_*(\mathbf{0})$	$= \mathbf{0}$
$\partial_*(\mathbf{1})$	$= \mathbf{1}$	$\tau_*(\mathbf{1})$	$= \mathbf{1}$
$\partial_*(?d.x)$	$= \partial_*(?d) = \mathbf{0}$	$\tau_*(⁉d.x)$	$= \tau.\tau_*(x)$
$\partial_*(a.x)$	$= a.\partial_*(x)$ if $a \notin \{!d, ?d\}$	$\tau_*(a.x)$	$= a.\tau_*(x)$
$\partial_*(x+y)$	$= \partial_*(x) + \partial_*(y)$	$\tau_*(x+y)$	$= \tau_*(x) = \tau_*(y)$

the bag. Now we will extend MA with a new operator, the *queue merge* operator $\between$, and call the resulting algebra the *Queueing Algebra* (QA). This new operator was inspired by the queue operator as it was introduced in [9]. We define the *Basic Queueing Processes* (BQP) as the bisimulation equivalence class of the transition system generated by a finite guarded recursive specification over QA.

In order to construct a transition system from a term over process algebra QA, we need to know the operational semantics for QA. To do this, we first need two predicates, functions from processes to the Booleans, $\neg!\colon \mathbf{P} \to \mathbb{B}$ and $\neg?\colon \mathbf{P} \to \mathbb{B}$. These predicates check if there is no send, or respectively no receive action enabled in the given process. We call these predicates *no-send* and *no-receive*.

Definition 5. *We define* $\neg!$ *and* $\neg?$ *as follows:*

$$\begin{array}{lcl@{\qquad}lcl}
\neg?(\mathbf{0}) & = & \mathit{True} & \neg!(\mathbf{0}) & = & \mathit{True} \\
\neg?(\mathbf{1}) & = & \mathit{True} & \neg!(\mathbf{1}) & = & \mathit{True} \\
\neg?(x+y) & = & \neg?(x) \wedge \neg?(y) & \neg!(x+y) & = & \neg?(x) \wedge \neg?(y) \\
\neg?(?a.p) & = & \mathit{False} & \neg!(?a.p) & = & \mathit{True} \\
\neg?(!a.p) & = & \mathit{True} & \neg!(!a.p) & = & \mathit{False}
\end{array}$$

Now we can define the SOS rules as follows. Consider the term $x \between y$. Whenever x can do a receive step to x', $x \between y$ can do a receive step to $x' \between y$. The dual holds for a send step on y. The left term of a queue merge (x in the example) can only do a send action whenever the right operand (y) cannot. Again the dual holds for the right operand with the receive action. Finally, the term $x \between y$ can only terminate if both the terms x and y can terminate. This behaviour is formalized using structural operational semantics in Table 4. It follows from the shape of the elements in these SOS rules that bisimulation is a congruence for the operators in QA (see [13]).

Table 4: Structural Operational Semantics rules for QA ($?a, !a \in \mathcal{A}$)

$$19\ \frac{x \xrightarrow{?a} x'}{x \between y \xrightarrow{?a} x' \between y} \qquad 20\ \frac{y \xrightarrow{!a} y'}{x \between y \xrightarrow{!a} x \between y'}$$

$$21\ \frac{x \xrightarrow{!a} x' \quad \neg!(y)}{x \between y \xrightarrow{!a} x' \between y} \qquad 22\ \frac{y \xrightarrow{?a} y' \quad \neg?(x)}{x \between y \xrightarrow{?a} x \between y'}$$

$$23\ \frac{x\!\downarrow \quad y\!\downarrow}{x \between y\!\downarrow}$$

We need to extend QA with additional operators to be able to obtain a finite axiomatization. The operators to be added are the *left queue merge*: $\between\!\!\backslash$ and the *right queue merge*: $/\!\!\between$. These were chosen in analogy to the left and right merge from the decomposition of the parallel composition operator. These two operators require additional SOS rules, which are presented in Table 5.

The left queue merge operator expresses that only an action at the left operand may be performed. This can either be a receive or a send action, depending on the right operand. The dual holds for the right queue merge. Termination is the same for the left- and right-queue merge; both $x \between\!\!\backslash y$ and $x /\!\!\between y$ can terminate if x and y can terminate. Note again that the shape of these SOS rules imply congruence (again, see [13]).

Axioms Our finite axiomatization of QA, using the auxiliary operators $\between\!\!\backslash$ and $/\!\!\between$, can be found in Table 6. Axiom MQ expresses that the queue merge of x and y is syntactically equivalent to the alternative composition of the left queue merge of x and y and the right queue merge of x and y. Axioms with the LMQ-prefix concern terms with the left queue merge operator. Dual axioms exist

Table 5: Additional SOS rules for the axiomatization of QA ($?a, !a \in \mathcal{A}$)

$$24\ \frac{x \xrightarrow{?a} x'}{x \between y \xrightarrow{?a} x' \between y} \qquad 25\ \frac{y \xrightarrow{!a} y'}{x \between y \xrightarrow{!a} x \between y'}$$

$$26\ \frac{x \xrightarrow{!a} x' \quad \neg!(y)}{x \between y \xrightarrow{!a} x' \between y} \qquad 27\ \frac{y \xrightarrow{?a} y' \quad \neg?(x)}{x \between y \xrightarrow{?a} x \between y'}$$

$$28\ \frac{x\downarrow \quad y\downarrow}{x \between y\downarrow} \qquad 29\ \frac{x\downarrow \quad y\downarrow}{x \between y\downarrow}$$

for the right queue merge; these are prefixed with RMQ. The given theory is a sound and ground-complete axiomatization of the model of transition system modulo bisimulation [5, 8]. Proofs of the soundness of a number of axioms is included in Appendix A.

Table 6: Equational theory of QA

MQ $\quad x \between y = x \between y + x \between y$

LMQ1	$\mathbf{0} \between x$	$= \mathbf{0}$	RMQ1	$x \between \mathbf{0}$	$= \mathbf{0}$
LMQ2	$\mathbf{1} \between \mathbf{0}$	$= \mathbf{0}$	RMQ2	$\mathbf{0} \between \mathbf{1}$	$= \mathbf{0}$
LMQ3	$\mathbf{1} \between \mathbf{1}$	$= \mathbf{1}$	RMQ3	$\mathbf{1} \between \mathbf{1}$	$= \mathbf{1}$
LMQ4	$\mathbf{1} \between (x + y)$	$= \mathbf{1} \between x + \mathbf{1} \between y$	RMQ4	$(x + y) \between \mathbf{1}$	$= x \between \mathbf{1} + y \between \mathbf{1}$
LMQ5	$\mathbf{1} \between \alpha.x$	$= \mathbf{0} \quad \alpha \in \{?a, !a\}$	RMQ5	$\alpha.x \between \mathbf{1}$	$= \mathbf{0} \quad \alpha \in \{?a, !a\}$
LMQ6	$?a.x \between y$	$= ?a.(x \between y)$	RMQ6	$x \between !a.y$	$= !a.(x \between y)$
LMQ7	$!a.x \between (!b.y + z)$	$= \mathbf{0}$	RMQ7	$(?a.x + y) \between ?b.z$	$= \mathbf{0}$
LMQ8	$\neg!(y) \Rightarrow !a.x \between y$	$= !a.(x \between y)$	RMQ8	$\neg?(x) \Rightarrow x \between ?a.y$	$= ?a.(x \between y)$
LMQ9	$(x + y) \between z$	$= x \between z + y \between z$	RMQ9	$x \between (y + z)$	$= x \between y + x \between z$

SQ1	$x \between \mathbf{1}$	$= x$
SQ2	$\mathbf{1} \between x$	$= x$
SQ3	$(x \between y) \between z$	$= x \between (y \between z)$
SQ4	$(x \between y) \between z$	$= x \between (y \between z)$
SQ5	$x \between (y \between z)$	$= (x \between y) \between z$

Axiom LMQ1 expresses that $\mathbf{0}$ is a left zero element for the left queue merge. When $\mathbf{1}$ is the left operand of the left queue merge, we have four different cases depending on the right operand. First, when the right operand is a $\mathbf{0}$ or $\alpha.x$ (where α can be a $?a$ or a $!a$ action), we end up in deadlock. If the right operand is $\mathbf{1}$, the term is equivalent with $\mathbf{1}$ and if the right operand is $x + y$, then it is equivalent with the alternative composition of $\mathbf{1} \between x$ and $\mathbf{1} \between y$. Axiom LMQ6 expresses that we can always do a receive action in the left operand, followed by the queue merge of the remaining actions of the left and right operand. Axioms LMQ7 and LMQ8 express that we are only allowed to do a send action in the

left operand, if the right operand cannot do a send action. Finally, axiom LMQ9 expresses that the left queue merge distributes over the alternative composition from the right.

Furthermore we have some general axioms prefixed with SQ. The first two, SQ1 and SQ2, express that $\mathbf{1}$ is a unit element of the queue merge. Axiom SQ3 expresses that the queue merge operator is associative. Finally, axiom SQ4 expresses that the left queue merge of a left queue merge is equivalent with the left queue merge of the leftmost component and the queue merge of the middle and last component. Axiom SQ5 is the dual of SQ4 for the right queue merge.

4 Modeling a Queue

Now that we have defined the operational semantics and equational theory for the queue merge operator, we can use this to construct a *guarded recursive specification* for the queue; recall the definition of guardedness (Definition 4). The reason we restrict ourselves to guarded recursive specifications is that each guarded recursive specification has a unique solution in the transition system model [10, 8].

Theorem 4.1. *It is possible using the axioms to bring any guarded recursive specification into* Greibach Normal Form *[4], that is, for every variable $X \in \mathcal{V}$ into the following form:*

$$X = \sum_{i \in I_X} a_i.\xi_i \ (+\mathbf{1}) \tag{1}$$

Here I_X is a finite index set, and ξ_i is defined as the queue merge over a sequence of variables, i.e., $X_{i_1} \between X_{i_2} \between \ldots \between X_{i_n}$.

This way, each right hand side of an equation can be written as a sum over a number of terms indexed by a finite set I_X. The empty sum is denoted by $\mathbf{0}$, and each term is of the form $\mathbf{1}$ or $a_i.\xi_i$. Note that action a_i can either be a send or a receive action. For a recursive specification in Greibach Normal Form, every state of the transition system is given by a sequence of variables. A proof of Theorem 4.1 is included in Appendix B.

An example process we have investigated is a *queue*. Suppose we have a finite data set D, we define the following actions $\mathcal{A}$, for each $d \in D$:

- $?d$: the receive action, i.e. put datum d on the back of the queue.
- $!d$: the send action, i.e. remove datum d from the front of the queue.

In line with the previous papers on the bag and the stack [7, 6], we define a queue by the following recursive specification:

$$Q = \sum_{d \in D} ?d.(Q \between !d.\mathbf{1}) + \mathbf{1} \tag{2}$$

In order to see that the above process Q indeed defines a queue, we define a process Q_σ denoting a queue where $\sigma \in D^*$ denotes the content of the queue. We define the empty queue as follows.

$$Q_\varepsilon = \sum_{d \in D} ?d.Q_d \tag{3}$$

The above equation models a queue with no contents that can only receive a datum d resulting in a queue with only datum d on it. The non-empty queue, with a datum e on front followed by other data ξ, can be defined as follows.

$$Q_{e\xi} = \sum_{d \in D} ?d.Q_{e\xi d} + !e.Q_{\xi} \tag{4}$$

This equation models the choice between receiving a new datum d, whereby it is added at the back of the queue, or sending the front element from the queue. When the front element is sent, it is removed from the queue, leaving the other data on it without changing the order of the elements. We use RSP to prove that our definition of a queue matches the definition of a queue with data as given above. This proof is included in Appendix C.

Forgetful queue A process in the queueing algebra QA can only terminate if all its components can terminate. In our definition of a queue, given by the process Q in Equation 2 above, this is not possible. Therefore we want to adapt the definition of the queue, such that a component has a termination option if and only if the corresponding variable has a termination option. In order to do that, we introduce the concept of transparent variables.

We now restrict ourselves to a setting where any data item is a recursive specification variable. We call a variable *transparent*[1] if its equation has a $\mathbf{1}$-summand. Furthermore, we use $\mathcal{V}^{+\mathbf{1}}$ to denote the set of transparent variables of a basic queueing process P. Similarly we denote the set of all variables of P with a summand with a receive, respectively send, action with $\mathcal{V}^{+?}$ and $\mathcal{V}^{+!}$. Note that a process variable can have both send and receive actions, so $\mathcal{V}^{+?}$ and $\mathcal{V}^{+!}$ do not have to be disjoint. Finally, we have a set of actions $\mathcal{A} = \mathcal{A}_? \cup \mathcal{A}_!$, with $\mathcal{A}_? \cap \mathcal{A}_! = \emptyset$, where $\mathcal{A}_?$ is the set of receive actions and $\mathcal{A}_!$ is the set of send actions.

Using this notion of transparent variables we can define the *forgetful queue*, which can terminate if and only if all variables on the queue can terminate. The forgetful queue is given by the following recursive specification, where we use subscript notation to denote when a certain variable is forgetful. We use the notation $(+\mathbf{1})_{X \in \mathcal{V}^{+\mathbf{1}}}$ to denote that if a variable X is in the set of transparent variables $\mathcal{V}^{+\mathbf{1}}$, then it has a $+\mathbf{1}$ summand.

$$Q = \sum_{X \in D} ?X.(Q \between (!X.\mathbf{1}\ (+\mathbf{1})_{X \in \mathcal{V}^{+\mathbf{1}}}) + \mathbf{1} \tag{5}$$

This forgetful queue can then be used, in parallel with a regular process, to bisimulate any other process in QA. We give a proof sketch of this in the next section.

5 Queue Automaton Proof Sketch

In this section we provide the sketch of a proof that every process in BQP is branching bisimilar to some regular process communicating with a queue. The proof is important to show that there exists for every term over QA a queue

[1] The concepts of transparency and forgetfulness are derived from their counterparts in the bag [7] and stack processes [6].

automaton which accepts it. So, for any process P in BQP we want to find a regular process R such that

$$P = \tau_*(\partial_*(R \parallel Q_\sigma)) \tag{6}$$

where Q_σ is a state of process Q (as defined by Equation 5 in Section 4). Without loss of generality we assume that P is given in Greibach Normal Form (see Theorem 4.1), and so has a structure as defined in Equation 1. For the queue process, we take the forgetful queue as introduced in the previous section. This leads to the following theorem:

Theorem 5.1. *For every process P in BQP there exists a process R given by a finite guarded recursive specification over MA such that $P = [R \parallel Q_\sigma]_*$ for some state Q_σ of the queue. From here on, $[p]_*$ is used as shorthand notation for $\tau_*(\partial_*(p))$.*

Proof sketch In the following text we introduce a *control* process which simulates process P. This process works in a similar fashion as the control processes in [7, 6]. Our control process works by repeatedly taking a data item from the queue and putting it back, while offering some options. It finds the data item corresponding to the first (respectively last) process variable with receive (respectively send) actions on the queue and offers those receive (or send) actions as a choice, or it can rotate the queue back to an initial state. The result of this is that the control process does not make a choice itself, and is always able to return to a previous state. In this way, any state of $[R \parallel Q_\sigma]_*$ can also be simulated by P.

We first introduce a number of equations used to obtain the desired process R, with an intuition as to the correctness of these equations. A formal proof of one of the equations is given as an example further on; proofs of the others are omitted.

After showing how R is obtained we show how lists of elements are added to the queue using Push and the FFWD equation which is key to rotating the queue. Then the Ctrl process is introduced and after that the equations First and Last which are used by the control process. Lastly, the correctness of the FFWD process is proven.

The data-set D we use for our solution – i.e., the data items to be put on the queue – are the set of variables $\mathcal{V}$ of P, supplemented with the special markers \$, γ and λ. These markers are not process variables, but are elements of the dataset used for bookkeeping purposes. To allow correct termination these markers are all defined to be in $\mathcal{V}^{+1}$. Their function is explained further on.

Let E be a *finite* recursive specification of P given in Greibach Normal Form, so that for every variable $X \in \mathcal{V}$ of the specification E, I_X being its index set, X is defined as follows:

$$X = \sum_{i \in I_X} a_i.\xi_i \ (+\mathbf{1})_{X \in \mathcal{V}^{+1}} \tag{7}$$

Here the right-hand side of the equation consists of a number of summands, each of which has an index i from the (finite) index set I_X, and corresponding actions a_i – which are either receive actions $?a_i$ or send actions $!a_i$ – and ξ_i, a queue merge composition of process variables s.t. $\xi_i = X_{i_1} \between X_{i_2} \between \ldots \between X_{i_n}$.

Now let F be a finite recursive specification. The way F is modeled, with a central Ctrl equation and a number of Push equations, is based on similar notions as present in [7, 6].

If X is a process variable in E, then we want to mimic execution of X by two variables in F called $\widehat{X_?}$ and $\widehat{X_!}$, where $[\widehat{X_?} + \widehat{X_!} \parallel Q_\$]_*$ is bisimilar to X. These variables are defined as follows:

$$\widehat{X_?} = \left(\sum_{i \in I_X \wedge a_i \in \mathcal{A}_?} a_i.\mathrm{Push}(\xi_i) \right) + \mathrm{Push}(X) \tag{8}$$

$$\widehat{X_!} = \left(\sum_{i \in I_X \wedge a_i \in \mathcal{A}_!} a_i.\mathrm{Push}(\xi_i) \right) + \mathrm{Push}(X) \tag{9}$$

If and only if it is possible to choose to execute action $?a_i$ of X in E, then similarly it is possible to execute action $?a_i$ of $\widehat{X_?}$ in F. Process E is then in a state where it should execute ξ_i. In the same way, it is possible to execute $!a_i$ of $\widehat{X_!}$ if and only if it is possible to execute $!a_i$ of X in E. Executing ξ_i is mimicked in F by pushing all variables in ξ_i onto the queue. Later on they can be read one by one and executed. Pushing them onto the queue is modeled by the Push equations (see Equations 10 and 11); as can be seen from Equations 8 and 9, it is also possible in F to *not* execute the desired action, but instead push X itself onto the queue. This is to make sure that R is not the process that initiates an action; since all internal actions (i.e., communications with the queue) will become τ-actions when taken through the encapsulation function τ_*, it is important that any choice made by taking such an action can be reverted, so that the original process X is not "forced" to take that action as well.

The Push equations work in the following way:

$$\mathrm{Push}(\emptyset) = \mathrm{FFWD} \tag{10}$$

$$\mathrm{Push}(X\xi) = !X.\mathrm{Push}(\xi) \tag{11}$$

Here X is the variable that is currently at the beginning of the original sequence ξ_i and ξ is the remaining sequence after removing X from it.

After all variables have been pushed, the FFWD equation comes into play. The reader will have noticed that we have started out with $\widehat{X_?} + \widehat{X_!}$ in parallel with $Q_\$$, a queue containing only the item \$. This symbol is the so-called *end-of-queue marker*, which is used to signify the end of the queue in its "unrotated" state, and it is always present, even if the queue is empty – that is why we started with only \$ on the queue. We have not rotated the queue yet, but this will happen in other equations, and we need a way to keep track of what the original configuration was; because as we do not want FFWD to make the actual choice of performing an action or not, it should always be possible to return to the previous state. The marker \$ is used to know when to terminate the FFWD process.

Now the function of the FFWD equation is to rotate the queue, by iteratively reading and writing back data elements, so that when it is finished, the end-of-queue marker is actually at the end of the queue. Without this marker the FFWD process would not be able to determine when to stop. Rotating is done as follows:

$$\mathrm{FFWD} = \left(\sum_{X \in \mathcal{V}} ?X.!X.\mathrm{FFWD} \right) + ?\$.!\$.\mathrm{Ctrl} \tag{12}$$

As can be seen from the FFWD definition, after it is done rotating the queue, the Ctrl equation is entered. Note that the bisimulation of X with $[\widehat{X_?} + \widehat{X_!} \parallel Q_\$]_*$ can still execute ξ_i in F. Since after rotating, the variables in ξ_i have been pushed onto the queue. The correct options from ξ_i to offered are simulated by First, Last and $\mathbf{1}$.

$$\mathrm{Ctrl} = \mathrm{First} + \mathrm{Last} + \mathbf{1} \tag{13}$$

First of all, in E, it is possible for ξ_i to terminate if all its queue components can terminate. Now in F, Ctrl can always terminate, and the queue which is parallel to it can terminate if and only if all the items on the queue have a $\mathbf{1}$-summand, which is true if their actual specifications have a $\mathbf{1}$-summand; so both can terminate under the same conditions.

Now in $\xi_i = X_{i_1} \between X_{i_2} \between \ldots \between X_{i_n}$, only the leftmost process variable with a receive action (the X_{i_j} for which $X_{i_j} \in \mathcal{V}^{+?}$ and $\forall_{k<j} X_{i_k} \notin \mathcal{V}^{+?}$) and the rightmost variable with a send action (the X_{i_j} for which $X_{i_j} \in \mathcal{V}^{+!}$ and $\forall_{k>j} X_{i_k} \notin \mathcal{V}^{+!}$) are allowed to perform an action. This behaviour is mimicked respectively by First, which only allows the first process variable with a receive action to be executed, and Last, which only allows the last process variable with a send action to be executed.

Equation First works by rotating the queue until the first variable with a receive action is found. This is done by repeatedly reading an element X off the queue, and then immediately writing it back to the queue (if it does not have a receive action) or removing it from the queue and executing the corresponding $\widehat{X_?}$-process for the found variable X (if it does have a receive action). If this happens, we are again in the same state as when entering the $\widehat{X_?}$-process before, except that the queue can now be in an arbitrary rotated state, with possibly a number of other variables still on the queue.

It is also possible in First to read the end-of-queue marker; this means that no receive actions have been found, so the process returns to Ctrl, in which case it is in the same state as it was before having entered First.

$$\mathrm{First} = \left(\sum_{X \in \mathcal{V}} ?X.((\widehat{X_?})_{X \in \mathcal{V}^{+?}} + (!X.\mathrm{First})_{X \notin \mathcal{V}^{+?}}) \right) + ?\$.!\$.\mathrm{Ctrl} \tag{14}$$

The function of the Last equations is analogous to that of the First equation, except that it should find and execute the last send action instead of the first receive action. The way it works is more complex, because rotation can only be performed in one direction, so it has to rotate the queue several times in order to find the last send action. It does so using two helper markers; the *goto* marker γ and the *last* marker λ, and two additional equations Last_γ and Last_λ. For these markers it does not matter whether they are in $\mathcal{V}^{+\mathbf{1}}$ or not, since they are on the queue only when F is in the Last equations, and these do not provide a termination option; so they are chosen to be in $\mathcal{V}^{+\mathbf{1}}$ for consistency with the end-of-queue marker.

In finding the last variable with a send action, we alternate between the Last equation, which finds the next send action and writes the goto marker γ in front of it, and the Last_γ equation, which rotates to the previously written γ marker and changes it into the last marker λ. The Last_γ process also erases any previously found λ markers, since if we have written a new γ marker, then

clearly the λ marker did not actually mark the last variable with a send action. If in trying to find the next send action (in the Last equation), we encounter the end-of-queue marker \$, then the previously written λ marker did actually denote the last process with a send action, so we enter the Last_λ process, which rotates to the one remaining λ marker, removes it and executes the corresponding $\widehat{X_!}$ process, in the same way as was done with $\widehat{X_?}$ in First. If no λ was found in Last_λ, then there is no variable with a send action, so we return to Ctrl.

$$\begin{aligned}
\text{Last} &= \left(\sum_{X\in\mathcal{V}} ?X.((!\gamma.!X.\text{Last}_\gamma)_{X\in\mathcal{V}^{+!}} + (!X.\text{Last})_{X\notin\mathcal{V}^{+!}})\right) \\
&+ \quad ?\$.!\$.\text{Last}_\lambda \qquad (15)\\
\text{Last}_\gamma &= \left(\sum_{X\in\mathcal{V}\cup\{\$\}} ?X.!X.\text{Last}_\gamma\right) + ?\lambda.\text{Last}_\gamma \\
&+ \left(?\gamma.!\lambda.\sum_{X\in\mathcal{V}} ?X.!X.\text{Last}\right) \qquad (16)\\
\text{Last}_\lambda &= \left(\sum_{X\in\mathcal{V}} ?X.!X.\text{Last}_\lambda\right) + \left(?\lambda.\sum_{X\in\mathcal{V}} ?X.\widehat{X_!}\right) + ?\$.!\$.\text{Ctrl} \qquad (17)
\end{aligned}$$

Once an action has been chosen from First or Last, $\widehat{X_?}$ and $\widehat{X_!}$ mimic the behaviour of X by pushing the appropriate process variables ξ_i on the queue (or pushing no variables in case a terminating queued component has been chosen).

This concludes our sketch of the equations and workings of a regular process communicating with a queue, that can mimic an arbitrary process in QA, and thus our sketch of the proof that such a process exists.

The way this process works is sketched in Figure 1. Beginning with the process Ctrl $\parallel Q_\xi$ (the Ctrl process parallel to the queue in some state ξ) the process can enter either First or Last – given that there are both receive- and send-actions enabled. In First it can perform a receive action $?a_i$ (or any other enabled receive action), push the corresponding row of symbols ξ_i onto the queue, and fast forward to the state Ctrl $\parallel Q_{\xi'}$; in Last it can perform a send action $!a_j$ (or any other enabled send action), push the corresponding symbols ξ_j and fast forward to Ctrl $\parallel Q_{\xi''}$. In both the First and the Last case, it is possible to not make a choice by performing a FFWD instead and returning to the original Ctrl $\parallel Q_\xi$ state.

To prove our main theorem, i.e. $P = [R \parallel Q_\sigma]_*$, we need to prove that all processes work as described. Thus we need to prove that FFWD, First, Last, Last_γ, Last_λ and Push work as described above. For example, for FFWD we need to prove that it converges to Ctrl $\parallel Q_\xi$, where the end-of-queue marker is at the back of the queue, given any state of the queue. For First we need to prove that it gives the choice between executing the receive actions of the first process variable with receive actions, or that is returns to Ctrl. Furthermore it should be proven for First that when there is no process variable with a send action on the queue, that it returns to Ctrl, with the queue in the correct state. Likewise, lemmas need to be added for Last and Push.

As an example, we prove here the correctness of FFWD. There are several reasons why this process was chosen. Firstly the proof for this process is com-

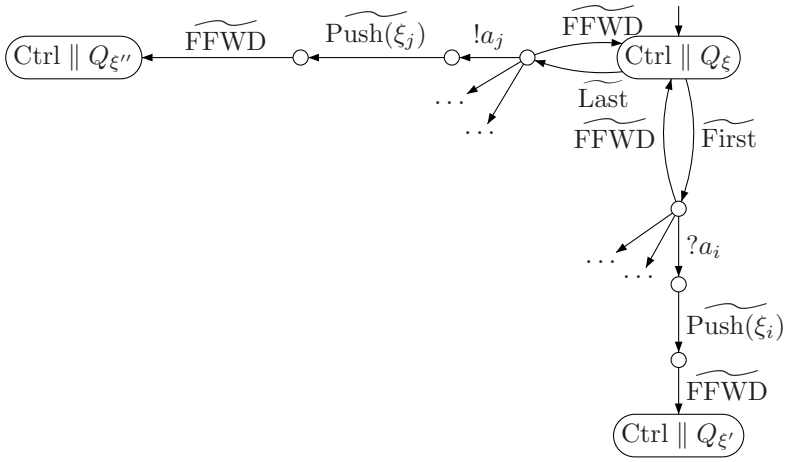

Figure 1: Transition system sketch of the workings of the Ctrl process in parallel with a queue. By $\langle \widetilde{equation} \rangle$ we indicate the τ-steps generated by that equation, up to the point where another process equation is entered.

pact. Also it illustrates how the other proofs can be done. Lastly, the process creates a τ-loop which, combined with the form of the equation, allows us to use KFAR [3] to show that the process does not make a choice. This also applies to the other proofs. Note that this is exactly what we want FFWD to do, rotating the contents of the queue without making a choice, which means that we can always return to a previous state as long as no explicit action is taken.

Lemma 5.2. FFWD *works such that it converges to* Ctrl *where the end-of-queue marker is at the back of the queue, given any state of the queue:* $[\text{FFWD} \parallel Q_{\xi\$\xi'}]_* = \tau.[\text{Ctrl} \parallel Q_{\xi'\xi\$}]_*$

Proof. This lemma can be proven by induction on the number of elements before the end-of-queue marker \$. This works as follows.

Base case Suppose $\xi = \varepsilon$, hence there are no elements before the end-of-queue marker \$. So we want to prove that: $[\text{FFWD} \parallel Q_{\$\xi'}]_* = \tau.[\text{Ctrl} \parallel Q_{\xi'\$}]_*$.

$$
\begin{aligned}
& [\text{FFWD} \parallel Q_{\$\xi'}]_* \\
= \quad & \{\text{definition FFWD}, Q_{\$\xi'}\} \\
& [(\sum_{X\in\mathcal{V}} ?X.!X.\text{FFWD})+?\$.!\$.\text{Ctrl} \parallel \sum_{d\in D} ?d.Q_{\$\xi' d} + !\$.Q_{\xi'}]_* \\
= \quad & \{\$ \notin \mathcal{V}, ?d \text{ can not communicate}\} \\
& \tau.[!\$.\text{Ctrl} \parallel Q_{\xi'}]_* \\
= \quad & \{\text{definition } Q_\varepsilon, \text{communication}\} \\
& \tau.\tau.[\text{Ctrl} \parallel Q_{\xi'\$}]_* \\
= \quad & \{\text{axiom}\} \\
& \tau.[\text{Ctrl} \parallel Q_{\xi'\$}]_*
\end{aligned}
$$

Hence, for $\xi = \varepsilon$ it holds that $[\text{FFWD} \parallel Q_{\$\xi'}]_* = \tau.[\text{Ctrl} \parallel Q_{\xi'\$}]_*$.

Step Now suppose $\xi = y\xi''$, then we want to prove that the following holds: $[\text{FFWD} \parallel Q_{y\xi''\$\xi'}]_* = \tau.[\text{Ctrl} \parallel Q_{\xi' y\xi''\$}]_*$. We can do this as follows.

$$
\begin{aligned}
& [\text{FFWD} \parallel Q_{y\xi''\$\xi'}]_* \\
= \quad & \{\text{definition FFWD}, Q_{y\xi''\$\xi'}\} \\
& [(\sum_{X\in\mathcal{V}} ?X.!X.\text{FFWD})+?\$.!\$.\text{Ctrl} \parallel \sum_{d\in D} ?d.Q_{y\xi''\$\xi' d} + !y.Q_{\xi''\$\xi'}]_* \\
= \quad & \{\text{communication}\} \\
& \tau.[!y.\text{FFWD} \parallel Q_{\xi''\$\xi'}]_* \\
= \quad & \{\text{definition } Q_{\xi''\$\xi'}, \text{communication}\} \\
& \tau.\tau.[\text{FFWD} \parallel Q_{\xi''\$\xi' y}]_* \\
= \quad & \{\text{Induction Hypothesis}\} \\
& \tau.[\text{FFWD} \parallel Q_{\$\xi' y\xi''}]_* \\
= \quad & \{\text{from the base case: } [\text{FFWD} \parallel Q_{\$\xi'}]_* = \tau.[\text{Ctrl} \parallel Q_{\xi'\$}]_*\} \\
& \tau.[\text{Ctrl} \parallel Q_{\xi' y\xi''\$}]_*
\end{aligned}
$$

Hence we have proven that: $[\text{FFWD} \parallel Q_{y\xi''\$\xi'}]_* = \tau.[\text{Ctrl} \parallel Q_{\xi' y\xi''\$}]_*$. This completes the proof of the lemma. □

The correctness of the other equations $\widehat{X_?}$, $\widehat{X_!}$, Push, Ctrl, First, Last, Last_γ and Last_λ, and their relation to the original process P, can be proven in a similar fashion.

6 Discussion

The Queue Algebra as presented in this paper is quite powerful, however it does not completely capture the queue nature which we originally set out to capture. To show this we compare the languages which can be generated by our operator to the queue languages. These queue languages are generated by queue grammars, as defined in [11].

Example 6.1 (**AntiDyck**). *The* AntiDyck *language (or FIFO) language is the language where elements which are read first are output first. This is the language most closely related to the queue nature.*

The following equation expresses AntiDyck in QA.

$$X \quad = \quad ?a.(X \between\!\!\!\between !a.\mathbf{1}) + ?b.(X \between\!\!\!\between !b.\mathbf{1}) + \mathbf{1}$$

Example 6.2 (**Anagram**). *Also the language of* Anagrams*, any permutation of $a^n b^n$, can be expressed as follows:*

$$X \quad = \quad ?a.(X \between\!\!\!\between !b.\mathbf{1}) + !b.(?a.\mathbf{1} \between\!\!\!\between X) + \mathbf{1}$$

Example 6.3 (**Palindrome**). *In [12] it is proven that the language of palindrome strings uu^R is not a queue language. However, the palindrome language can easily be represented in QA:*

$$X \quad = \quad ?a.(X \between ?a.\mathbf{1}) + ?b.(X \between ?b.\mathbf{1}) + \mathbf{1}$$

Since we can express a palindrome language, QA does not capture exactly the class of queue languages as defined in [11], but also expresses languages not contained therein.

In the previous example it was shown that QA is in some regards more powerful than queue languages. However, it remains to be seen whether the queue languages are strictly contained within QA. We check this by trying to express the queue language Castastrophe in QA.

Example 6.4 (**Catastrophe**). *The queue language* Catastrophe *is defined as follows:* $\langle a^{n_0}ba^{n_1}\ldots ba^{n_k}ba^*|n_h \leq 2n_{h-1}$ for $1 \leq h \leq k, n_0 = 1, k \geq 1\rangle$.

We conjecture that the catastrophe language cannot be generated by QA, hence not all queue languages are expressible in QA. This is due to the locality of the queue merge operator which cannot add to the end of a queue merge composition while removing from the beginning: If we have a queue merge composition $X_1 \between \ldots \between X_n$, where X_i (for $1 \leq i \leq n$) is an arbitrary process, it is not possible to perform an action that modifies both X_1 and X_n. The AntiDyck language was possible since we could split actions into matching input and output actions. For Catastrophe we cannot do this because of a required order on the actions.

Expressivity of QA How does the algebra QA relate to other defined notions of queueing formalisms? As can be seen from the previous examples, QA has overlap with the queue languages, but each can express languages not expressible by the other. This is not what we aimed for, since in the case of the stack and bag automata, they do represent the same class of languages as the sequential process algebra and basic parallel process algebra, respectively.

However, QA and the queue languages do share some languages that fundamentally have the "queue nature", such as AntiDyck. This leads us to propose a new Chomsky hierarchy including BQP as in Figure 2.

7 Concluding Remarks

We created and studied the queue merge operator and found that it is an operator with interesting properties, but lacking the strong link to queue languages that we desired.

The question whether there is a binary operator – instead of our queue merge operator – which has the "queue nature", or whether it can be proven that such an operator is non-algebraic and thus does not exist in process algebra, is still open and left for future work. We conjecture that such an operator does not exist. We think this is true due to the difference between the local nature of a binary operator in process algebra, and the global nature of the queue. In

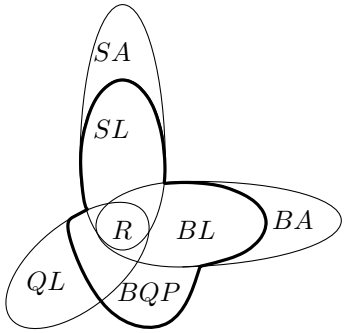

Figure 2: Chomsky hierarchy including expressivity of BQP. S and B stand for stack and bag respectively. A stands for automata, L stands for language. R is the class of regular languages. QL are the queue languages.

the queue, items are taken from the front and added at the back. It should be possible with an operator which has a variable number of operands.

A proof sketch was given of how every process in BQP can be expressed as a regular process communicating with a queue. Similarly it might be interesting to study how regular process in parallel with a queue can be written as a queue automaton.

Furthermore we conclude that a regular process communicating with the queue as defined in this paper is Turing-complete, following from the fact that this queue actually has queue semantics (see Appendix C), and by result from Post [17] that a finite automaton communicating with a queue can produce the same languages as a Turing machine.

Future Work The sequential composition and the parallel composition were studied in [6, 7]; in this paper we studied the queue merge operator. We can apply the same idea of studying the link between process algebra and automata theory to other operators and data structures, such as heaps, tree-like structures, or probabilistic or timed operators.

As already stated above, either trying to prove that a queue operator is essentially non-algebraic (which we think is the case), or finding semantics for an operator which *does* follow the queue languages, would be very interesting as well.

References

[1] L. Aceto, W.J. Fokkink, and C. Verhoef. Structural operational semantics. In J. Bergstra, A. Ponse, and S. Smolka, editors, *Handbook of Process Algebra, Chapter 3*, pages 197–292. Elsevier Science, Dordrecht, The Netherlands, 2001.

[2] J.C.M. Baeten, T. Basten, and M.A. Reniers. *Process Algebra: Equational Theories of Communicating Processes*. Cambridge University Press, 2008.

[3] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. On the consistency of Koomen's fair abstraction rule. *Theoretical Computer Science*, 51(1–2):129–176, 1987.

[4] J.C.M. Baeten, J.A. Bergstra, and J.W. Klop. Decidability of bisimulation equivalence for processes generating context-free languages. *Journal of the ACM*, 40(3):653–682, 1993.

[5] J.C.M. Baeten and M. Bravetti. A ground-complete axiomatization of finite state processes in process algebra. In M. Abadi and L. de Alfaro, editors, *Proceedings of CONCUR 2005*, number 3653 in LNCS, pages 246–262. Springer, 2005.

[6] J.C.M. Baeten, P.J.L. Cuijpers, and P.J.A. van Tilburg. A context-free process as a pushdown automaton. In *Proceedings of CONCUR 2008*, number 5201 in LNCS, pages 98–113, Berlin-Heidelberg, 2008. Springer-Verlag.

[7] J.C.M. Baeten, P.J.L. Cuijpers, and P.J.A. van Tilburg. A basic parallel process as a parallel pushdown automaton. In *Proceedings of EXPRESS 2009*, ENTCS. Elsevier, 2009. To appear.

[8] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge University Press, 1990.

[9] J. A. Bergstra and J. Tiuryn. Process algebra semantics for queues. Technical Report IW 241/83, Mathematisch Centrum, 1983.

[10] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1/3):109–137, 1984.

[11] L. Breveglieri. *The word problem of queue languages is NP-complete*. PhD thesis, Politecnico di Milano, 2002.

[12] A. Cherubini, C. Citrini, S. Crespi Reghizzi, and D. Mandrioli. Breadth and depth grammars and deque automata. *International Journal of Foundations of Computer Science*, vol 1 no. 3:219–232, 1990.

[13] R. de Simone. Higher-level synchronizing devices in MEIJE-SCCS. *Theoretical Computer Science (TCS)*, 37:245–267, 1985.

[14] R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, 1996.

[15] F. Moller. The importance of the left merge operator in process algebras. In M.S. Paterson, editor, *Proceedings of ICALP'90*, number 443 in LNCS, pages 752–764. Springer-Verlag, 1990.

[16] F. Moller. Infinite results. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: Concurrency Theory, 7th International Conference*, volume 1119 of *Lecture Notes in Computer Science*, pages 195–216, Pisa, Italy, 26–29 August 1996. Springer-Verlag.

[17] E. Post. Formal reductions of the general combinatorial decision problem. *American Journal of Mathematics*, 65:197–215, 1943.

A Soundness Proofs

In this section, we give soundness proofs for the axioms of QA. All axioms have been verified as sound, but not all of these proofs have been written out here.

A.1 Soundness of Axiom SQ3

Theorem A.1. *The queue merge operator $\between$ is an associative operator, i.e. axiom SQ3 is sound according to the semantics of QA.*

Proof. In order to prove soundness of $(x \between y) \between z) = x \between (y \between z)$, for all processes x, y and z in BQP, consider the following relation R:

$$R = \{((x \between y) \between z, x \between (y \between z)) \mid x, y, z \in \mathbf{P}\}$$

Obviously, R is a witness for the axiom. To prove that it is a bisimulation relation, we consider the following four cases when pRq:

1. pRq and $p \xrightarrow{a} p'$.
   In this case, p can only be of the form $(x \between y) \between z$, and q can only be of the form $x \between (y \between z)$. Regarding a we distinguish the following two cases:
   (a) $a \in \mathcal{A}_?$, so $a = ?b$ for some $?b \in \mathcal{A}_?$.
   According to the semantics, $(x \between y) \between z \xrightarrow{?b} p'$ can only originate from the rules 19 and 22, so we distinguish the following two cases:
      i. Rule 19: $p' = p'' \between z$ and $x \between y \xrightarrow{?b} p''$. This can once again be done using rules 19 and 22:
         A. In the first case, $p'' = x' \between y$ and $x \xrightarrow{?b} x'$, so $(x \between y) \between z \xrightarrow{?b} (x' \between y) \between z$ if $x \xrightarrow{?b} x'$. This can be simulated by $q = x \between (y \between z)$ by applying rule 19, so $q \xrightarrow{?b} x' \between (y \between z)$. Then we have $((x' \between y) \between z, x' \between (y \between z)) \in R$.
         B. In the second case, $p'' = x \between y'$, $y \xrightarrow{?b} y'$ and $\neg?(x)$, so $(x \between y) \between z \xrightarrow{?b} (x \between y') \between z$ if $y \xrightarrow{?b} y'$ and $\neg?(x)$. This can be simulated by $q = x \between (y \between z)$ by first applying rule 22 and then rule rule 19. Then we have $((x \between y') \between z, x \between (y' \between z)) \in R$.
      ii. Rule 22: $p' = (x \between y) \between z'$, $z \xrightarrow{?b} z'$, $x \between \neg?(y)$. $x \between \neg?(y)$ holds if there is no SOS-rule that can make $x \between y$ perform a ?-step, i.e., if rules 19 and 22 are not applicable on $x \between y$; this is the case when $\neg?(x)$ and $\neg?(y)$. So we have $(x \between y) \between z \xrightarrow{?b} (x \between y) \between z'$ if $z \xrightarrow{?b} z'$ and $\neg?(x)$ and $\neg?(y)$. This can be simulated by $q = x \between (y \between z)$ by applying rule 22 twice; then we have $((x \between y) \between z', x \between (y \between z')) \in R$.
   (b) $a \in \mathcal{A}_!$, so $a = !b$ for some $!b \in \mathcal{A}_!$. Now according to the semantics, $(x \between y) \between z \xrightarrow{!b} p'$ can only originate from rules 20 and 21, so we distinguish the following two cases:
      i. Rule 20: $p' = (x \between y) \between z'$, and $z \xrightarrow{!b} z'$, so $(x \between y) \between z \xrightarrow{!b} (x \between y) \between z'$. This can be simulated by applying rule 20 twice on q; this yields the relation $((x \between y) \between z', x \between (y \between z')) \in R$.

ii. Rule 21: $p' = p'' \between z$ and $(x \between y) \xrightarrow{!b} p''$ and $\neg!(z)$. Here rules 20 and 21 are applicable once more:

A. In the first case, $p'' = x \between y'$ and $y \xrightarrow{!b} y'$, so $(x \between y) \between z \xrightarrow{!b} (x \between y') \between z$ for $\neg!(z)$ and $y \xrightarrow{!b} y'$. This can be simulated by applying rules 20 and 21 on q, yielding the relation $((x \between y') \between z, x \between (y' \between z)) \in R$.

B. In the second case, $p'' = x' \between y$, for $x \xrightarrow{!b} x'$ and $\neg!(y)$, so we have $(x \between y) \between z \xrightarrow{!b} (x' \between y) \between z$ if $x \xrightarrow{!b} x'$, $\neg!(y)$ and $\neg!(z)$. This can be simulated by applying rule 21 on q, and observing (analogous to case 1(a)ii above) that $y \between \neg!(z)$ iff $\neg!(y)$ and $\neg!(z)$. This results in the relation $((x' \between y) \between z, x' \between (y \between z)) \in R$.

Since $\mathcal{A}_? \uplus \mathcal{A}_! = \mathcal{A}$, these two cases exhaustively enumerate the possible values for a, and concludes that for $(p, q) \in R$, when p can do a step to p', than q can also do a step to q' with $(p', q') \in R$.

2. pRq and $q \xrightarrow{a} q'$.
The case were q can take a step to q' and is simulated by p, is a dual to case 1 (where p can take a step to p' and is simulated by q) above.

3. pRq and $p\downarrow$.
In this case, p can only be of the form $(x \between y) \between z$, and q can only be of the form $x \between (y \between z)$. Looking at the semantics, $p\downarrow$ can only originate from rule 23, so we have $(x \between y) \between z\downarrow$ if $x \between y\downarrow$ and $z\downarrow$. Again only rule 23 can be applied on $x \between y\downarrow$, resulting in $x\downarrow$ and $y\downarrow$. In total, we get $(x \between y) \between z\downarrow$ if $x\downarrow$ and $y\downarrow$ and $z\downarrow$. This can be simulated in q by applying rule 23 to q twice, so q can terminate when p can terminate.

4. pRq and $q\downarrow$.
The case where q can terminate is a dual to case 3 above, so p can terminate when q can terminate.

Summing up, the four requirements for a bisimulation relation are met, so R is a bisimulation relation. Since $((x \between y) \between z, x \between (y \between z)) \in R$, $(x \between y) \between z$ is bisimilar to $x \between (y \between z)$, so $\between$ is associative. □

A.2 Soundness of Axiom MQ

Theorem A.2. *Axiom MQ is sound according to the semantics QA and the left and right queue merge.*

Proof. In order to prove soundness of $x \between y = x \between\!\!\!\backslash\, y + x \,/\!\!\!\between y$, for all processes x and y in BQP, consider the following relation R:

$$R = \{(x \between y, x \between\!\!\!\backslash\, y + x \,/\!\!\!\between y), (x \between y, x \between y) \mid x, y \in \mathbf{P}\}$$

The first relation pair of R is a witness for the axiom MQ. To prove that R is a bisimulation relation, we focus on the first pair of R; since in the second pair $x \between y$ is syntactically equivalent to $x \between y$, bisimilarity between these two is implied. For the first pair, if we have pRq then $p = x \between y$ and $q = x \between\!\!\!\backslash\, y + x \,/\!\!\!\between y$, and we distinguish the following four cases:

1. pRq and $p \xrightarrow{a} p'$.
   Regarding a we distinguish:

   (a) $a \in \mathcal{A}_?$, so $a = ?b$ for some $?b \in \mathcal{A}_?$. Now according to the semantics, rules 19 and 22 are applicable, so we distinguish between these two:

      i. By rule 19, $x \between y \xrightarrow{?b} x' \between y$ if $x \xrightarrow{?b} x'$. This can be simulated in q by applying rules 3 and 24, resulting in $x \between y + x \between y \xrightarrow{?b} x' \between y$ if $x \xrightarrow{?b} x'$. Then we have $(x' \between y, x' \between y) \in R$.

      ii. By rule 22, $x \between y \xrightarrow{?b} x \between y'$ if $y \xrightarrow{?b} y'$ and $\neg?(x)$. This can be simulated in q by applying rules 4 and 27, resulting in $x \between y + x \between y \xrightarrow{?b} x \between y'$ if $y \xrightarrow{?b} y'$ and $\neg?(x)$. Then we have $(x \between y', x \between y') \in R$.

   (b) $a \in \mathcal{A}_!$, so $a = !b$ for some $!b \in \mathcal{A}_!$. This is a dual to case 1a above, with send actions and rules instead of receive actions and rules.

2. pRq and $q \xrightarrow{a} q'$.
   Since $q = x \between y + x \between y$, rules 3 and 4 are applicable:

   (a) Using rule 3, $x \between y + x \between y \xrightarrow{a} q'$ if $x \between y \xrightarrow{a} q'$. Then we distinguish on a:

      i. $a = ?b$ for some $?b \in \mathcal{A}_?$. Then only rule 24 is applicable, so $x \between y \xrightarrow{?b} q'$ if $x \xrightarrow{?a} x'$ and $q' = x' \between y$. Thus, we get $x \between y + x \between y \xrightarrow{?b} x' \between y$ if $x \xrightarrow{?a} x'$. This can be simulated in p by applying to it rule 19, yielding $(x' \between y, x' \between y) \in R$.

      ii. $a = !b$ for some $!b \in \mathcal{A}_!$. Then only rule 26 is applicable, so $x \between y \xrightarrow{!b} q'$ if $x \xrightarrow{!b} x'$, $\neg!(y)$ and $q' = x' \between y$. In this way, we get $x \between y + x \between y \xrightarrow{!b} x' \between y$ if $x \xrightarrow{!b} x'$ and $\neg!(y)$. This can be simulated in p by applying rule 21, again yielding $(x' \between y, x' \between y) \in R$.

   (b) Using rule 4, $x \between y + x \between y \xrightarrow{a} q'$ if $x \between y \xrightarrow{a} q'$. This is a dual to case 2a above, with steps in the right queue merge instead of the left queue merge.

3. pRq and $p\downarrow$.
   The only applicable rule here is rule 23, so $x \between y\downarrow$ if $x\downarrow$ and $y\downarrow$. This can be simulated in q by applying either rule 5 and then 28 or rule 6 and then 29.

4. pRq and $q\downarrow$.
   The two applicable rules are 5 and 6, so distinguish on those:

   (a) With rule 5, $x \between y + x \between y\downarrow$ if $x \between y\downarrow$; with rule 28, this holds if $x\downarrow$ and $y\downarrow$. This can be simulated in p by applying rule 23.

   (b) With rule 6, $x \between y + x \between y\downarrow$ if $x \between y\downarrow$; with rule 29, this also holds if $x\downarrow$ and $y\downarrow$, so this can also be simulated in p by applying rule 23.

As can be seen from the above, R is a bisimulation relation. Since $(x \between y, x \between y + x \between y) \in R$, $x \between y$ is bisimilar to $x \between y + x \between y$, so axiom MQ is sound. □

Proof. This is a dual to the proof of the soundness of axiom LMQ9, described in section A.7 above, only substituting the semantics of the right queue merge for that of the left queue merge. □

A.3 Soundness of Axiom LMQ6

Theorem A.3. *Axiom LMQ6 is sound according to the semantics of QA and the left and right queue merge.*

Proof. In order to prove soundness of $?a.x \between y = ?a.(x \between y)$, for all processes x, y, z in BQP, consider the following relation R:

$$R = \{(?a.x \between y, ?a.(x \between y)), (x \between y, x \between y) \mid x, y, z \in \mathbf{P}\}$$

R is a witness for axiom LMQ6, so we need to prove that R is a bisimulation relation. For each p and q in R it holds that if we have pRq then $p = ?a.x \between y$ and $q = ?a.(x \between y)$. We distinguish here the following four cases:

1. pRq and $p \xrightarrow{a} p'$.
   Note that $a \in \mathcal{A}_?$. Now we can only apply SOS rule 24 on p, resulting in $?a.x \between y \xrightarrow{?a} x \between y$. q can simulate this behavior by applying rule 2, resulting in $?a.(x \between y) \xrightarrow{?a} x \between y$. As the result of both equations is the same, we can apply the induction hypothesis $(x \between y, x \between y) \in R$, and these results are bisimilar. As the transitions are equal and the resulting states are bisimilar, the two states before the transactions must therefore also be bisimilar.

2. pRq and $q \xrightarrow{a} q'$.
   Again note that $a \in \mathcal{A}_?$, therefore we can only apply rule 2 on q; $?a.(x \between y) \xrightarrow{?a} x \between y$. This is simulated by p using rule 24; $?a.x \between y \xrightarrow{?a} x \between y$. By the same reasoning as above $(x \between y, x \between y) \in R$, and thus are bisimilar.

3. pRq and $p\downarrow$.
   We could try to apply 28 on $?a.x \between y\downarrow$, and we would get $?a.x\downarrow$ and $y\downarrow$ as conditional cases. Clearly there is no semantical rule for $?a.x\downarrow$, so we can not use 28 on $p\downarrow$. In other words p can not terminate. As we are trying to prove bisimilarity, and p can not terminate, this proof obligation can be dropped, as it is a tautology.

4. pRq and $q\downarrow$.
   As there is no rule that fits $?a.(x \between y)\downarrow$, we can apply the same reasoning as used in the case $p\downarrow$.

It is evident from the above that R is a bisimulation relation. Since $(?a.x \between y, ?a.(x \between y)) \in R$, $?a.x \between y$ is bisimilar to $?a.(x \between y)$, so axiom LMQ6 is sound. □

A.4 Soundness of Axiom RMQ6

Theorem A.4. *Axiom RMQ6 is sound according to the semantics of QA and the left and right queue merge.*

Proof. This is a dual to the proof of the soundness of axiom LMQ6, described in section A.3 above, only substituting the semantics of the right queue merge for that of the left queue merge. □

A.5 Soundness of Axiom LMQ8

Theorem A.5. *Axiom LMQ8 is sound according to the semantics of QA and the left and right queue merge operators.*

Proof. In order to prove soundness of $\neg!(y) \Rightarrow !a.x \between y = !a.(x \between y)$, for all processes x and y in BQP, consider the following relation R:

$$R = \{((!a.x \between y, !a.(x \between y)), (x \between y, x \between y) \mid y, z \in \mathbf{P} \wedge \neg!(y)\}$$

R is a witness for axiom LMQ8, so we need to prove that R is a bisimulation relation. It is evident that the second pair in R is bisimilar due to syntactical equivalence, so we focus on the first pair. For each p and q in R it holds that if we have pRq then $p = !a.x \between y$ and $q = !a.(x \between y)$, where $\neg!(y)$ holds. We distinguish the following cases:

1. pRq and $p \xrightarrow{a} p'$.
   Note that $a \in \mathcal{A}_!$. Now we can only apply SOS rule 26 on p, resulting in $!a.x \between y \xrightarrow{!a} x \between y$. q can simulate this behavior by applying rule 2, resulting in $!a.(x \between y) \xrightarrow{!a} x \between y$. As the result of both equations is the same, we can apply the induction hypothesis $(x \between y, x \between y) \in R$, and these results are bisimilar. As the transitions are equal and the resulting states are bisimilar, the two states before the transactions must therefore also be bisimilar.

2. pRq and $q \xrightarrow{a} q'$.
   Again note that $a \in \mathcal{A}_!$, therefore we can only apply rule 2 on q; $!a.(x \between y) \xrightarrow{!a} x \between y$. This is simulated by p using rule 26; $!a.x \between y \xrightarrow{!a} x \between y$. By the same reasoning as above $(x \between y, x \between y) \in R$, and thus are bisimilar.

3. pRq and $p\downarrow$.
   We could try to apply 28 on $!a.x \between y\downarrow$, and we would get $!a.x\downarrow$ and $y\downarrow$ as conditional cases. Clearly there is no semantical rule for $!a.x\downarrow$, so we can not use 28 on $p\downarrow$. In other words p can not terminate. As we are trying to prove bisimilarity, and p can not terminate, this proof obligation can be dropped, as it is a tautology.

4. pRq and $q\downarrow$.
   As there is no rule that fits $!a.(x \between y)\downarrow$, we can apply the same reasoning as used in the case $p\downarrow$.

□

A.6 Soundness of Axiom RMQ8

Theorem A.6. *Axiom RMQ8 is sound according to the semantics of QA and the left and right queue merge operators.*

Proof. This is a dual to the proof of the soundness of axiom LMQ8, described in Section A.5 above, only substituting the semantics of the right queue merge for that of the left queue merge, and swapping the send and receive actions. □

A.7 Soundness of Axiom LMQ9

Theorem A.7. *Axiom LMQ9 is sound according to the semantics QA and the left and right queue merge.*

Proof. In order to prove soundness of $(x+y) \between z = (x \between z) + (y \between z)$, for all processes x, y, z in BQP, consider the following relation R:

$$R = \{((x+y) \between z, (x \between z) + (y \between z)), (x \between y, x \between y) \mid x, y, z \in \mathbf{P}\}$$

R is a witness for axiom LMQ9, so we need to prove that R is a bisimulation relation. It is evident that the second pair in R is bisimilar due to syntactical equivalence, so we focus on the first pair, for which it holds that if we have pRq then $p = (x+y) \between z$ and $q = (x \between z) + (y \between z)$. We distinguish here the following four cases:

1. pRq and $p \xrightarrow{a} p'$.
   Regarding a we distinguish:
   (a) $a = ?b$ for some $?b \in \mathcal{A}_?$. Then the only rule applicable to p is rule 24, so $(x+y) \between z \xrightarrow{?b} p'' \between z$ and $x + y \xrightarrow{?b} p''$ for some p''. This is possible using rule 3 and 4:
      i. Using rule 3, $x+y \xrightarrow{?b} x'$ for $x \xrightarrow{?b} x'$, so $(x+y) \between z \xrightarrow{?b} x' \between z$. This can be simulated in q with rules 3 and 24, resulting in $(x \between z)+(y \between z) \xrightarrow{?b} x' \between z$ if $x \xrightarrow{?b} x'$, and we get $(x' \between z, x' \between z) \in R$.
      ii. Using rule 4, $x + y \xrightarrow{?b} y'$ for $y \xrightarrow{?b} y'$, so $(x+y) \between z \xrightarrow{?b} y' \between z$. This can be simulated in q by applying rules 4 and 24, resulting in $(x \between z) + (y \between z) \xrightarrow{?b} y' \between z$ if $y \xrightarrow{?b} y'$, so we get $(y' \between z, y' \between z) \in R$.
   (b) $a = !b$ for some $!b \in \mathcal{A}_!$. Here the only rule applicable to p is rule 26, so $(x+y) \between z \xrightarrow{!b} p'' \between z$ and $(x+y) \xrightarrow{!b} p''$ and $\neg!(z)$ for some p''. Now either rule 3 or 4 can be applied:
      i. Using rule 3 we get $(x+y) \between z \xrightarrow{!b} x' \between z$ for $x \xrightarrow{!b} x'$ and $\neg!(z)$. This can be simulated in q by applying rules 3 and then 26, resulting in $(x \between z)+(y \between z) \xrightarrow{!b} x' \between z$, so $(x' \between z, x' \between z) \in R$.
      ii. Using rule 4 we get $(x+y) \between z \xrightarrow{!b} y' \between z$ for $y \xrightarrow{!b} y'$ and $\neg!(z)$. Again, this can be simulated in q by applying rules 4 and 26, resulting in $(x \between z) + (y \between z) \xrightarrow{!b} y' \between z$, so $(y' \between z, y' \between z) \in R$.
2. pRq and $q \xrightarrow{a} q'$.
   There are two rules applicable here, 3 and 4:
   (a) $(x \between z) + (y \between z) \xrightarrow{a} q''$, for $x \between z \xrightarrow{a} q''$. Then we can distinguish on a:

i. $a = ?b$ for some $?b \in \mathcal{A}_?$. Then only rule 24 is applicable, resulting in $(x \between z) + (y \between z) \xrightarrow{?b} x' \barwedge z$ for $x \xrightarrow{?b} x'$. This can be simulated in p by applying rules 24 and 3, which yields $(x + y) \between z \xrightarrow{?b} x' \barwedge z$ for $x \xrightarrow{?b} x'$, so $(x' \barwedge z, x' \barwedge z) \in R$.

ii. $a = !b$ for some $!b \in \mathcal{A}_!$. Then the only applicable rule is rule 26, so $(x \between z) + (y \between z) \xrightarrow{!b} x' \barwedge z$ for $x \xrightarrow{!b} x'$ and $\neg!(z)$. This can be simulated in p by applying rules 26 and 3, resulting in $(x + y) \between z \xrightarrow{!b} x' \barwedge z$ for $x \xrightarrow{!b} x'$ and $\neg!(z)$, so once again we have $(x' \barwedge z, x' \barwedge z) \in R$.

(b) $(x \between z) + (y \between z) \xrightarrow{a} q''$ for $y \between z \xrightarrow{a} q''$. This case is a dual to case 2a above.

3. pRq and $p\downarrow$.
We can only apply rule 28, so $(x + y) \between z\downarrow$ if $x + y\downarrow$ and $z\downarrow$. On $x + y\downarrow$ we can apply either rule 5 or 6:

(a) Using rule 5, $(x + y) \between z\downarrow$ if $x\downarrow$ and $z\downarrow$. In this case, q can terminate by applying rule 5 and rule 28.

(b) Using rule 6, $(x + y) \between z)\downarrow$ if $y\downarrow$ and $z\downarrow$, and q can terminate by applying rules 6 and 28.

4. pRq and $q\downarrow$.
Here we can apply either rule 5 or 6:

(a) Using rule 5, $(x \between z) + (y \between z)\downarrow$ if $x \between z\downarrow$, using rule 28 is true if $x\downarrow$ and $z\downarrow$. In this case, p can terminate using rules 28 and 5.

(b) Using rule 6, q can terminate if $y \between z\downarrow$, which is true using rule 28 if $y\downarrow$ and $z\downarrow$. In this case, p can terminate using rules 28 and 6.

It is evident from the above that R is a bisimulation relation. Since $((x + y) \between z, (x \between z) + (y \between z)) \in R$, $(x + y) \between z$ is bisimilar to $(x \between z) + (y \between z)$, so axiom LMQ9 is sound. □

A.8 Soundness of Axiom RMQ9

Theorem A.8. *Axiom RMQ9 is sound according to the semantics of QA and the left and right queue merge.*

Proof. This is a dual to the proof of the soundness of axiom LMQ9, described in section A.7 above, only substituting the semantics of the right queue merge for that of the left queue merge. □

B Any Term in Greibach Normal Form

We want to show that using the axioms of QA, every term over QA can be brought into *Greibach Normal Form* (GNF):

$$X = \sum_{i \in I_X} a_i.\xi_i \ (+\mathbf{1}) \tag{18}$$

This way, each right hand side of an equation can be written as a sum over a number of terms indexed by a finite set I_X (the empty sum is $\mathbf{0}$). Each term is of the form $\mathbf{1}$ or $a_i.\xi_i$, where ξ_i is defined as the queue merge over a number of variables, i.e. $X_{i_1} \between X_{i_2} \between \ldots \between X_{i_n}$. Note that action a_i can either be a send ($!a_i$) or a receive ($?a_i$) action.

We use structural induction to show that every term over QA can be brought into GNF. Recall the definition of QA.

$$Q := \ ?a.Q \mid !a.Q \mid \mathbf{1} \mid \mathbf{0} \mid Q + Q \mid Q \between Q \mid Q \between\!\backslash Q \mid Q \backslash\!\between Q \mid \neg!(x) \mid \neg?(x) \quad (19)$$

We distinguish the following cases.

case $X = ?a.Y$ We define the following process:

$$X = ?a.Y, \qquad Y = Z \quad (20)$$

The process $Y = Z$ is in GNF by the induction hypothesis, which implies that $X = ?a.Z$ is also in GNF.

case $X = !a.Y$ Exactly the same as the case $X = ?a.Y$, but with the ? replaced by the !.

case $X = \mathbf{1}$ Trivial, a single $\mathbf{1}$ summand.

case $X = \mathbf{0}$ Trivial, the empty sum.

case $X = Y + Z$ Terms Y and Z are in GNF by induction hypothesis. By applying the definition we come to the following process.

$$X = \sum_{i \in I_Y} a_i.(\xi_i)\ (+\mathbf{1})_{Y \in \mathcal{V}^{+\mathbf{1}}} + \sum_{i \in I_Z} a_i.(\xi_i)\ (+\mathbf{1})_{Z \in \mathcal{V}^{+\mathbf{1}}} \quad (21)$$

This can be rewritten to the following form, which is in GNF. A term $(X)_{cond}$ is equal to the term (X) when condition *cond* holds. So the term $(+\mathbf{1})_{Y \in \mathcal{V}^{+\mathbf{1}}}$ is equal to $+\mathbf{1}$ when $Y \in \mathcal{V}^{+\mathbf{1}}$, i.e. when Y has a $\mathbf{1}$-summand.

$$\sum_{i \in I_Y \cup I_Z} a_i.(\xi_i)\ (+\mathbf{1})_{(Y \in \mathcal{V}^{+\mathbf{1}} \vee Z \in \mathcal{V}^{+\mathbf{1}})} \quad (22)$$

case $X = Y \between\!\backslash Z$ By applying the definitions, we can rewrite $X = Y \between\!\backslash Z$ to the following form.

$$\Big(\sum_{i \in I_Y} a_i.(\xi_i)(+\mathbf{1})_{Y \in \mathcal{V}^{+\mathbf{1}}}\Big) \between\!\backslash \Big(\sum_{i \in I_Z} a_i.(\xi_i)(+\mathbf{1})_{Z \in \mathcal{V}^{+\mathbf{1}}}\Big) \quad (23)$$

By applying axiom LMQ9 this can be rewritten to:

$$\sum_{i \in I_Y} \Big(a_i.(\xi_i) \between\!\backslash \sum_{i \in I_Z} a_i.(\xi_i)(+\mathbf{1})_{Z \in \mathcal{V}^{+\mathbf{1}}}\Big) + \Big(1 \between\!\backslash \sum_{i \in I_Z} a_i.(\xi_i)(+\mathbf{1})_{Z \in \mathcal{V}^{+\mathbf{1}}}\Big)_{Y \in \mathcal{V}^{+\mathbf{1}}} \quad (24)$$

Which can be rewritten using LMQ4, LMQ5 and LMQ3 to the following form:

$$\sum_{i \in I_Y} \Big(a_i.(\xi_i) \between\!\backslash \sum_{i \in I_Z} a_i.(\xi_i)(+\mathbf{1})_{Z \in \mathcal{V}^{+\mathbf{1}}}\Big)(+\mathbf{1})_{Y,Z \in \mathcal{V}^{+\mathbf{1}}} \quad (25)$$

In order to check if this process can be written in GNF, we need to check what the actions a_i can do. Therefore we do a case distinction over the following term:

$$a_i.(\xi_i) \between \sum_{i \in I_Z} a_i.(\xi_i)(+\mathbf{1})_{Z \in \mathcal{V}^{+1}} \tag{26}$$

case $\exists_{i \in I_Y}(a_i = ?a)$ When there exists a receive action on the left hand side of the left queue merge we have the following situation:

$$?a.(\xi_i) \between \sum_{i \in I_Z} a_i.(\xi_i)(+\mathbf{1})_{Z \in \mathcal{V}^{+1}} \tag{27}$$

This can be written by axiom LMQ6 into the following form, which is in GNF.

$$?a.(\xi_i \between Y), \qquad Y = \sum_{i \in I_Z} a_i.(\xi_i)(+\mathbf{1})_{Z \in \mathcal{V}^{+1}} \tag{28}$$

case $\exists_{i \in I_Y}(a_i = !a)$ When there exists a send action on the LHS of the left queue merge we have the following situation:

$$!a.(\xi_i) \between \sum_{i \in I_Z} a_i.(\xi_i)(+\mathbf{1})_{Z \in \mathcal{V}^{+1}} \tag{29}$$

Now, we need to make another case distinction. This is because the $!a$ can only be done when there are no send actions on the RHS of the left queue merge. We therefore need to check whether there is a send action on the RHS of the left queue merge.

case $\exists_{i \in I_Z}(a_i = !b)$ In case there is a send action on the RHS of the left queue merge, the process is in deadlock by axiom LMQ7. Hence, in this case the process is equal to $\mathbf{0}$, which is in GNF as has been shown earlier.

case $\neg\exists_{i \in I_Z}(a_i = b)$ In case there is no send action on the RHS of the left queue merge, all actions on the side have to be receive actions; $\forall_{i \in I_Z}(a_i \in \mathcal{A}_?)$. Therefore we can use axiom NL2 and NL3 to write the process as follows:

$$!a.(\xi_i) \between \sum_{i \in I_Z} \neg!(a_i.(\xi_i))(+\neg!(\mathbf{1}))_{Z \in \mathcal{V}^{+1}} \tag{30}$$

Using axiom NL6 this can be rewritten to:

$$!a.(\xi_i) \between \neg!(\sum_{i \in I_Z} a_i.(\xi_i)(+\mathbf{1})_{Z \in \mathcal{V}^{+1}}) \tag{31}$$

Which can rewritten using axiom LMQ8 to the following form.

$$!a.\Big(\xi_i \between \neg!(\sum_{i \in I_Z} a_i.(\xi_i)(+\mathbf{1})_{Z \in \mathcal{V}^{+1}})\Big) \tag{32}$$

Because $\neg\exists_{i \in I_Z}(a_i = b)$, the following holds:

$$\sum_{i \in I_Z} a_i.(\xi_i)(+\mathbf{1})_{Z \in \mathcal{V}^{+1}} = \neg!(\sum_{i \in I_Z} a_i.(\xi_i)(+\mathbf{1})_{Z \in \mathcal{V}^{+1}}) \tag{33}$$

Hence, we can rewrite the original process to the following form, where Y is in GNF.

$$!a.Y, \qquad Y = \Big(\xi_i \between \sum_{i \in I_Z} a_i.(\xi_i)(+\mathbf{1})_{Z \in \mathcal{V}^{+1}}\Big) \tag{34}$$

Since we have shown that $!a.Q$ can be written in GNF if Q is in GNF, it follows that $X = Y \between Z$ can be written in GNF.

case $X = Y \between Z$ This can be proven in a similar way as in the case of $X = Y \between Z$.

case $X = Y \between Z$ By axiom MQ, $X = Y \between Z$ can be rewritten to the following form.

$$X = Y \between Z + Y \between Z \tag{35}$$

We have shown that the $Y \between Z$ and $Y \between Z$ can be written in GNF. Furthermore we have shown that the alternative composition of two processes in GNF is again in GNF. Hence $Y \between Z$ can be brought into GNF.

case $X = \neg!(X)$ From the equational theory of QA (Table 6) it follows that every term $\neg!(X)$ can be rewritten into a term without the $\neg!$-operator. The resulting term is then covered by any of the other cases.

case $X = \neg?(X)$ This can be proven in a similar way as in the $\neg!(X)$-case.

C RSP Proof

Recall the definition of the queue:

$$Q = \sum_{d \in D} ?d.(Q \between !d.\mathbf{1}) + \mathbf{1} \tag{36}$$

We want to proof using RSP that this process is equal to the following process:

$$Q_\varepsilon = \sum_{d \in D} ?d.Q_d + \mathbf{1} \tag{37}$$

$$Q_{e\xi} = \sum_{d \in D} ?d.Q_{e\xi d} + !e.Q_\xi \tag{38}$$

For the proof we need that $Q_\xi = Q \between \xi$. In the term $Q \between \xi$, the term ξ represents a term of the form $!d_1.\mathbf{1} \between !d_2.\mathbf{1} \ldots \between !d_n.\mathbf{1}$. The notation is as follows:

$$\begin{aligned} Q \between \varepsilon &= Q \\ Q \between \xi d &= Q \between !d.\mathbf{1} \between \xi \\ Q \between d\xi &= Q \between \xi \between !d.\mathbf{1} \end{aligned}$$

We first prove using RSP that $Q_\xi = Q \between \xi$.
The base case is as follows:

$$
\begin{array}{ll}
 & Q \between \varepsilon \\
= & \{\text{notation}\} \\
 & Q \\
= & \{\text{definition} Q\} \\
 & \sum_{d \in D} ?d.(Q \between !d.1) + \mathbf{1} \\
= & \{\text{notation}\} \\
 & \sum_{d \in D} ?d.(Q \between d) + \mathbf{1} \\
= & \{\text{definition } Q_\varepsilon\} \\
 & Q_\varepsilon
\end{array}
$$

The inductive case is as follows:

$$
\begin{array}{ll}
 & Q \between (\xi \between !e.\mathbf{1}) \\
= & \{\text{definition Q}\} \\
 & (\sum_{d \in D} ?d.(Q \between !d.\mathbf{1}) + \mathbf{1}) \between (\xi \between !e.\mathbf{1}) \\
= & \{\text{axiom MQ}\} \\
 & (\sum_{d \in D} ?d.(Q \between !d.\mathbf{1}) + \mathbf{1}) \leftthreetimes (\xi \between !e.\mathbf{1}) \\
 & \quad +(\sum_{d \in D} ?d.(Q \between !d.\mathbf{1}) + \mathbf{1}) \rightthreetimes (\xi \between !e.\mathbf{1}) \\
= & \{\text{axioms LMQ9, LMQ5 and LMQ6 }\} \\
 & (\sum_{d \in D} ?d.(Q \between !d.\mathbf{1} \between \xi \between !e.\mathbf{1})) + (\sum_{d \in D} ?d.(Q \between !d.\mathbf{1}) + \mathbf{1}) \rightthreetimes (\xi \between !e.\mathbf{1}) \\
= & \{\text{notation}\} \\
 & (\sum_{d \in D} ?d.(Q \between e\xi d)) + (\sum_{d \in D} ?d.(Q \between !d.\mathbf{1}) + \mathbf{1}) \rightthreetimes (\xi \between !e.\mathbf{1}) \\
= & \{\text{axiom MQ}\} \\
 & (\sum_{d \in D} ?d.(Q \between e\xi d)) + (\sum_{d \in D} ?d.(Q \between !d.\mathbf{1}) + \mathbf{1}) \rightthreetimes (\xi \leftthreetimes !e.\mathbf{1} + \xi \rightthreetimes !e.\mathbf{1}) \\
= & \{\text{axiom LMQ7, } \xi \text{ only consists of send actions}\} \\
 & (\sum_{d \in D} ?d.(Q \between e\xi d)) + (\sum_{d \in D} ?d.(Q \between !d.\mathbf{1}) + \mathbf{1}) \rightthreetimes (\xi \rightthreetimes !e.\mathbf{1}) \\
= & \{\text{axioms RMQ6 and SQ1}\} \\
 & (\sum_{d \in D} ?d.(Q \between e\xi d)) + (\sum_{d \in D} ?d.(Q \between !d.\mathbf{1}) + \mathbf{1}) \rightthreetimes (!e.\xi) \\
= & \{\text{axiom RMQ6}\} \\
 & (\sum_{d \in D} ?d.(Q \between e\xi d)) + !e.((\sum_{d \in D} ?d.(Q \between !d.\mathbf{1}) + \mathbf{1}) \between \xi)
\end{array}
$$

$$
\begin{aligned}
&= \quad \{\text{definition Q}\} \\
&\quad \sum_{d \in D} ?d.(Q \between e\xi d)) + !e.(Q \between \xi) \\
&= \quad \{\text{Induction Hypothesis}\} \\
&\quad \sum_{d \in D} ?d.Q_{e\xi d} + !e.Q_{\xi} \\
&= \quad \{\text{definition } Q_{e\xi}\} \\
&\quad Q_{e\xi}
\end{aligned}
$$

And from the base case, trivially: $Q = Q_{\varepsilon}$.