# From data structure to process structure*

Ed Brinksma
Tele-Informatics Group
Dept. of Computer Science, University of Twente
PO Box 217, 7500 AE Enschede, The Netherlands
brinksma@cs.utwente.nl

**Abstract**

This paper deals with transformations in a process algebraic formalism that has been extended with an abstract data type language. We show how for a well-known class of processes (bags, queues, stacks, etc.) descriptions in terms of simple process definitions and complex state parameters can be transformed in a stepwise fashion into equivalent systems of interacting processes with state parameters of reduced complexity. The key to the solution are so-called *context equations.*

## 0 Introduction

Interchanging the complexity of the control structure and the parameter structure of a program is a well-known program transformation principle. In this paper we investigate this principle in the particular setting of a process algebraic formalism that has been complemented with an abstract data type (ADT) language for the definition of data structures. The formalism resembles the ISO specification language LOTOS [BoBr87,ISO89], which contains a dialect of the ADT language ACT ONE [EhMa85]. It abstracts from the concrete syntactic structure of LOTOS, which has been optimized for the structured representation of the functionality of large distributed systems. Our results, however, can be directly translated back to any process algebraic formalism that has a sufficiently related abstract syntactic and semantic definition.

Transformations that exchange the complexity of the state parameters for complexity of the control structure of processes or vice versa are widely studied as methods to improve their structure and/or efficiency. In the design of concurrent systems it is of interest to find transformations that can be used to decompose specifications of processes that are described in terms of an explicit global state into a number of concurrently interacting processes with less complicated local states. In fact, the various formal definitions can be differentiated in terms of their *specification styles* which relate the use of different operator signatures to the (informal) purpose of a specification, e.g. specification-oriented or implementation-oriented, see [VSSB90].

*This work has been partly supported by the CEC as part of the ESPRIT/LOTOSPHERE project (ESPRIT project 2304)

So far, the main contributions of the process algebraic approach to distributed systems has been in providing useful semantic frameworks for proving the *correctness* of such transformations, see e.g. the implementation of a queue in [Mil80] and the correctness of the AB-protocol in [LaMi87]. It is, however, of great methodological interest to have methods to obtain such transformations. It is to this area that we claim to contribute. We show how for a well-known class of processes (bags, queues, stacks, etc.) descriptions in terms of simple process definitions and complex state parameters can be replaced by a systems of interacting processes whose parameters do not exceed the complexity of a single data element. This is done by deriving *context equations* (from the complex-state descriptions) as in [Lar90], but with the important difference that in our set-up the context is the unknown entity. By solving the equations we obtain the building bricks for the desired transformations.

## 1 A process algebraic calculus

We use a process algebraic language that is an abstract version of ISO specification language LOTOS [ISO89]. It is also strongly related to other process algebraic calculi, most notably to TCSP [Hoa85], CIRCAL [Mil85] and CCS [Mil80,Mil89]. Parts of our results will therefore be, *mutatis mutandis*, transferable to other calculi. A more detailed account of the language than is given here can be found in the full paper [BrKa91] and [EhMa85,Bri88,ISO89].

The basic calculus is built around a set *Act* providing the alphabet of *actions*, and a set *PId* of process identifiers. The set *BExpr* of behaviour expressions is defined by the following BNF-schema:

$$B ::= \textbf{stop} \mid a; B \mid \tau; B \mid \sum\{B_i \mid i \in I\} \mid B_1 \parallel_A B_2 \mid B//A \mid B[S] \mid p$$

where $a \in Act$, $\tau \notin Act$, $\{B_i \mid i \in I\}$ an indexed set of behaviour expressions, $A \subseteq Act$, $S : Act \rightarrow Act$, and $p \in PId$. For this language we define attributes such as the label sort $L(B)$ etc. in the usual way. $S$ is extended to $Act \cup \{\tau\}$ by defining $S(\tau) = \tau$. The meaning of the process identifiers $p$ is given by an environment of *process definitions* $PE = \{p_j := B_j \mid j \in J\}$. The SOS-rules defining the operational semantics of the language are contained in table 1.

The main differences with respect to CCS are:

- synchronization is on the basis of *identical* labels, not complementary ones;
- synchronization does not result in the silent action $\tau$, but the action on which was synchronized; this enables synchronization between more than two processes;
- restriction is 'built in' the (indexed) parallel combinator;
- as hiding of actions cannot be achieved via synchronization, this requires an extra combinator; the treatment of parallelism, hiding and restriction is thus closer to that in TCSP [Hoa85];
- the relabelling function $S$ can be non-injective.

Some derived operators of this calculus are:

- $B_1 \mathbin{[]} B_2 =_{df} \sum\{B_1, B_2\}$

| $B$ | **rules** | **condition** |
|---|---|---|
| **stop** | no rules | |
| $\mu; B$ | $\vdash \mu; B \xrightarrow{\mu} B$ | $\mu \in Act \cup \{\tau\}$ |
| $\sum\{B_i \mid i \in I\}$ | $B_i \xrightarrow{\mu} B_i' \vdash \sum\{B_i \mid i \in I\} \xrightarrow{\mu} B_i'$ | $i \in I$ |
| $B_1 \parallel_A B_2$ | $B_1 \xrightarrow{\mu} B_1' \vdash B_1 \parallel_A B_2 \xrightarrow{\mu} B_1' \parallel_A B_2$ | $\mu \notin A$ |
| | $B_2 \xrightarrow{\mu} B_2' \vdash B_1 \parallel_A B_2 \xrightarrow{\mu} B_1 \parallel_A B_2'$ | $\mu \notin A$ |
| | $B_1 \xrightarrow{\mu} B_1', B_2 \xrightarrow{\mu} B_2' \vdash B_1 \parallel_A B_2 \xrightarrow{\mu} B_1' \parallel_A B_2'$ | $\mu \in A$ |
| $B//A$ | $B \xrightarrow{\mu} B' \vdash B//A \xrightarrow{\mu} B'//A$ | $\mu \notin A$ |
| | $B \xrightarrow{\mu} B' \vdash B//A \xrightarrow{\tau} B'//A$ | $\mu \in A$ |
| $B[S]$ | $B \xrightarrow{\mu} B' \vdash B[S] \xrightarrow{S(\mu)} B'[S]$ | |
| $p$ | $B \xrightarrow{\mu} B' \vdash p \xrightarrow{\mu} B'$ | $p := B \in PE$ |

Table 1: SOS rules for the basic calculus

- $B_1 \parallel B_2 =_{df} B_1 \parallel_{Act} B_2$
- $B_1 \,|||\, B_2 =_{df} B_1 \parallel_{\emptyset} B_2$
- **exit** $=_{df} \delta;$ **stop** for a reserved action name $\delta \in Act$ marking *successful termination*;
- $B_1 \gg B_2 =_{df} (B_1[ok/\delta] \parallel_{\{ok\}} ok; B_2)//\{ok\}$, where $ok \notin L(B_1) \cup L(B_2)$.

To extend this basic calculus with value-passing and parameterization constructs it is combined with an abstract data type formalism. This formalism is used to define a $\Sigma$-algebra $A$ for a signature of sorts and operations $\Sigma$, providing a *data-type environment* for the specification of process behaviour. The signature is used to generate terms with which data-values can be represented. The sort *bool* of Boolean values with constants *true* and *false* is assumed to be predefined. For every sort $s$, an operation if_then_else_ : $bool, s, s \rightarrow s$ is implicitly defined.

In this setting we can endow the elements of $Act$ and $PId$ with some substructure, viz.

- $Act =_{df} \{a_v \mid a \in L, v \in D\}$, where $L$ is a set of *port/gate/label-names* and $D$ is the domain of the defined $\Sigma$-algebra $A$;
- $PId =_{df} \{p_v \mid p \in P, v \in D^*\}$, where $P$ is a set of *process names.*

In the extended language the attribute $L(B)$, and indexed combinators like $\parallel_A$ and $//A$ refer to subsets of $L$, but should, as usual, be interpreted as their obvious extensions to $Act$.

The language of behaviour expressions is then extended with a number of new constructs:

- $a?x : s; B(x) =_{df} \sum\{a_v; B(t_v) \mid v \in D_s\}$, where $D_s \subseteq D$ is the subdomain of sort $s$ in $A$, and $t_v$ is a term with value $v$; we write $a(-); B$ if the name of the variable is immaterial (e.g. when the subsequent behaviour $B$ does not depend on it);
- $a!t; B =_{df} a_v; B$, where $v$ is the value of term $t$;
- $[t] \rightarrow B =_{df}$ if $A \models t = true$ then $B$ else **stop**, for Boolean terms $t$.

Process definitions are generalized to the format

$$p(x_1 : s_1, \ldots, x_n : s_n) := B(x_1 : s_1, \ldots, x_n : s_n)$$

which are interpreted as sets of elementary process definitions, viz.

$$\{p_v := B(t_1, \ldots, t_n) \mid v = \langle val(t_1), \ldots, val(t_n)\rangle \in D_{s_1} \times \ldots \times D_{s_n}\}.$$

In the context of this paper we will need the equations over the behaviour expressions induced by the *strong bisimulation equivalence* $\sim$ and the *observation congruence* $\approx^c$ (see e.g. [Mil89]). The resulting laws are the expected analogies of the laws as they are known for related calculi. The full paper [BrKa90] contains a list of those that are needed for the proofs of our results. The reader may also consult [Bri88,ISO89] on this matter.

## 2 Context equations

In this paper an important role is played by the concept of a *context*. It can be most easily imagined as a behaviour expression with a number of *holes* in it. It is convenient to introduce a set *Var* of process variables, whose elements we will denote with $X, Y, \ldots$. A context $C[X_1, \ldots, X_n]$ then is a behaviour expression in which the variables $X_1, \ldots, X_n$ may occur as sub-behaviour expressions. To deal successfully with issues of infinity due to infinite value domains, we allow contexts $C$ that are parameterized with possibly infinite (indexed) sets of process variables, which we denote by $C[X_i | i \in I]$. A context $C[X_i | i \in I]$ is *(weakly) guarded* if every occurrence of an $X_i$ $(i \in I)$ in $C$ is contained in a subexpression $\mu; B$ with $\mu \in Act \cup \{\tau\}$. $C[X_i | i \in I]$ is *observably guarded* if every occurrence of an $X_i$ $(i \in I)$ in $C$ is contained in a subexpression $a; B$ with $a \in Act$. A system of *context equations* is a set of the form $\{X_i = C_i[X_i | i \in I] \mid i \in I\}$ where $=$ denotes an appropriate instance of equivalence, in our case $\sim$ or $\approx^c$. The proof of the following theorem is a simple variation of analogous ones existing in the literature, e.g. the one in [Mil88].

**Theorem 1**
Let $\{X_i \sim C_i[X_i | i \in I] \mid i \in I\}$ be a system of weakly guarded context equations then there exists a unique set of solutions (modulo $\sim$) $\{B_i \mid i \in I\}$ such that $B_i \sim C_i[B_i | i \in I]$ for all $i \in I$, viz. $B_i = p_i$ defined by the process environment $\{p_i := C_i[p_i | i \in I] \mid i \in I\}$. □

The generalization of this theorem to the $\approx^c$ is more specific to the combinator signature of our calculus and results from the application of theorem 4.8.5 in [Bri88].

**Theorem 2**
Let $\{X_i \approx^c C_i[X_i | i \in I] \mid i \in I\}$ be a system of observably guarded context equations *not containing applications of the hiding operator* $//A$, then there exists a unique set of solutions (modulo $\approx^c$) $\{B_i \mid i \in I\}$ such that $B_i \approx^c C_i[B_i | i \in I]$ for all $i \in I$, viz. $B_i = p_i$ defined by the process environment $\{p_i := C_i[p_i | i \in I] \mid i \in I\}$. □

The transformations between data-oriented specifications and process-oriented specifications as studied in this paper build on solving context equations *with unknown contexts*, i.e. to determine $C[X]$ such that $C[B_1] = B_2$ for known $B_1, B_2$. It will be sufficient in this case to work with contexts $C$ that contain one occurrence of a process variable. Just as behaviours can be specified by listing the involved transitions $B \xrightarrow{a} B'$, such contexts

$$\begin{array}{rl}
\mathit{Multiset(Nat)} = & \mathit{Nat} + \\
\textbf{sorts}: & \mathit{mult} \\
\textbf{opns}: & \emptyset :\rightarrow \mathit{mult} \\
 & \mathit{add}, \mathit{rem} : \mathit{nat}, \mathit{mult} \rightarrow \mathit{mult} \\
 & \_\in\_ : \mathit{nat}, \mathit{mult} \rightarrow \mathit{bool} \\
\textbf{eqns}: & x, y : \mathit{nat}; m, n : \mathit{mult} \\
 & \mathit{add}(x, \mathit{add}(y, m)) = \mathit{add}(y, \mathit{add}(x, m)) \\
 & \mathit{rem}(x, \emptyset) = \emptyset \\
 & \mathit{rem}(x, \mathit{add}(y, m)) = \text{if } \mathit{eq}(x, y) \text{ then } m \text{ else } \mathit{add}(y, \mathit{rem}(x, m)) \\
 & x \in \emptyset = \mathit{false} \\
 & x \in \mathit{add}(y, m) = \text{if } \mathit{eq}(x, y) \text{ then } \mathit{true} \text{ else } x \in m
\end{array}$$

Table 2: *Multiset*(*Nat*)

may be characterized by *transductions*, see e.g. [Lar90]. Let *Con* be a set of *context variables* whose elements we denote by $C, D, E, \ldots$ , then we write such transductions as $C -[a/b]\rightarrow C'$, meaning that context $C$ can change into context $C'$ by consuming action $b$ (from a process that is substituted for $X$ in $C[X]$) and producing an action $a$. It corresponds to an SOS-inference rule of the form $X -b\rightarrow X' \vdash C[X] -a\rightarrow C'[X']$. If a context moves independently of the process in it, this is denoted by $C -[a/0]\rightarrow C'$, which corresponds to the SOS-rule $\vdash C[X] -a\rightarrow C'[X]$. The following theorem is at the heart of our transformational methods.

**Theorem 3**
Let *Trans* be a set of transductions over *Con* of the form $C -[a/0]\rightarrow C'$ or $C -[a/a]\rightarrow C'$. Let $M \subseteq L$, $M' =_{df} \{a' \mid a \in M\}$, where $' : M \rightarrow L$ is an injection such that $M \cap M' = \emptyset$, and let $S_M : L \rightarrow L$ with $S_M(a) = a$ $(a \notin M')$ and $S_M(a') = a$ $(a' \in M')$, and $\{p_C\}_{C \in Con}$ a family of process identifiers defined by

$$\begin{array}{l}
p_C := \sum\{a'; p_{C'} \mid C -[a/0]\rightarrow C' \in \mathit{Trans}\} \;[] \\
\qquad\; \sum\{a; p_{C'} \mid C -[a/a]\rightarrow C' \in \mathit{Trans}\}
\end{array}$$

then for all $C \in Con$, and for all $X \in BExpr$ such that $L(X) \subseteq M$

$$C[X] \sim (p_C \parallel_M X)[S_M] \qquad (1)$$

## 3 The bag

We will start with what is arguably the simplest of the *reactive data structures* that we will study: the *bag*. The definition of the corresponding data type, the multiset of natural numbers can be found in table 2.

The specification of the bag that we wish to transform is:

$$\begin{array}{ll}
\mathit{Bag} & := \mathit{MSet}(\emptyset) \\
\mathit{MSet}(v : \mathit{mult}) & := \mathit{in}?x : \mathit{nat}; \mathit{MSet}(\mathit{add}(x, v)) \\
 & \qquad [] \sum\{\mathit{out}!w; \mathit{MSet}(\mathit{rem}(w, v)) \mid w \in v\}
\end{array} \qquad (2)$$

To transform this specification we will try to find a parameterized context $C_x[X]$ with $x : nat$ and

$$C_x[\mathit{MSet}(v)] \sim \mathit{MSet}(\mathit{add}(x, v)) \qquad (3)$$

The intuitive idea behind (3) is that we simulate the effect of the *add* operation on multisets by the context $C_x[X]$. By studying the transitions of $MSet(add(x,v))$ we find that it is sufficient if $C_x[X]$ satisfies the following transductions:

$$\begin{array}{ll} C_x -[out_x/0]\rightarrow I & \\ C_x -[a_w/a_w]\rightarrow C_x & a \in \{in, out\}, w \in D_{nat} \end{array} \tag{4}$$

where $I$ denotes the *identity context* that is completely defined by $I -[a/a]\rightarrow I$ for all $a \in Act \cup \{\tau\}$.

Applying theorem 3 we find the following solution for $C_x[X]$ under the assumption that $L(X) \subseteq \{in, out\}$:

$$\begin{array}{rl} C_x[X] & := (p_C(x) \parallel_{\{in,out\}} X)[out/out'] \\ \text{with}\; p_C(x) & := out'!x; p_I \;[]\; in(-); p_C(x) \;[]\; out(-); p_C(x) \\ p_I & := in(-); p_I \;[]\; out(-); p_I \end{array} \tag{5}$$

We can now immediately derive a solution to our transformation problem. Define

$$NewBag := in?x{:}nat; C_x[NewBag] \tag{6}$$

then we have the following theorem, which is obtained by applying theorem 1.

**Theorem 4** $\quad Bag \sim NewBag$

When we expand (6) in its full form we get the following definition for *NewBag*

$$\begin{array}{rl} NewBag & := in?x{:}nat; (p_C(x) \parallel_{\{in,out\}} NewBag)[out/out'] \\ \text{with}\; p_C(x) & := out'!x; p_I \;[]\; in(-); p_C(x) \;[]\; out(-); p_C(x) \\ p_I & := in(-); p_I \;[]\; out(-); p_I \end{array} \tag{7}$$

Analysing the structure of $p_C(x)$ a bit we can prove the following lemma by application of standard laws for $\sim$ and theorem 1.

**Lemma 5** $\quad p_C(x) \sim out'!x; \mathbf{stop} \;|||\; p_I \qquad (8)$

With this result we can get (7) in a much more agreeable form by using that $p_I$ imitates the identity context $I$.

**Theorem 6** $\quad NewBag \sim in?x{:}nat; (out!x; \mathbf{stop} \;|||\; NewBag) \qquad (9)$

**Corollary 7** $\quad Bag \sim NewBag \sim BestBag$
where $BestBag := in?x{:}nat; (out!x; \mathbf{stop} \;|||\; BestBag)$

This solution is of course well-known in the process-algebraic literature, and therefore we proceed with less transparent cases to show the usefulness of our method.

## 4 The queue

The data-oriented specification of an ordinary (FIFO) queue is parameterized by values of the main sort of the data type *String* defined in table 3.

The corresponding process definition is

$$\begin{array}{rl} Queue & := FIFO(empty) \\ FIFO(s{:}string) & := in?x{:}nat; FIFO(add(x,s)) \\ & \quad [] \; [\neg Empty(s)] \rightarrow out!first(s); FIFO(rest(s)) \end{array} \tag{10}$$

| | |
|---|---|
| $String(Nat) =$ | $Nat +$ |
| **sorts :** | $string$ |
| **opns :** | $empty : \rightarrow string$ |
| | $add : nat, string \rightarrow string$ |
| | $rest : string \rightarrow string$ |
| | $first : string \rightarrow nat$ |
| | $Empty : string \rightarrow bool$ |
| **eqns :** | $x : nat; s, t : string$ |
| | $Empty(empty) = true$ |
| | $Empty(add(x, s)) = false$ |
| | $first(empty) = 0$ |
| | $first(add(x, s)) = \text{if } Empty(s) \text{ then } x \text{ else } first(s)$ |
| | $rest(empty) = empty$ |
| | $rest(add(x, s)) = \text{if } Empty(s) \text{ then } empty \text{ else } add(x, rest(s))$ |

Table 3: *String*(*Nat*)

We could try to follow the same approach as with the *Bag*, i.e. to find a context $D_x[X]$ simulating the effect of the *add* operation, such that $D_x[FIFO(s)] \sim FIFO(add(x, s))$. If we try this we find that the transductions $D_x -[in_w/0]\rightarrow D_w \circ D_x$ are necessary because $add(w, add(x, s)) \neq add(x, add(w, s))$, unlike the addition of elements to a *Bag*. Instead of trying to construct contexts that satisfy such transductions we decompose the queue in a different way. We search for contexts $D_x[X]$ such that for $s \neq empty$

$$D_{first(s)}[FIFO(rest(s))] \sim FIFO(s) \tag{11}$$

The idea here is to decompose the queue into the next element that can be taken from the queue and the rest, instead of the element last put into the queue and the rest. Analysing the behaviour of $FIFO(s)$ we find that the following transductions define a context satisfying (11).

$$\begin{array}{ll} D_x -[out_x/0]\rightarrow I & \\ D_x -[in_w/in_w]\rightarrow D_x & \qquad w \in D_{nat} \end{array} \tag{12}$$

i.e. the output is performed by the context, which then disappears, whereas the inputs are delegated to the process in the context (i.e. the rest of the queue). Again, we have transductions that satisfy the general format for which we have a standard solution under the assumption that $L(X) \subseteq \{in, out\}$, viz.:

$$\begin{array}{rl} D_x[X] & := (p_D(x) \parallel_{\{in,out\}} X)[out/out'] \\ \text{with } p_D(x) & := out'!x; p_I \,[]\, in(-); p_D(x) \\ p_I & := in(-); p_I \,[]\, out(-); p_I \end{array} \tag{13}$$

As before, we can now derive a first solution quite simply by:

$$NewQueue := in?x{:}nat; D_x[NewQueue] \tag{14}$$

**Theorem 8** $\quad Queue \sim NewQueue$

As before we will try to simplify this solution by transforming it.

**Lemma 9** $\quad p_D(x) \sim (out'!x; (out(-))^\omega) \,|||\, (in(-))^\omega$
where $a(-)^\omega := a(-); a(-)^\omega$

We thus obtain our simplification.

**Theorem 10** $NewQueue \sim in?x\!:\!nat; (out'!x; (out(-))^{\omega} \;||_{\{out\}}\; NewQueue)[out/out']$ (15)

In the case of the *Bag* we were done with our transformations after having proved simplification (9). The resulting specification *BestBag* could be regarded as a resource-oriented specification in which after each receipt of a new data element a new memory cell is allocated to store it and offer it to the environment independently (interpretation of $|||$) of the subsequent behaviour. The equivalence (15), however, is still constraint-oriented [VSSB90,Bri89] in style, where after the storage of a new element the subsequent behaviour is constrained via $||_{\{out\}}$ by the memory cell. It is interesting as an example of how one can specify infinite FIFO-queues without using heavy parameterized process definitions and without introducing internal moves $\tau$ into the specification.

To obtain a resource-oriented variant of this specification that corresponds more closely to an implementation in terms of elementary, communicating memory cells, we want to replace the constraints in the form of continuous synchronization by a number of more sparsely exchanged signals. The basic idea is to replace the constraining behaviour $out'!x; (out(-))^{\omega}$ by $out'!x; ok; \mathbf{stop}$, where $ok$ signals the end of constraints on the occurrence of *out*-actions. The synchronizing occurrence of $ok$ is placed immediately before all *out*-actions that currently have to synchronize with the first *out*-action of $(out(-))^{\omega}$, and $||_{\{out\}}$ is replaced by $||_{\{ok\}}$. The new *ok*-action is hidden to make it invisible for the environment.

In the full paper [BrKa91] we carry out the proposed transformation in detail, where the main ingredient is the application of theorem 2, besides some more specialized operations on contexts. The resulting theorem is as follows.

**Theorem 11**
$$\begin{aligned} & NewQueue \approx^{c} AuxQueue//\{ok\} \\ \text{with } & AuxQueue := F[AuxQueue] \\ \text{where } & F[X] =_{df} in?x\!:\!nat; ((ok'; out!x; ok; \mathbf{stop} \;||_{\{ok\}}\; X)//\{ok\})[ok/ok'] \end{aligned}$$

**Corollary 12**
$$\begin{aligned} & Queue \sim NewQueue \approx^{c} BestQueue \\ \text{with } & BestQueue := AuxQueue//\{ok\} \\ & AuxQueue \;\; := F[AuxQueue] \end{aligned}$$

## 5 The stack

The monolithic specification of a stack (or LIFO queue) has the same structure as that of the ordinary (FIFO) queue. The only difference is in the definition of the operations of the corresponding data type, which is specified in table 4. For reasons of clarity we have chosen identifiers for the specification of the stack that are in most cases different from those used for the queue. We leave it to the reader to convince himself of the structural similarity between the two specifications.

$$\begin{aligned} Stack \quad & := LIFO(empty) \\ LIFO(s\!:\!stack(nat)) & := in?x\!:\!nat; LIFO(add(x,s)) \\ & \quad\; [] \; [\neg Empty(s)] \rightarrow out!top(s); LIFO(rest(s)) \end{aligned} \tag{16}$$

The decomposition principles that were chosen for the *Bag*, building a context that depends on the last value accepted, and for the *Queue*, a context depending on the next

| | |
|---|---|
| $\mathit{Stack(Nat)} =$ | $\mathit{Nat}\ +$ |
| **sorts :** | $\mathit{stack}$ |
| **opns :** | $\mathit{empty} :\rightarrow \mathit{stack}$ |
| | $\mathit{add} : \mathit{nat}, \mathit{stack} \rightarrow \mathit{stack}$ |
| | $\mathit{rest} : \mathit{stack} \rightarrow \mathit{stack}$ |
| | $\mathit{top} : \mathit{stack} \rightarrow \mathit{nat}$ |
| | $\mathit{Empty} : \mathit{stack} \rightarrow \mathit{bool}$ |
| **eqns :** | $x : \mathit{nat}; s : \mathit{stack}$ |
| | $\mathit{Empty}(\mathit{empty}) = \mathit{true}$ |
| | $\mathit{Empty}(\mathit{add}(x, s)) = \mathit{false}$ |
| | $\mathit{top}(\mathit{empty}) = 0$ |
| | $\mathit{top}(\mathit{add}(x, s)) = x$ |
| | $\mathit{rest}(\mathit{empty}) = \mathit{empty}$ |
| | $\mathit{rest}(\mathit{add}(x, s)) = s$ |

Table 4: *Stack(Nat)*

value to be output, coincide in the case of the *Stack* because of the LIFO discipline. As there is no obvious alternative, we derive the implied context transductions based on the decomposition given by

$$G_x[\mathit{LIFO}(s)] \sim \mathit{LIFO}(\mathit{add}(x, s)) \tag{17}$$

yielding

$$\begin{array}{ll} G_x -[\mathit{out}_x/0]\rightarrow I & \\ G_x -[\mathit{in}_w/0]\rightarrow G_w \circ G_x & w \in D_{\mathit{nat}} \end{array} \tag{18}$$

These transductions do satisfy the requirements for a general solution as defined by theorem 3. This solution is not adequate, however, for our purposes, as the second transduction generates combined contexts $G_{w_1} \circ G_{w_2} \circ \ldots \circ G_{w_n}$ of arbitrary length. This implies that the corresponding processes $p_{G \circ G \circ \ldots \circ G}$ are parameterized by an ever increasing number of parameters $w_1, \ldots, w_n$, i.e. the complexity of the process states is not a priori bounded.

To find again a solution in which the data complexity of each of the concurrent processes is at most that of one data element, we prove another type of solution for context transductions that are like the ones in (18). Their main characteristic is that all contexts that are unequal to the identity context $I$ have only actions that are independent of the initial actions of an argument process, i.e. their transductions are of the form $C -[a/0]\rightarrow C'$.

**Theorem 13**
Let *Trans* be a set of transductions over *Con* of the form $C -[a/0]\rightarrow C'$ or $I -[a/a]\rightarrow I$. Let $\{p_C\}_{C \in \mathit{Con}}$ be a family of process definitions given by

$$\begin{array}{ll} p_C := \sum\{a; p_{C'} \mid C -[a/0]\rightarrow C' \in \mathit{Trans}\} & \text{if } C \neq I \\ p_I := \mathit{ok}; \mathbf{stop} & \end{array}$$

then for all $C \in \mathit{Con} \setminus \{I\}$, $X \in \mathit{BExpr}$ with $\mathit{ok} \notin L(X)$

$$C[X] \approx^c (p_C \parallel_{\{\mathit{ok}\}} \mathit{ok}; X)//\{\mathit{ok}\} \tag{19}$$

□

Taking advantage of the fact that $(p_C \,\|_{\{ok\}}\, ok; X)//\{ok\}$ in facts implements the LOTOS sequential composition $\gg$, redefining $p_I :=$ **exit**, (19) may be reformulated to

$$C[X] \approx^c p_C \gg X \tag{20}$$

If we now turn back to solving (18) we find that there is a solution of the form

$$G_x[X] := (out!x; \mathbf{exit} \;[]\; in?y; q(x,y)) \gg X \tag{21}$$

where $q(x,y)$ corresponds to $p_{C'}$ for $C' = G_y \circ G_x$. To meet our objective of having only processes with at most parameter complexity of a single data element, we must decompose $q(x,y)$ into simpler processes. From (18) and (21) we find that for $w \in D_{nat}$

$$G_x[X] \xrightarrow{in_w} q(x,w) \gg X$$
$$G_x[X] \xrightarrow{in_w} G_w \circ G_x[X]$$

Rewriting $G_w \circ G_x[X]$, using (21), and applying the associativity of $\gg$ , it follows that

$$q(x,w) \gg X \approx ((out!w; \mathbf{exit} \;[]\; in?y; q(w,y)) \gg (out!x; \mathbf{exit} \;[]\; in?y; q(x,y))) \gg X$$

Because $X$ can be an arbitrary process we can conclude

$$q(x,w) \approx (out!w; \mathbf{exit} \;[]\; in?y; q(w,y)) \gg (out!x; \mathbf{exit} \;[]\; in?y; q(x,y))$$

whence

$$in?w; q(x,w) \approx^c in?w; (out!w; \mathbf{exit} \;[]\; in?y; q(w,y)) \gg (out!x; \mathbf{exit} \;[]\; in?y; q(x,y))$$

i.e., $in?w; q(x,w)$ is a solution of

$$X(x) \approx^c in?w; (out!w; \mathbf{exit} \;[]\; X(w)) \gg (out!x; \mathbf{exit} \;[]\; X(x))$$

whose unique solution (modulo $\approx^c$) by theorem 2 is

$$Cell(x) := in?w; (out!w; \mathbf{exit} \;[]\; Cell(w)) \gg (out!x; \mathbf{exit} \;[]\; Cell(x))$$

It follows that $G_x[X] \approx^c (out!x; \mathbf{exit} \;[]\; Cell(x)) \gg X$. Using standard techniques we obtain our final result.

**Theorem 14** $\quad Stack \approx^c NewStack$

$$\begin{aligned} \text{where } NewStack &:= in?x;\ Top(x) \gg NewStack \\ Top(x) &:= (out!x; \mathbf{exit} \;[]\; Cell(x)) \\ Cell(x) &:= in?w;\ Top(w) \gg Top(x) \end{aligned}$$

**Remark**

We could have tried to decompose the process $q(x,y)$ corresponding to $p_{C'}$ for $C' = G_y \circ G_x$ in the context of the solution for (18) generated by theorem 3 instead of that of theorem 13, as we have done now. The reader is encouraged to check that this cannot succeed: the renaming strategy in (1) does not have the associativity properties of $\gg$.

## 6 Conclusion

We have developed a technique to solve certain classes of context equations with unknown contexts in a fairly general process algebraic formalism. Using this technique we have shown how we may obtain specifications with an implementation-oriented substructure by distributing the global state of a high-level monolithic specification based on a single process definition and a complicated parameter structure in a controlled step-by-step fashion. In the full paper [BrKa91] we will report on our experience in applying this technique to more involved examples, and indicate ways to extend our results.

## Acknowledgements

The author gratefully acknowledges the help of SICS, Kim Larsen, Rom Langerak and Pim Kars, who all contributed to the realization of this paper.

## References

[BoBr87] Bolognesi, T., Brinksma, E.: Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems* 14, 22–59 (1987).

[Bri88] Brinksma, E.: *On the design of Extended LOTOS.* Doctoral Dissertation, University of Twente, The Netherlands, 1988.

[Bri89] Brinksma, E.: Constraint-Oriented Specification in a Constructive Formal Description Technique, in: LNCS 430, pp. 130–152, Springer, 1989.

[BrKa91] Brinksma, E., Kars, W.: From data structure to process structure. Memorandum INF-91-38/TIOS-91-11, University of Twente, The Netherlands, 1991.

[EhMa85] Ehrig, H., Mahr, B.: *Fundamentals of Algebraic Specification 1.* Springer, 1985.

[Hoa85] Hoare, C.A.R.: *Communicating Sequential Processes.* Prentice-Hall, 1985.

[ISO89] ISO: *LOTOS, A formal description technique based on the temporal ordering of observational behaviour.* International Standard ISO 8807, 1989.

[Lar90] Larsen, K.G.: Ideal Specification Formalism = Expressivity + Compositionality + Decidability + Testability + ... , in: Baeten, J.C.M., Klop, J.W. (eds.): *CONCUR'90*, LNCS 458, Springer, pp. 33–56 (1990).

[LaMi87] Larsen, K., Milner, R.: Verifying a protocol using relativized bisimulation, in: *Proc. ICALP'87*, LNCS 267, Springer, 1987.

[Mil80] Milner, R.: *A Calculus of Communicating Systems.* LNCS 92, Springer, 1980.

[Mil85] Milne, G.: CIRCAL and the Representation of Communication, Concurrency and Time. *ACM Trans. on Progr. Languages and Systems* 7, 270–298 (1985).

[Mil89] Milner, R.: *Communication and Concurrency.* Prentice-Hall, 1989.

[VSSB91] Vissers, C.A., Scollo, G., Van Sinderen, M., Brinksma, E.: Specification Styles in Distributed Systems Design and Verification, to appear in: Special Issue TCS Tapsoft'89.