

# Distribution of a simple shared dataspace architecture <sup>1</sup>

Simona Orzan      Jaco van de Pol

*CWI, P.O.Box 94079, NL-1090 GB Amsterdam, The Netherlands*  
simona@cw.nl      vdpol@cw.nl

---

## Abstract

We study a simple software architecture, in which components are coordinated by writing into and reading from a global set. This simple architecture is inspired by the industrial software architecture Splice. We present two results. First, a distributed implementation of the architecture is given and proved correct formally. In the implementation, local sets are maintained and data items are exchanged between these local sets. Next we show that the architecture is sufficiently expressive in principle. In particular, every global specification of a system's behaviour can be divided into components, which coordinate by read and write primitives on a global set only. We heavily rely on recent concepts and proof methods from process algebra.

---

## 1 Introduction

The complexity of designing distributed systems is generally managed by introducing a *software architecture*, defining how components are coordinated. By fixing the architecture, two separate tasks can be distinguished. First, the architecture must be implemented on a distributed network. Second, components must be designed that together implement the requirements of the system under design, using the coordination primitives provided by the architecture. The architecture and its implementation are likely to be reused for other systems in a similar application domain. The choice of architecture is a delicate issue. From the application programmer's point of view a rich set of coordination primitives, and the guarantee of system-wide consistency are preferable. At the same time, this demands much overhead from the distributed implementation, and may lead to bad performance, or even to an unrealizable architecture.

---

<sup>1</sup> Partially supported by PROGRESS, the embedded systems research program of the Dutch Organisation for Scientific Research NWO, the Dutch Ministry of Economic Affairs and the Technology Foundation STW, grant CES.5009.

Inspired by the industrial architecture Splice (see below), we study the consequences of choosing an extremely weak and simple coordination model: communication via a global set. The coordination primitives between components are restricted to writing and reading. We imagine that the application components reside at certain physical locations, abstractly represented by natural numbers. Coordination primitive  $write(i, v)$  represents that the component at location  $i$  adds value  $v$  to the global set; if  $v$  is present already this action has no effect. The other primitive,  $read(i, v)$  denotes a non-destructive, blocking read of a particular value (or template)  $v$  by a component at location  $i$ . That is, it waits until it actually finds  $v$  in the global set, and then proceeds. Note that test for absence and deletion of items is not possible.

From the separate tasks we mentioned before, two natural questions on architectures arise, which are addressed in this paper. The *first question* is whether the architecture itself has an efficient distributed implementation. This is addressed in section 3. We define a distributed implementation of the architecture, in which every component has its own local set. Data items are exchanged between these local sets asynchronously. We prove that the implementation based on local sets is *behaviourally equivalent* to the specification based on a conceptual global set. The fact that the difference cannot be noticed is mainly due to the careful selection of the weak coordination primitives. This result is essentially the same as in [5,7,8], albeit in a slightly more general setting. However, the proof is much simpler due to the application of powerful process algebraic proof principles.

The *second question* is whether the architecture is sufficiently expressive to allow the distributed implementation of any system specification. This is investigated in section 4 from a functional point of view – i.e. without taking into account issues like performance or fault tolerance. We show that every specification of functional behaviour has a distributed implementation, i.e. one where different types of actions are performed at different physical locations. In particular, the components only use the weak coordination primitives  $read$  and  $write$  on a global set. As far as we know, this main result is not comparable to existing results on expressiveness of coordination models.

As an example, consider the very simple logging system, with its behavioural specification  $input.log$ , indicating that some  $input$  action precedes some  $log$  action. This system probably uses two physical devices (e.g. a monitor and an actuator) with their own controllers, so  $input$  and  $log$  happen at different locations. A distributed implementation with our primitives could be:  $input.write(l_1, d) \parallel read(l_2, d).log$ . Here  $l_1$  and  $l_2$  are the locations of the components, and  $d$  is some data value. With  $\parallel$  we denote parallel composition. Assuming that the system starts with the empty data space, the second process is initially blocked, so the only execution of this little program should be  $input.write(l_1, d).read(l_2, d).log$ . If we hide the communication actions  $read$  and  $write$ , we indeed get the desired system behaviour  $input.log$ . We remark that the system  $button_1 + button_2$ , in which non-deterministically

either  $button_1$  or  $button_2$  is pressed, also has a distributed implementation, but this is much harder. In particular, our solution will use an unbounded number of internal communications.

### 1.1 Relationship with Splice

The choice of architecture in this paper is influenced by Splice [6] (Subscription Paradigm for the Logical Interconnection of Concurrent Engines). Splice is a data-oriented software architecture for complex control systems, developed and used at the company Hollandse Signaalapparaten bv (currently Thales Nederland). Components are considered as publishers of, and subscribers to data. Each component is accompanied by an agent, which stores data items locally, and forwards these to agents of subscribed components. The advantage of the Splice architecture is that the components are loosely coupled, thus increasing the amount of fault tolerance [6]. The data is present at several locations, making replication of components relatively easy.

Recent research papers propose to view Splice conceptually as a shared data space, i.e. a set of data common to all components [5,12]. Viewing the data as a global data space has the advantage that all programs perceive the same data at any moment. In addition, viewing it as a set (instead of a multi-set) opens the way to *transparent* replication of components [12]. See section 5.1 for further related work.

### 1.2 A Process-algebraic Approach

A common theme has been to embed the coordination primitives in a host language and give semantics to the resulting coordination language. As an alternative, we adopt a process algebraic point of view. In this view everything is a process, or more precisely: the behaviour of every system can be modeled as process. A system can be modeled as a process at various abstraction levels. Typically, two descriptions are distinguished: *Spec* and *Impl*. The process *Spec* specifies the global behaviour of the system, whereas the process *Impl* describes its implementation, typically as the parallel composition of certain communicating processes. The typical process algebraic correctness statement is then:  $Spec = \tau_I(Impl)$ , i.e. the specification is behaviourally equivalent to the implementation, after abstraction of internal communications in  $I$ .

In our case, the components of the application are processes. Also the architecture itself will be a process; we will define our architecture as the process GSRW (global set with read and write) in section 2.1. Our first problem is to find a distributed implementation of GSRW, called DSRW (Distributed sets with read and write) together with a proof that  $GSRW = \tau_I(DSRW)$ . The second problem requires for any specification of a system's global behaviour  $B$ , a number of components  $P_i$  (satisfying certain syntactic criteria on locations) such that  $B = \tau_I(GSRW || P_1 || \dots || P_n)$ .

We have chosen the process algebraic approach for a number of reasons.

First, it clarifies the concepts. By choosing a formalism, rather vague claims on realizability and expressiveness are turned into clear theorems. Process algebra provides the means to focus on the essential interfaces, by distinguishing external and internal actions, and by encapsulation of data in processes. The next advantage is that our approach yields rigorous formal proofs, apt for mechanic verification. The full proofs are available in technical reports [23,24]. The third advantage is that we can use powerful proof principles developed for process algebra. Finally, by using a standard process algebra existing tools ([4]) can be used for simulation and model checking. This has been demonstrated in [20,25].

## 2 Preliminaries: Process Algebra with Data

For good introductions to process algebra see [1,14]. We will present and prove our ideas using the formalism  $\mu\text{CRL}$  [18], which is a combination of the standard process algebra ACP [2] with abstract data types.

### 2.1 GSRW in the syntax of $\mu\text{CRL}$

Processes are built from atomic actions (e.g. *input*, *log*) by certain connectives. In particular,  $\mu\text{CRL}$  inherited the typical process algebra connectives from ACP. For any processes  $p$  and  $q$ ,  $p + q$  denotes non-deterministic choice between  $p$  and  $q$ ,  $p \cdot q$  denotes their sequential composition, and  $p \parallel q$  denotes the parallel composition (defined in terms of interleaving and synchronous communication). The operators encapsulation ( $\partial_H$ ) and hiding ( $\tau_I$ ) will be explained later. Two special processes are  $\delta$  (deadlock, the unit of  $+$ ) and  $\tau$  (internal action).

Besides processes, a  $\mu\text{CRL}$  specification contains abstract data types. A signature of multiple sorts and functions can be declared, and axiomatized by equations. We will tacitly assume the following standard sorts with the usual operations: *Bool* (booleans), *Nat* (natural numbers, to represent locations), *D* (to represent data values, intentionally left unspecified) and *Set* (finite sets over *D*). For  $A : \text{Set}$  and  $v : D$ ,  $A + v$  denotes  $A \cup \{v\}$ . It is routine to specify these types algebraically.

The following connectives connect processes with abstract data types. First, atomic actions can be parameterized with data elements, as in  $\text{read}(v)$ . Then,  $\sum_{d \in D} p(d)$  denotes alternative (possibly infinite) choice over data domain *D*. Finally, if  $b$  is a term of data domain *Bool* and  $p$  and  $q$  are processes, then the conditional  $(p \triangleleft b \triangleright q)$  is the process “ $p$  if  $b$ , else  $q$ ”.

We now formally define GSRW. To this end we introduce the parameterized atomic actions  $\text{Read}(i : \text{Nat}, v : D)$  and  $\text{Write}(i : \text{Nat}, v : D)$ , where  $i$  denotes the location (or: service access point) and  $v$  the datum. Given these basic actions, the architecture GSRW is now defined by the following recursive specification, parameterized with the current set  $A$  of values of sort *D*:

$$\begin{aligned} \text{GSRW}(A : \text{Set}) &= \sum_{i:\text{Nat}} \sum_{v:D} \text{Write}(i, v). \text{GSRW}(A + v) \\ &+ \sum_{i:\text{Nat}} \sum_{v:D} \text{Read}(i, v). \text{GSRW}(A) \triangleleft v \in A \triangleright \delta \end{aligned}$$

Thus, **GSRW** maintains the global set  $A$ . At any moment this process allows that either an element is written, or a value can be read, *provided* it is actually present in  $A$ . In this way, the blocking character of *read* is captured.

Application processes can read and write by synchronizing with the *Read* and *Write* of **GSRW**. To this end we introduce the actions  $\text{read}(i : \text{Nat}, v : D)$  and  $\text{write}(i : \text{Nat}, v : D)$ . These actions should synchronize (cf. function calls or method invocations), so we define the communication function as follows:  $\text{Read} \mid \text{read} = R$  and  $\text{Write} \mid \text{write} = W$ . As usually in  $\mu\text{CRL}$ , the unsynchronized actions are *encapsulated* by the  $\partial_{\{\text{Read}, \text{read}, \text{Write}, \text{write}\}}$  construct (in order to enforce communication), and the internal communications are *hidden* using the  $\tau_{\{R, W\}}$  construct (in order to abstract from internal detail). The semantics of the previous example from the introduction is now captured formally by the following  $\mu\text{CRL}$ -expression:

$$\tau_{\{R, W\}}(\partial_{\{\text{Read}, \text{Write}, \text{read}, \text{write}\}}(\text{GSRW}(\emptyset) \parallel \text{input.write}(l_1, d) \parallel \text{read}(l_2, d). \text{log}))$$

And indeed, it is a trivial exercise to prove that this is behaviourally equivalent to the specification  $\text{input.log}.\delta$  (termination is not preserved).

## 2.2 Proof Methods from Process Algebra

We noted already that the typical process algebraic notion of refinement is given by the equation  $\tau_I(\text{Impl}) = \text{Spec}$ . As equivalence relation between processes we use branching bisimulation [16], which is slightly finer than weak bisimulation. Note that our results also apply to weak bisimulation. In [18] branching bisimulation on  $\mu\text{CRL}$  processes is axiomatized algebraically. Recent papers developed more practical proof methods that will be used here. These methods are related to a particular process format, called *linear process equation*.

### 2.2.1 Linear Process Equations and Invariants

In [17] it is demonstrated that a large class of  $\mu\text{CRL}$  specifications can be transformed to linear process equations (LPE). Process terms have an implicit notion of state. The point of the LPE format is that the state is encoded explicitly in a data vector. An LPE is essentially a list of condition-action-effect triples. Given an index  $i$  from a finite index set  $J$ , action  $a_i$  with data parameter  $f_i(d, e_i)$  is enabled in state  $d$ , if  $b_i(d, e_i)$  holds. This action leads to the next state  $g_i(d, e_i)$ . Here  $e_i$  is a local variable, used to encode arbitrary

input. Formally, an LPE is a recursive specification of the following form:

$$\text{Impl}(d : D) = \sum_{i \in J} \sum_{e_i : E_i} a_i(f_i(d, e_i)). \text{Impl}(g_i(d, e_i)) \triangleleft b_i(d, e_i) \triangleright \delta$$

The advantage of this format is that properties and proof methods can be uniformly expressed, in terms of the state  $d$  and the constituents  $f_j$ ,  $g_j$  and  $b_j$ .

We assume a special action  $\tau$ , denoting hidden steps. An LPE is *convergent*, if it doesn't admit infinite sequences of  $\tau$ -steps. The principle CL-RSP (Recursive Specification Principle for Convergent LPEs) [3,18] states that a convergent LPE has a unique solution. A predicate  $I(d)$  is an *invariant* if and only if it is preserved by all transitions, formally iff the following conjunction holds:

$$\bigwedge_{i \in J} \forall (d : D, e_i : E_i). I(d) \wedge b_i(d, e_i) \rightarrow I(g(d, e_i))$$

In [19,18] the *focus and cones* method is described for proving equality between implementation and specifications, which we recall in the next section. This method is only applicable in case of convergent LPEs. If  $\tau$ -loops exist, we need a fairness assumption on executions in order to ensure that eventually an exit from the  $\tau$ -loop is chosen. To this end, a fairness rule will be introduced in section 2.2.3.

### 2.2.2 State mappings, Cones and Focus Points

The summands of *Impl* above can be split into internal  $\tau$  steps and external steps,  $J = \text{Int} \uplus \text{Ext}$ , where  $\text{Int} = \{i \in J \mid a_i = \tau\}$ . Besides the implementation, we assume a given specification:

$$\text{Spec}(d' : D') = \sum_{a_i \in \text{Ext}} \sum_{e_i : E_i} a_i(f'_i(d', e_i)). \text{Spec}(g'_i(d', e_i)) \triangleleft b'_i(d', e_i) \triangleright \delta$$

Note that the specification must not contain  $\tau$ -steps. We also assume that the implementation is convergent. Then every state has internal steps to a focus point, i.e. one in which no further  $\tau$ -steps are possible. The focus points can be easily characterized by the focus condition:  $\text{FC}(d) = \bigwedge_{i \in \text{Int}} \neg \exists (e_i : E_i). b_i(d, e_i)$ .

An implementation and a specification in the format above can be proved behaviourally equivalent by providing a state mapping  $h : D \rightarrow D'$ , and proving that the matching criteria  $\text{MC}_h(d)$  hold, where  $\text{MC}_h(d)$  is defined as the conjunction of the following:

- (i) for each  $i \in \text{Int}$ ,  $\forall (e_i : E_i). b_i(d, e_i) \rightarrow h(d) = h(g_i(d, e_i))$   
i.e. internal steps don't change the related state.
- (ii) for each  $i \in \text{Ext}$ ,  $\forall (e_i : E_i). b_i(d, e_i) \rightarrow b'_i(h(d), e_i)$   
i.e. the specification can mimic all external steps of the implementation (soundness).

- (iii) for each  $i \in Ext$ ,  $\forall(e_i : E_i). b'_i(h(d), e_i) \wedge FC(d) \rightarrow b_i(d, e_i)$   
i.e. each external step of the specification can be mimicked in the related focus points of the implementation (completeness).
- (iv) for each  $i \in Ext$ ,  $\forall(e_i : E_i). b_i(d, e_i) \rightarrow f_i(d, e_i) = f'_i(h(d), e_i)$   
i.e. the data labels on the external transitions coincide.
- (v) for each  $i \in Ext$ ,  $\forall(e_i : E_i). b_i(d, e_i) \rightarrow h(g_i(d, e_i)) = g'_i(h(d), e_i)$   
i.e. the next states after a visible transition are related.

**Theorem 2.1** (from [19]) *For specification and convergent implementation in the format above, and given a state mapping  $h$  and an invariant  $I$  such that  $I(d)$  holds and  $\forall(d : D). I(d) \rightarrow MC_h(d)$ , we have*

$$Spec(d) \triangleleft FC(d) \triangleright \tau.Spec(d) = Impl(h(d)) \triangleleft FC(d) \triangleright \tau.Impl(h(d))$$

The essence of this proof method is that given a state mapping  $h$ , and invariant  $I$ , the correctness proof boils down to a check of a number of simple criteria.

### 2.2.3 Fair abstraction

The focus and cones method only works for convergent LPEs. But we will encounter  $\tau$ -loops of arbitrary length. In order to eliminate these loops, we need a fairness principle, which states that eventually an exit of the loop is chosen. For this we will use Koomen's Fair abstraction rule (KFAR <sub>$n$</sub>  for  $n > 1$ ) [1]. Assume we have a  $v$ -loop with exits, of the following form:

$$\begin{aligned} X_1 &= v.X_2 + s_1 \\ X_2 &= v.X_3 + s_2 \\ &\dots \\ X_n &= v.X_1 + s_n \end{aligned}$$

Then after abstraction from  $v$  we would get a non-convergent LPE. However, according to KFAR <sub>$n$</sub>  we are sure that after some time one of the exits  $s_i$  is taken, so we get:

$$\tau.\tau_{\{v\}}(X_1) = \tau.\tau_{\{v\}}(s_1 + \dots + s_n)$$

## 3 Distributed Implementation

In this section a distributed implementation of GSRW is defined and a correctness proof is given. We first introduce the data type *List*, representing a list of local data spaces. It has constructors  $\epsilon$  (empty list) and  $::$  (cons). The elements of the lists are sets of values. The lists are specified in such a way that they “grow on demand”. We write  $L_i$  for the  $i$ -th element of  $L$  (counting

from 0). If  $i$  exceeds the length of  $L$ , then  $L_i$  is taken to be the empty set. With  $L[i : +v]$  we denote the list  $L_0, \dots, L_{i-1}, L_i + v, L_{i+1}, \dots, L_n$ . When necessary,  $L[i : +v]$  extends  $L$  with empty sets to have length at least  $i$ , and adds  $v$  to  $L_i$ .

$$\begin{aligned}
 \epsilon_i &= \emptyset & \epsilon[0 : +v] &= [\{v\}] \\
 (A :: L)_0 &= A & \epsilon[(i+1) : +v] &= \emptyset :: \epsilon[i : +v] \\
 (A :: L)_{i+1} &= L_i & (A :: L)[0 : +v] &= (A + v) :: L \\
 & & (A :: L)[(i+1) : +v] &= A :: (L[i : +v])
 \end{aligned}$$

In the distributed version **DSRW**, each component  $i$  will write to its private set  $K_i$  and reads from its private set  $L_i$ . Elements of  $K_i$  are sent to all the  $L_j$  separately. Hence **DSRW** has as parameters the lists  $K$  and  $L$  and is defined as follows:

$$\begin{aligned}
 \text{DSRW}(K, L : \text{List}) &= \\
 &\sum_{i:\text{Nat}} \sum_{v:D} \text{Write}(i, v). \text{DSRW}(K[i : +v], L) \\
 &+ \sum_{i:\text{Nat}} \sum_{v:D} \text{Read}(i, v). \text{DSRW}(K, L) \triangleleft v \in L_i \triangleright \delta \\
 &+ \sum_{v:D} \sum_{i,j:\text{Nat}} \text{Send}(i, v, j). \text{DSRW}(K, L[j : +v]) \triangleleft v \in K_i \setminus L_j \triangleright \delta
 \end{aligned}$$

According to **DSRW**, written elements are not immediately available. Data items might even arrive in a different order in different processes. Nevertheless, we have the following correctness theorem:

**Theorem 3.1**  $\text{GSRW}(\emptyset) = \tau_{\{\text{Send}\}}(\text{DSRW}(\epsilon, \epsilon))$ .

**Proof.** We view **GSRW** as a specification and  $\tau_{\{\text{Send}\}}(\text{DSRW})$  as its implementation; the latter equals **DSRW** with  $\text{Send}(i, v, j)$  replace by  $\tau$ . By the focus and cones method, it suffices to give a state mapping and an invariant, and check the matching criteria. As state mapping we define  $h(K, L) = (\bigcup K \cup \bigcup L)$ . We need the invariant  $\text{Inv} = \forall i. L_i \subseteq \bigcup K$ , which can be checked easily. The focus condition  $\text{FC}(K, L)$  is  $\neg \exists (i, j, v). v \in K_i \setminus L_j$ . Assuming the invariant, this can be simplified to  $\forall j. L_j = \bigcup K$  (i.e. all written values have arrived and are ready to be read). Convergence of the implementation follows easily: in  $\tau_{\{\text{Send}\}}(\text{DSRW})$  the number  $\sum_i \sum_j \#(K_i \setminus L_j)$  decreases with each  $\tau$ -step. Now the matching criteria are (skipping the trivial ones):

- (1)  $v \in K_i \setminus L_j \rightarrow \bigcup K \cup \bigcup L = \bigcup K \cup \bigcup L[i : +v]$
- (2)  $v \in L_i \rightarrow v \in \bigcup K \cup \bigcup L$
- (3)  $(v \in \bigcup K \cup \bigcup L) \wedge (\forall j. L_j = \bigcup K) \rightarrow (v \in L_i)$
- (4)  $(\bigcup K \cup \bigcup L) + v = \bigcup K[i : +v] \cup \bigcup L$

These can be proved by simple set-theoretic calculations. Initially, we have  $Inv(\epsilon, \epsilon)$  and  $FC(\epsilon, \epsilon)$ , whence the result follows by Theorem 2.1.  $\square$

In fact this means that  $GSRW$  and  $\tau_{\{Send\}}(DSRW)$  are indistinguishable. This is a generalization of [7,5], because the application processes may use non-deterministic choice, recursion, or even use synchronous communication. Our proof is a standard application of the focus and cones method [19].

## 4 Expressiveness

In this section we will investigate the expressiveness of  $GSRW$ , from a system engineering point of view: given the requirements specification of a system under design, can a distributed implementation on  $GSRW$  be constructed? We assume that the requirements specification is given by a description of the global behaviour, and a localization function. The *behavioural specification* is a process  $Spec$ . The alphabet of a process is the set of action labels that occur in it. Let  $A$  be the alphabet of  $Spec$ . We also assume some set  $L$  of locations, describing for instance physical devices. A localization function is a function  $\lambda : A \rightarrow L$ .

A component  $X$  is *consistent with the localization function* if there exists a fixed location  $\ell$ , such that the alphabet of  $X$  contains only the actions *read*, *write* and external actions  $a$  with  $\lambda(a) = \ell$ . For instance, if  $\lambda(scan) \neq \lambda(log)$ , the implementation can have a component with alphabet  $\{read, write, scan\}$  and another with  $\{read, write, log\}$ . This notion can be seen as a syntactic criterion to enforce correct distribution and to enforce that processes can only communicate via the coordination primitives.

A *distributed implementation of  $Spec, \lambda$  on  $GSRW$*  consists of an initial database  $A_0$ , together with a number of components  $X_1, \dots, X_n$  that are consistent with  $\lambda$ , and behave like  $Spec$ , i.e.

$$Spec = \tau_{\{R,W\}} \partial_{\{read, Read, write, Write\}}(GSRW(A_0) \parallel X_1 \parallel \dots \parallel X_n).$$

The matter of distributing functionalities of a requirements specification over more communicating components was also studied in [21] for LOTOS expressions; the synchronization is solved there with message passing, while  $GSRW$  coordinates the components using persistent data.

**Example 4.1** We describe a possible implementation on  $GSRW$  of a very simple buffer specification. For this, we consider the datasort *Queue*, representing a queue of natural numbers (data must be sent out in the same order in which it was scanned). It has the constant *em*, representing the empty queue, and the operations: *push*:  $Nat \times Queue \rightarrow Queue$ , which adds an element to a queue; *pop*:  $Queue \rightarrow Queue$ , which extracts the top element of the (not empty) parameter queue; *top*:  $Queue \rightarrow Nat$ , which returns the top element of the given queue; and *notempty*:  $Queue \rightarrow Bool$ , which adds an element to a queue. The

buffer interacts with the world through the actions  $IN$ , which inputs a natural number to the buffer and  $OUT$ , which outputs a natural number from the buffer. Then the  $\mu CRL$  specification of the buffer is:

$$\begin{aligned} BufSpec(Q: Queue) = \sum_{d: Nat} (IN(d). BufSpec(push(d, Q)) \\ + OUT(top(d)). BufSpec(pop(Q)) \triangleleft notempty(Q) \triangleright \delta) \end{aligned} \quad (1)$$

In order to build an implementation of  $BufSpec$  on  $GSRW$ , we use a global set that memorizes values of sort  $Nat \times Nat$ , representing pairs (sequence number, data item). We instantiate the architecture to  $GSRW(A : Set(Nat \times Nat))$  and we choose a localization function  $\lambda : \lambda(IN) = l_1, \lambda(OUT) = l_2$ . A possible distributed implementation on  $GSRW$  of the buffer is :

$$BufImpl = B_{in}(0) \parallel B_{out}(0) \parallel GSRW(\emptyset) \quad (2)$$

where

$$\begin{aligned} B_{in}(n : Nat) &= \sum_{d: Nat} IN(d).write(l_1, (n, d)).B_{in}(n + 1) \\ B_{out}(n : Nat) &= \sum_{d: Nat} read(l_2, (n, d)).OUT(d).B_{out}(n + 1) \end{aligned}$$

(2) is indeed an implementation of (1), since it can be proved that  $BufSpec(\mathbf{em})$  is branching bisimilar to  $\tau_{\{R, W\}} \partial_{\{Read, Write, read, write\}} BufImpl$ .

#### 4.1 The translation scheme

In the sequel we will show how to construct  $X_i$  and  $A_0$  for any requirements specification. That is, we describe a translation scheme from an LPE  $Spec(d)$  together with a localization function  $L$  to a set of processes  $X_1, \dots, X_n$  and some initial database  $A_0$  satisfying the above criteria. The localization criterion will be solved by mapping each action label to a different component. This results in the maximally distributed, most fine-grained implementation of the given specification, from which an implementation with less parallel components can always be obtained by bundling several components  $X_i$ . Then we will prove that this translation scheme is correct.

We assume that a requirements specification is given in LPE format (see section 2):

$$Spec(d : D) = \sum_{i \in I} \sum_{e_i : E_i} a_i(f_i(d, e_i)).Spec(g_i(d, e_i)) \triangleleft b_i(d, e_i) \triangleright \delta \quad (3)$$

Each summand of  $\sum_{i \in I}$  defines a set of transitions from state  $d$  to state  $g_i(d, e_i)$  and it is enabled for all  $e_i$  for which the guard  $b_i(d, e_i)$  is true. Moreover, we assume a localization function  $\lambda : \{a_i \mid i \in I\} \rightarrow L$  for a set of locations  $L$ .

Let  $n = |I|$ . The distributed implementation will have  $n$  components, each responsible for one action  $a_i$ . They communicate via GSRW, using a global set of *pairs* (*timestamp, data*) of the sort  $Nat \times D$ . The timestamp represents the moment when the pair was added to the database or, in other words, the number of visible + invisible steps executed until the time of insertion. The data is one of the global states of the system.

Components are triggered in turns, by the timestamp, in a circular infinite pass: component  $i$  will be activated at all moments  $t = k \cdot n + i$  ( $\forall k$ ). When activated, it will choose to execute its action or not to execute it. In both cases, it will increase the “global time” and pass the turn to its next sister. This cycle is needed to ensure that the nondeterminism that may exist in the global specification  $Spec(d)$  is preserved in the distributed implementation. At any time, all possible actions must have a chance to execute.

In a formal definition, the component  $X_i$ , responsible of action  $a_i$  is:

$$\begin{aligned}
 X_i(m) = & \sum_{d:D} read(\lambda(a_i), (m, d)). \\
 & (\sum_{e_i:E_i} (a_i(f_i(d, e_i)).write(\lambda(a_i), (m + 1, g_i(d, e_i))) \\
 & \quad \triangleleft b_i(d, e_i) \triangleright \delta) \quad (4) \\
 & + write(\lambda(a_i), (m + 1, d))) \\
 & . X_i(m + n)
 \end{aligned}$$

and the initial state of the implementation is

$$\parallel_i X_i(i) \parallel GSRW(\{(0, d)\}) \quad (5)$$

The parameter  $m$  of  $X_i$  is the moment when  $X_i$  expects to be activated next. As mentioned before,  $m$  is always of the form  $k \cdot n + i$ . At moment  $m$ ,  $read(\ell, (m, d))$  from  $X_i$  synchronizes with  $Read(\ell, (m, d))$  from  $GSRW(A)$ , for some  $d$ . This activates  $X_i$ . After “acting”,  $X_i$  will set its parameter to the next active moment  $(k + 1) \cdot n + i$ , i.e.  $m + n$ . In its life,  $X_i$  passes only through the following local states: 0 –ready to read, 1 –activated; make a choice (execute action or pass the turn), 2 –action performed; pass the turn.

We will prove that this distributed implementation on GSRW of a LPE is almost equivalent to the specification. That is: if we abstract from the actions dealing with the global set  $(R, W)$ , then we get the specification  $Spec(d)$  with an extra initialization step:

**Theorem 4.2** *For every requirements specification expressible as a LPE  $Spec(d)$ , the components  $X_i$  resulted by applying the translation scheme satisfy:*

$$\tau.Spec(d) = \tau.\tau_{\{R,W\}}\partial_{\{Read,Write,read,write\}}(\parallel_i X_i(i) \parallel GSRW(\{(0, d)\})).$$

## 4.2 Correctness proof

This subsection is devoted to proving that the translation defined above is correct. That is, to prove theorem 4.2. First of all, to be able to compare the two processes appearing in the theorem, we need to bring the implementation (5) to a linearized form (the specification  $Spec(d)$  already is, by assumption). We do this in 4.2.1. Further, having both specification and implementation in linearized form, we can use the focus points method [19,18] to prove their equivalence. But not immediately, since this method requires that the implementation should be convergent (without infinite  $\tau$ -loops) and this is not the case for ours - infinite  $\tau$ -loops occur when abstracting from  $R$  and  $W$ . Therefore, in 4.2.2, we will consider an intermediate specification  $Y$ , in which we abstract only from  $R$ 's and the second  $W$  (the one generated by the communication between  $write(m + 1, d)$  and  $Write$  from  $GSRW$ , see 4), while keeping the other  $write(m + 1, g_i(d, e_i))$  as a visible action - but renamed to an action without arguments  $v$ . In  $Y$  we also eliminate the database  $A$ . Now we can prove, using the focus points method, that the linearized implementation is branching bisimilar to  $Y$ . Afterwards we abstract from the remaining visible action  $v$  and prove by *fair abstraction* (4.2.3) that  $\tau.\tau_{\{v\}}Y = \tau.Spec$ .

### 4.2.1 Linearization of implementation

In the linearized version of a process, we view everything globally. The state of the system will be described by the parameters  $A$ ,  $\mathbf{m} \in N^n$ ,  $\mathbf{l} \in \{0, 1, 2\}^n$  and  $\mathbf{d} \in D^n$ .  $A$  is the set of pairs, the database appearing as parameter of process  $GSRW$ .  $\mathbf{m}$  is the vector of “moments”, an element  $m_i$  (the parameter of process  $X_i$ ) is the moment when  $X_i$  will be activated next.  $\mathbf{l}$  is the vector of local states ( $l_i$  is the current local state of component  $i$ ). Finally,  $\mathbf{d}$  is the vector of data items;  $d_i$  is the data that component  $i$  knows of, currently. Although in principle there is only one global view on data, components may have temporary different views. That’s why we need  $\mathbf{d}$  as parameter, instead of just  $d$ .

In the initial state,  $A = \{(0, d)\}$  (we are at moment 0 and the current data is the global specification’s parameter  $d$ );  $\mathbf{l} = 0$  (all the components are in the “start” local state 0);  $\mathbf{m} = (0, 1, \dots, n - 1)$  (component  $i$  waits to be activated at moment  $i$  and first component to be activated is 0, triggered by  $(0, d)$ , the only pair from the database  $A$ );  $\mathbf{d} = \mathbf{0}$  (in the initial state the values in this vector don’t matter, since they will be used only after being initialized by a reading action).

Due to the fact that all components  $X_i$  from (5) are independent, the linearized version is just the sum of their separate interactions with  $GSRW(A)$ . After renaming one of the write actions to  $v$  and hiding the read action and

the other write, we get the following linearized implementation:

$$\begin{aligned}
 Impl(A, \mathbf{l}, \mathbf{m}, \mathbf{d}) = & \sum_{i=0}^{n-1} ( \\
 & \sum_y \tau. Impl(A, \mathbf{l}[l_i := 1], \mathbf{m}, \mathbf{d}[d_i := y]) \\
 & \quad \triangleleft l_i = 0 \wedge (m_i, y) \in A \triangleright \delta \\
 & + v. Impl(A \cup \{(m_i + 1, d)\}, \mathbf{l}[l_i := 0], \mathbf{m}[m_i := m_i + n], \mathbf{d}) \\
 & \quad \triangleleft l_i = 1 \triangleright \delta \\
 & + \sum_{e_i: E_i} (a_i(f_i(d, e_i)). Impl(A, \mathbf{l}[l_i = 2], \mathbf{m}, \mathbf{d}[d_i := g_i(d, e_i)]) \\
 & \quad \triangleleft l_i = 1 \wedge b_i(d, e_i) \triangleright \delta) \\
 & + \tau. Impl(A \cup \{(m_i + 1, d_i)\}, \mathbf{l}[l_i := 0], \mathbf{m}[m_i := m_i + n], \mathbf{d}) \\
 & \quad \triangleleft l_i = 2 \triangleright \delta)
 \end{aligned} \tag{6}$$

The formula

$$\begin{aligned}
 \tau_{\{R, W\}} \partial_{\{Read, Write, read, write\}} ( \|_i X_i(i) \| GSRW(\{(0, d)\}) ) \\
 = \tau_{\{v\}} Impl(\{(0, d)\}, \mathbf{0}, (0, 1, \dots, n-1), \mathbf{0}) \tag{7}
 \end{aligned}$$

summarizes what has happened in the linearization step.

#### 4.2.2 Pre-abstraction

We define the intermediate specification  $Y$  as follows:

$$\begin{aligned}
 Y(\mathbf{c}, \mathbf{d}) = & \sum_{i=0}^{n-1} ( \\
 & v. Y((i+1) \bmod n, \mathbf{d}) \quad \triangleleft i = \mathbf{c} \triangleright \delta \\
 & + \sum_{e_i: E_i} (a_i(f_i(d, e_i)). Y((i+1) \bmod n, g_i(d, e_i)) \\
 & \quad \triangleleft i = \mathbf{c} \wedge b_i(\mathbf{d}, e_i) \triangleright \delta) ) \tag{8}
 \end{aligned}$$

The parameter  $\mathbf{c}$  is a natural number from the set  $\{0, \dots, n-1\}$  and points to the active component  $X_{\mathbf{c}}(m_{\mathbf{c}})$ .  $\mathbf{c}$ 's values in the successive calls of  $Y$  ( $Y(0, -), Y(1, -), Y(2, -), \dots, Y(n-1, -), Y(0, -), Y(1, -), \dots$ ) reflect the order in which components become active. The other parameter,  $\mathbf{d}$ , is the global state of the system.

We aim to show, by using an appropriate state mapping, that this intermediate specification is equivalent to the linearized implementation, i.e. that

$$\tau. Impl(\{(0, d)\}, \mathbf{0}, (0, 1, \dots, n-1), \mathbf{0}) = \tau. Y(0, d). \tag{9}$$

The state mapping must relate equivalent states of  $Impl$  and  $Y$ . To ensure this, the focus points method ([19,18], see section 2.2.2) requires that certain *matching criteria* should be satisfied, which are easy (but tedious) to prove, using invariants on  $Impl$ 's states. For the complete proof of (9), including a

list of the invariants, we refer the reader to [23]. Here we will only show some of the invariants and briefly discuss the state mapping.

One of the invariants is that for any “moment”  $t$  there is at most one data item  $d$  such that  $(t, d) \in A$ . When this item exists, we will denote it by  $\mathbf{data}(A, t)$ . Another important invariant is that for any state  $\langle A, \mathbf{l}, \mathbf{m}, \mathbf{d} \rangle$  there is exactly one  $x \in \{0 \cdots n - 1\}$  for which  $(m_x, -) \in A$  (where  $-$  denotes any data instance). The state mapping  $h : \text{States}(Impl) \rightarrow \text{States}(Y)$  can be now defined as follows:

$$h(\langle A, \mathbf{l}, \mathbf{m}, \mathbf{d} \rangle) = \begin{cases} \langle x, \mathbf{data}(A, m_x) \rangle & \text{if } l_x \in \{0, 1\} \text{ and } (m_x, -) \in A \\ \langle (x + 1) \bmod n, d_x \rangle & \text{if } l_x = 2 \text{ and } (m_x, -) \in A \end{cases}$$

The idea of this mapping is that it extracts from a global state  $\langle A, \mathbf{l}, \mathbf{m}, \mathbf{d} \rangle$  the essential information that characterize it, namely the index of the active component and the data that this component gets as input.

If we hide  $v$  in both  $Impl$  and  $Y$ , (9) becomes

$$\tau.\tau_{\{v\}}Impl(\{(0, d)\}, \mathbf{0}, (0, 1, \dots, n - 1), \mathbf{0}) = \tau.\tau_{\{v\}}Y(0, d). \quad (10)$$

#### 4.2.3 Abstraction

By instantiating the definition (8) for  $\mathbf{c} \in \{0 \cdots n - 1\}$  and using the observation that there are no summands for which  $i \neq \mathbf{c}$ , we obtain:

$$\begin{aligned} Y(0, d) &= v.Y(1, d) + \sum_{e_0: E_0} a_0(f_0(d, e_0)). Y(1, g_0(d, e_0)) \triangleleft b_0(d, e_0) \triangleright \delta \\ Y(1, d) &= v.Y(2, d) + \sum_{e_1: E_1} a_1(f_1(d, e_1)). Y(2, g_1(d, e_1)) \triangleleft b_1(d, e_1) \triangleright \delta \\ &\vdots \\ Y(n - 1, d) &= v.Y(0, d) + \sum_{e_{n-1}: E_{n-1}} a_{n-1}(f_{n-1}(d, e_{n-1})). Y(0, g_{n-1}(d, e_{n-1})) \\ &\quad \triangleleft b_{n-1}(d, e_{n-1}) \triangleright \delta \end{aligned}$$

It is easy to see that  $Y(0, d) \cdots Y(n - 1, d)$  form a  $\{v\}$ -cluster, with exits

$$\{a_i(f_i(d, e_i)). Y((i + 1) \bmod n, g_i(d, e_i)) \triangleleft b_i(d, e_i) \triangleright \delta \mid 0 \leq i < n, e_i \in E_i\}$$

KFAR<sub>n</sub> [1] (Koomen’s Fair Abstraction Rule) states that in a fair execution, one of the exits will eventually be taken. In our case, this means that we can write, for all  $k \in \{0 \cdots n - 1\}$ :

$$\begin{aligned} \tau.\tau_{\{v\}}Y(k, d) &= \\ \tau. \sum_{i=0}^{n-1} \sum_{e_i: E_i} a_i(f_i(d, e_i)). \tau_{\{v\}}Y((i + 1) \bmod n, g_i(d, e_i)) &\quad (11) \\ \triangleleft b_i(d, e_i) \triangleright \delta & \end{aligned}$$

The right-hand side of this formula does not depend on  $k$ , which allows us to say that  $\tau.\tau_{\{v\}}Y(0, d) = \tau.\tau_{\{v\}}Y(1, d) = \dots = \tau.\tau_{\{v\}}Y(n-1, d)$ . Consequently, we can replace in (11)  $k$  with 0 and

$a_i(f_i(d, e_i)).\tau_{\{v\}}Y((i+1) \bmod n, g_i(d, e_i))$  with  $a_i(f_i(d, e_i)).\tau_{\{v\}}Y(0, g_i(d, e_i))$  and obtain:

$$\tau.\tau_{\{v\}}Y(0, d) = \tau. \sum_{i=0}^{n-1} \sum_{e_i \in E_i} a_i(f_i(d, e_i)).\tau_{\{v\}}Y(0, g_i(d, e_i)) \triangleleft b_i(d, e_i) \triangleright \delta$$

Comparing with (3), we see that  $\tau.\tau_{\{v\}}Y(0, d)$  and  $\tau.Spec(d)$  are solutions of the same equation, thus, by *RSP*, they are equal. This equality, together with the linearization (7) and the equivalence to the intermediate specification (10), proves the theorem 4.2.

## 5 Conclusion

We have studied the architecture *GSRW*, based on write and blocking, non-destructive read primitives on a global set. By viewing the architecture as a separate component defined by process algebra, we obtained a nice separation between the tasks of application programming on the architecture, and the distributed implementation of the architecture itself.

*GSRW* provides a conceptual global view to application programmers, making the development and analysis of applications easier. Our first result shows that maintaining the global view doesn't lead to any overhead in the distributed implementation, like locking protocols. For this, the limited set of coordination primitives is essential. Due to these restrictions, application processes just cannot observe that their local set is not (yet) up-to-date. Our second result supports this architecture, by indicating that despite these restrictions, the architecture is sufficiently expressive from a functional point of view.

Non-functional requirements, like performance and fault tolerance might lead to stronger coordination primitives, such as destructive or non-blocking read, as in Linda [11]. However, these don't come for free. Either, we have to give up the global view, as shown in [7,8], or complicated protocols are needed in order to guarantee global consistency, as the two-phase-commit protocol in *JavaSpaces<sup>tm</sup>* [15]. The former compromises ease of application program construction and analysis, the latter might comprise performance on a different level.

Future work could be directed to investigating other distribution schemes, based on different criteria. We could look for "efficient" implementations – for instance, schemes that would minimize the number of communication steps (i.e., interactions with the database). To this end, it might be necessary to add new primitives to the current *GSRW* model or to consider weaker equivalences between specification and implementation.

### 5.1 Related Work

In [13] a more detailed description of Splice is given, at the level of agents communicating on an Ethernet network. However, an abstract specification of this fragment is not given. Instead the model is validated by verifying that a number of scenarios satisfy certain desired temporal logic properties.

The distributed implementation that we give is at the same level of abstraction as in [5,7,8]. This is sufficient to show that for read/write primitives a global set is equivalent to a number of local sets. In [7,8] operational semantics corresponding to these views are given, and it is proved that for each program these views yield behaviourally equivalent semantics. Several other variants were considered, based on e.g. multi-sets and stronger coordination primitives. A semantics of JavaSpaces along the same lines is defined in [10]. In [5] denotational semantics are given for distributed and local versions, and it is proved and formally checked by a proof checker, that both semantics yield the same *write*-traces and end up in the same data space.

Although our realizability result resembles this work, the setting is quite different. As we have the architecture as a separate component, we can prove that the global architecture and its distributed implementation are behaviourally equivalent. Therefore our result is language independent and immediately applies to the case where components may use recursion and internal choice. This combination has not been considered in [5,7,8]. The proof we give is simpler in our view, as it mainly consists of checking some simple matching criteria, which are generated by a standard application of the cones-and-foci method [19,18].

In [5] an imperative language is used with as primitive  $read(x, q); P$ , which is blocked until some value  $v$  satisfying query  $q$  exists which is then bound in  $P$  to  $x$ . We obtain the same effect by the process  $\sum_x (read(x).P \triangleleft q(x) \triangleright \delta)$ . Instead of the action of writing or reading, these authors regard the arrival in the database observable, which we have hidden by a  $\tau_{\{Send\}}$  in DSRW. It is interesting future research to see how their semantics can be formally connected with ours.

Our expressiveness result should be contrasted with the result of [9], where it is shown that additional primitives, like the test-for-absence, are needed to get Turing completeness. There, components are restricted to finite state machines, and the computation power entirely comes from the coordination primitives. We take a system's engineering view, by focusing on the question whether the read and write primitives are sufficiently expressive for solving the coordination between (probably infinite state) application programs. We also focus on the real task of the components: implement the system's external global behaviour.

Our construction has similarities with transformations in [21], where a requirements specification is split in parallel parts communicating via message passing, and [22], where an encoding of choice in the a-synchronous  $\pi$ -calculus

is provided. Both papers introduce internal loops to resolve external choices, similar to our translation. However, those papers are based on event-based coordination, whereas our approach uses a persistent data approach. For this reason, we had to use increasing sequence numbers, and couldn't find a finite state solution.

## References

- [1] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge University Press, 1990.
- [2] J.A. Bergstra and J.W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37(1):77–121, 1985.
- [3] M.A. Bezem and J.F. Groote. Invariants in process algebra with data. In B. Jonsson and J. Parrow, editors, *Proceedings Concur'94*, number 836 in LNCS, pages 401–416, Uppsala, Sweden, 1994. Springer-Verlag.
- [4] S.C.C. Blom, W.J. Fokkink, J.F. Groote, I.A. Langevelde, B. Lissner, and J.C. van de Pol.  $\mu$ CRL: a toolset for analysing algebraic specifications. In *Proc. of CAV 2001*, number 2102 in LNCS, pages 250–254, 2001.
- [5] R. Bloo, J.J.M. Hooman, and E. de Jong. Semantical aspects of an architecture for distributed embedded systems. In *Proceedings of the 2000 ACM Symposium on Applied Computing (SAC 2000)*, pages 149–155. ACM, 2000.
- [6] M. Boasson. Control systems software. *IEEE Transactions on Automatic Control*, 38(7):1094–1106, July 1993.
- [7] M.M. Bonsangue, J.N. Kok, and G. Zavattaro. Comparing coordination models based on shared distributed replicated data. In J. Carroll, H. Haddad, D. Oppenheim, B. Bryant, and G.B. Lamont, editors, *Proceedings of the 1999 ACM Symposium on Applied Computing (SAC '99)*, pages 146 – 155, San Antonio, Texas, USA, February 1999. ACM press.
- [8] M.M. Bonsangue, J.N. Kok, and G. Zavattaro. Comparing software architectures for coordination languages. In P. Ciancarini and A. Wolf, editors, *Proceedings of the 3rd International Conference on Coordination Languages and Models (Coordination 99)*, number 1594 in LNCS, pages 150–164. Springer-Verlag, 1999.
- [9] N. Busi, R. Gorrieri, and G. Zavattaro. On the Turing equivalence of Linda coordination primitives. In *Proceedings of Express '97*, volume 7 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1997.
- [10] N. Busi, R. Gorrieri, and G. Zavattaro. Process calculi for coordination: From Linda to JavaSpaces. In T. Rus, editor, *8th International Conference on Algebraic Methodology and Software Technology*, number 1816 in LNCS, Iowa, USA, 2000. Springer-Verlag.

- [11] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.
- [12] P.F.G. Dechering and E. de Jong. Transparent object replication: A formal model. In *Fifth Workshop on Object-oriented Real-Time Dependable Systems (WORDS'99F)*, Monterey, California, USA, 2000. IEEE Computer Society.
- [13] P.F.G. Dechering and I.A. van Langevelde. The verification of coordination. In A. Porto and C. Roman, editors, *Proceedings of the Fourth International Conference on Coordination Models and Languages*, number 1906 in LNCS, Limassol, Cyprus, 2000. Springer-Verlag.
- [14] W. Fokkink. *Introduction to Process Algebra*. Texts in Theoretical Computer Science (EATCS). Springer-Verlag, 2000.
- [15] E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces principles, patterns, and practice*. Addison-Wesley, Reading, MA, USA, 1999.
- [16] R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, May 1996.
- [17] J.F. Groote, A. Ponse, and Y.S. Usenko. Linearization in parallel pCRL. *Journal of Logic and Algebraic Programming*, 48(1-2):39–72, June 2001.
- [18] J.F. Groote and M.A. Reniers. Algebraic process verification. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, pages 1151–1208. North-Holland, 2001.
- [19] J.F. Groote and J.S. Springintveld. Focus points and convergent process operators: a proof strategy for protocol verification. *Journal of Logic and Algebraic Programming*, 49(1-2):31–60, 2001.
- [20] J.M.M. Hooman and J.C. van de Pol. Formal verification of replication on a distributed data space architecture. In *Proceedings of SAC 2002 (Madrid)*, pages 351–358. ACM, 2002.
- [21] R. Langerak. *Transformations and Semantics for LOTOS*. PhD thesis, Department of Computer Science, University of Twente, November 1992.
- [22] U. Nestmann and B.C. Pierce. Decoding choice encodings. In U. Montanari and V. Sassone, editors, *Proc. of the 7th Int. Conf. on Concurrency Theory (CONCUR 96)*, number 1119 in LNCS, pages 179–194. Springer-Verlag, 1996.
- [23] S.M. Orzan. Distributing requirements specifications on Basic Splice. Technical Report SEN-R0101, CWI, Amsterdam, The Netherlands, 2001. <http://db.cwi.nl/rapporten/index.php>.
- [24] J.C. van de Pol. Expressiveness of Basic Splice. Technical Report SEN-R0033, CWI, Amsterdam, The Netherlands, 2000. <http://db.cwi.nl/rapporten/index.php>.
- [25] J.C. van de Pol and M. Valero Espada. Formal specification of JavaSpaces<sup>TM</sup> architecture using  $\mu\text{crl}$ . In F. Arbab and C. Talcott, editors, *Proc. of COORDINATION*, number 2315 in LNCS, pages 274–290. Springer, 2002.