View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by CWI's Institutional Repository

Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

Distributing Requirements Specifications on Basic Splice

S.M. Orzan

Software Engineering (SEN)

SEN-R0101 January 31, 2001

Report SEN-R0101
ISSN 1386-369X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

CWI is the National Research Institute for Mathematics and Computer Science. CWI is part of the Stichting Mathematisch Centrum (SMC), the Dutch foundation for promotion of mathematics and computer science and their applications.
SMC is sponsored by the Netherlands Organization for Scientific Research (NWO). CWI is a member of ERCIM, the European Research Consortium for Informatics and Mathematics.

Copyright © Stichting Mathematisch Centrum
P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

Distributing Requirements Specifications on Basic Splice

Simona Orzan
Centrum voor Wiskunde en Informatica
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
Email: Simona.Orzan@cwi.nl

ABSTRACT
This is an extension of work presented in [12]. It is proved that the seemingly weak architecture Basic Splice introduced there –in which the coordination of processes is done using only a global set with read/write primitives– can support a distributed implementation of a large class of requirements specifications, namely LPEs (a μCRL intermediate representation of specifications).

2000 Mathematics Subject Classification: 68M14, 68N30, 68Q85
Keywords and Phrases: Splice, shared data space, software architecture, distributed systems, expressiveness
Note: Research carried out for the STW-project CES.5009: Formal Design, Tooling, and Prototype Implementation of a Real-Time Distributed Shared Dataspace.

1. Introduction

Shared dataspaces are a widely used mechanism for specifying the communication and synchronization of parallel processes. The common dataspace is accessed through a certain set of operations(primitives) like adding, deleting, test for absence/presence of data etc.; different coordination models can be obtained by considering different sets of primitives. A natural question is how expressive these models are. This can be addressed in two ways: one can think of "relative" expressiveness, i.e. wonder which models can express in an easier way the same requirements —a few classes of coordination models were compared from this point of view in [5]; alternatively, one can envision "absolute" expressiveness, i.e. wonder which exactly are the requirements specifications that can be implemented on a certain model.

We consider a very simple shared dataspace model, with only the primitives *write* and *nondestructive blocking read*. It was introduced in [12] and was called Basic Splice because it is the ultimate simplification of the data-oriented software architecture for complex control systems SPLICE [2] (Subscription Paradigm for the Logical Interconnection of Concurrent Engines), developed at Hollandse Signaalapparaten. The main characteristic of this architecture is that the processes can not exchange messages directly, they only communicate through the shared data. This gives processes a high degree of independence and insures a certain flexibility for the whole system; i.e., easy adaptation to changes in the environment, easy dynamic reconfiguration, suitable basis for fault-tolerance techniques. The idea of viewing Splice as a shared dataspace was proposed in [1, 6]. Alternative Splice descriptions are discussed in [3, 4].

The role of an architecture in the design of a complex distributed system is to describe how application processes should coordinate. One of the problems that we encounter after we have the architecture is how to distributedly implement on it the requirements of the system under design. That is, how to transform the requirements into a set of application processes that communicate using the architecture's rules. We address this problem by studying the "absolute" expressiveness of the seemingly very weak shared dataspace model Basic Splice.

Following [12], we view Basic Splice as a process which is running in parallel with the application processes coordinated by it. All processes are specified in the language μCRL; we work with a special format called Linear Process Equation(LPE), for which powerful techniques(focus and cones method, fair abstraction rules) have been developed [10].

We give a mapping from the requirements specification expressed as an LPE to a set of processes (also written as LPEs) that, when running in parallel and coordinated by Basic Splice, show the same behavior as the initial specification. The equivalence used is branching bisimilarity [8]. In this way we prove that it is possible to distributedly implement on Basic Splice a class of processes as large as LPEs. That is, almost all requirements specifications, since it was proved [9] that a very large part of μCRL specifications can be expressed as LPEs.

1.1 LPEs

We will briefly present μCRL (micro Common Representation Language) , which is a language for specifying interacting processes that rely on data. Its signature includes datasorts (like $Bool$, Nat) and a distinct sort $Proc$ of processes. The notation $\vec{d}$ will be used for tuples of data variables. We have a set of action labels Act and $\{a(\vec{d})|a \in Act\}$ are actions parameterized by data terms.

Process terms are constructed using the operators $+$ (choice), . (sequential composition), $\|$ (parallel composition), $|$ (communication merge), $\sum$ and $\lhd\ \rhd$. The operator $\sum$ is the *generalized choice.* $\sum_{d:Nat} P(d)$, for instance, denotes the possibly infinite sum $P(0)+P(1)+P(2)\cdots$. The *conditional* $P \lhd b \rhd Q$, with P and Q being processes and b a boolean, means "if b then P else Q". Constant δ of the sort $Proc$ represents *deadlock* and is neutral element for $+$. Constant τ of the sort $Proc$ represents *the silent step*(not observable). The *encapsulation operator* $\partial_H(P)$ enforces actions in H to communicate; within process P they can not execute alone, but only as part of a communication step. The *hiding operator* $\tau_I(P)$ renames in P all the actions from I to τ.

In the following, I is a finite set of indices, D and E_i are datasorts (not necessarily finite), d and e_i are data variables of sorts D and E_i, f_i, g_i and b_i are functions, $f_i, g_i : D \times E_i \longrightarrow D$, $b_i : D \times E_i \longrightarrow Bool$. An LPE is a recursive specification of the form:

$$Spec\,(d:D) = \sum_{i\in I}\sum_{e_i:E_i} a_i(f_i(d,e_i)).Spec\,(g_i(d,e_i)) \quad \lhd b_i(d,e_i) \rhd \quad \delta \tag{1.1}$$

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.

1.2 Basic Splice

In our setting, Basic Splice is viewed as a special process $\mathsf{SPLICE}(A)$, running in parallel with the application processes. A is the shared dataspace, the global set in which values of some sort $\mathcal{D}$ are added. This set can only grow, as the *read* primitive in our model is nondestructive.

Application processes should be allowed to communicate only through the shared dataspace, never directly. This is done by making them use the special actions *write* and *read* and introducing the co-actions $Write$ and $Read$ in the dataspace process $\mathsf{SPLICE}(A)$. The communications $read(x)|Read(x)$ and $write(x)|Write(x)$ are considered actions themselves: $R(x)$ and $W(x)$.

The architecture is described by the following recursive specification:

$$\mathsf{SPLICE}(A : Set(\mathcal{D})) = \sum_{x:\mathcal{D}} Write(x).\mathsf{SPLICE}(A \cup \{x\}) \;+\; \sum_{x:\mathcal{D}} Read(x).\mathsf{SPLICE}(A) \lhd x \in A \rhd \delta$$

When a communication $W(x)$ occurs, $\mathsf{SPLICE}(A \cup \{x\})$ proceeds; that is, the value x is added to the global set - unless this value was already present, in which case nothing happens. The blocking behavior of the *read* primitive is nicely expressed by the condition $\lhd x \in A \rhd$. The communication $R(x)$ can only happen if x is present in the global set (otherwise $Read(x)$ doesn't exist among the actions of $\mathsf{SPLICE}(A)$). So, if an application process wants to do $read(3)$, it has to wait until 3 comes in the database; this will add $Read(3)$ to the sum that represents S, which will allow the communication. Note that after this communication S proceeds with an unchanged database, since *read* doesn't delete the read value.

This view of the architecture as a special component and communication done by the *read*/*Read* and *write*/*Write* pairs provide an elegant formal semantics to the two primitives of the architecture.

Example 1 To make clear how μCRL and Basic Splice function, we describe an implementation on Basic Splice of a very simple buffer specification.

For this, we consider the datasort Queue, to represent a queue of natural numbers. It has the constant $\mathsf{emptyqueue}$ and the following operations: push: $Nat \times \mathsf{Queue} \longrightarrow \mathsf{Queue}$, which adds an element to a queue; pop:$\mathsf{Queue} \longrightarrow \mathsf{Queue}$, which extracts the top element of the (not empty) parameter queue; top:$\mathsf{Queue} \longrightarrow Nat$, which returns the top element of the given queue; and $\mathsf{notempty}$:$\mathsf{Queue} \longrightarrow Bool$, which adds an element to a queue. And we define the following μCRL specification of the buffer:

$$BufSpec\,(\mathsf{Q}\colon \mathsf{Queue}) = \sum_{d:Nat} SCAN(d).BufSpec(\mathsf{push}(d,\mathsf{Q})) \; + \; SEND(\mathsf{top}(d)).BufSpec(\mathsf{pop}(\mathsf{Q})) \lhd \mathsf{notempty}(\mathsf{Q}) \rhd \delta \tag{1.2}$$

The visible actions are $SCAN$, which inputs a natural number to the buffer and $SEND$, which outputs a natural number from the buffer. Data must be sent out in the same order in which it was scanned.

We need that the global set memorizes values of sort $Nat \times Nat$ – a value represents (sequence number, data item). So, our Splice process will be $\mathsf{SPLICE}(A : Set(Nat \times Nat))$. Then the implementation on Splice of the buffer can be:

$$BufImpl = B_{scan}\,(0) \,\|\, B_{send}\,(0) \,\|\, S\,(\emptyset)$$

where

$$B_{scan}\,(n : Nat) = \sum_{d:Nat} SCAN(d).write(n,d).B_{scan}\,(n+1)$$

$$B_{send}\,(n : Nat) = \sum_{d:Nat} read(n,d).SEND(d).B_{send}\,(n+1)$$

(1) is indeed the implementation of (1.2), since it can be proved that

$$BufSpec\,(\mathsf{emptyqueue}) = \tau_{\{R,W\}} \partial_{\{Read,Write,read,write\}}\, BufImpl$$

(branching bisimilar)

1.3 The problem

We are looking for a *distributed implementation* of LPEs on the architecture Basic Splice. Given a requirements specification $Spec(d)$, we want to build a set of processes $P_1, \cdots, P_n$ and some initial database A_0 such that $P_1 \| \cdots \| P_n \| \mathsf{SPLICE}(A_0)$, after hiding *read*, *write*, *Read* and *Write*, equals $Spec(d)$ (in the sense of branching bisimilarity).

There can be imagined many sets of constraints to impose on processes P_i. Studying different schemes from the efficiency point of view is interesting and is subject of further work. Now the goal is only to prove that distribution is possible, so we consider a very simple constraint: each P_i should "produce" only one action from those in $Spec(d)$. If, for instance, the specification requirements is $a.b$, the distributed implementation should have a component for a and another for b.

The matter of distributing functionalities of a requirements specification over more communicating components was also studied in [11] for LOTOS expressions; the synchronization is solved there with message passing, while Basic Splice coordinates the components using persistent data.

Section 2 describes the translation scheme and section 3 proves it correct.

2. The translation scheme

Consider a specification in LPE format $Spec(d)$ (1.1), in with $|I| = n$. The corresponding distributed process will have n components, each responsible for one action a_i. They communicate via SPLICE, through a global set of *pairs* $(Nat \times D)$. An element (t, d) of the global set represents:
t (a natural number) = *timestamp* - this is the moment when the pair was added to the database; it is also the number of visible + invisible steps executed until the time of insertion
d = the current *data*; the global state of the system, according to the specification

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. In both cases, it will increase the "global time" and pass the turn to its next sister. This cycle is needed to insure 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:

$$\begin{aligned} X_i(m) = \sum_{d:D} & read(m, d). \\ & \Big(\sum_{e_i:E_i} \big(a_i(f_i(d, e_i)).write(m + 1, g_i(d, e_i)) \lhd b_i(d, e_i) \rhd \delta \big) + write(m + 1, d) \Big) . X_i(m + n) \end{aligned} \tag{2.1}$$

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, $read(m, d)$ from X_i synchronizes with $Read(m, d)$ from $\mathsf{SPLICE}(A)$, 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. $m + n$.

The initial state of the implementation is

$$\|_i X_i(i) \parallel \mathsf{SPLICE}(\{(0, d)\}) \tag{2.2}$$

We will prove that this distributed implementation on Basic Splice of a LPE is almost equivalent to the specification. That is: if we hide in the implementation the actions dealing with the global set ($read$,$write$,$Read$,$Write$) and we abstract from the communication actions (R,W), then we get (approximately) the specification $Spec(d)$.

Theorem 1 *For every requirements specification expressible as a LPE $Spec(d)$, the components X_i resulted by applying the translation scheme satisfy:*

$$\tau.Spec(d) = \tau.\tau_{\{R,W\}}\partial_{\{Read,Write,read,write\}}(\|_i X_i(i) \| \mathsf{SPLICE}(\{(0, d)\})).$$

Example 2 The regular process $X = (a + a.b + b.a).\delta$ can be written as this LPE:

$$\begin{aligned} X(d : \{0, 1, 2, 3\}) = {} & a.X(3) \lhd d = 0 \vee d = 2 \rhd \delta \\ & + a.X(1) \lhd d = 0 \rhd \delta \\ & + b.X(2) \lhd d = 0 \rhd \delta \\ & + b.X(3) \lhd d = 1 \rhd \delta \end{aligned}$$

By applying the translation scheme, we get the following distribution:

$$
\begin{aligned}
X_0(m) &= \sum_{d:\{0,1,2,3\}} read(m,d).(a.write(m+1,3) \triangleleft d=0 \vee d=2 \triangleright \delta + write(m+1,d)).X_0(m+4)\\
X_1(m) &= \sum_{d:\{0,1,2,3\}} read(m,d).(a.write(m+1,1) \triangleleft d=0 \triangleright \delta + write(m+1,d)).X_1(m+4)\\
X_2(m) &= \sum_{d:\{0,1,2,3\}} read(m,d).(b.write(m+1,2) \triangleleft d=0 \triangleright + write(m+1,d)).X_2(m+4)\\
X_3(m) &= \sum_{d:\{0,1,2,3\}} read(m,d).(b.write(m+1,5) \triangleleft d=1 \triangleright \delta + write(m+1,d)).X_3(m+4)
\end{aligned}
$$

And according to theorem 1:

$$\tau.X = \tau.\tau_{\{R,W\}}\partial_{\{Read,Write,read,write\}}(\mathsf{SPLICE}(\{(0,0)\}) \parallel X_0(0) \parallel X_1(1) \parallel X_2(2) \parallel X_3(3))$$

Example 3 Alternatively, the same X can be written as another LPE (clustered LPE), branching bisimilar to the first one:

$$
\begin{aligned}
&X(d:\{0,1,2,3\}) = \sum_{e:Bool} a.X(\text{“if } (d=0 \wedge e) \text{ then 1 else 3}'') \triangleleft d=0 \vee d=2 \triangleright \delta\\
&\quad + b.X(\text{“if d=0 then 2 else 3}'') \triangleleft d=0 \vee d=1 \triangleright \delta
\end{aligned}
$$

Then the distributed version would be:

$$
\begin{aligned}
&X_0(m) = \sum_{d:\{0,1,2,3\}} \sum_{e:Bool} read(m,d).\\
&\quad (a.write(m+1, \text{“if } (d=0 \wedge e) \text{ then 1 else 3”}) \triangleleft d=0 \vee d=2 \triangleright \delta + write(m+1,d)).X_0(m+2)
\end{aligned}
$$

$$
\begin{aligned}
&X_1(m) = \sum_{d:\{0,1,2,3\}} read(m,d).\\
&\quad (b.write(m+1, \text{“if d=0 then 2 else 3”}) \triangleleft d=0 \triangleright \delta + write(m+1,d)).X_1(m+2)
\end{aligned}
$$

And again, by theorem 1,

$$\tau.X = \tau.\tau_{\{R,W\}}\partial_{\{Read,Write,read,write\}}(\mathsf{SPLICE}(\{(0,0)\}) \parallel X_0(0) \parallel X_1(1))$$

We showed two ways of distributing on SPLICE the process $(a + a.b + b.a).\delta$. Note how the “degree of distribution” can be changed by clustering the actions, as opposite to spreading them over more summands. In the first example, different a actions were placed on different summands of the LPE; in the second, all a-actions were grouped on the same summand.

3. Correctness proof

This chapter is devoted to thoroughly prove that the translation defined above is correct. That is, to prove theorem 1.

First of all, to be able to compare the two processes appearing in the theorem, we need them in a linearized form. The specification $Spec(d)$ is an LPE already by definition; it's left to linearize the implementation (2.2). In [9] it was shown that this is possible.

3.1 Implementation linearized

Each component X_i from the implementation (definition 2.2) passes in its life only through the following locations: 0 –ready to read, 1 –activated; make a choice (execute action or pass the turn), 2 –action performed; pass the turn. A "life" example: $0 \xrightarrow{R} 1 \xrightarrow{W} 0 \xrightarrow{R} 1 \xrightarrow{W} 0 \xrightarrow{R} 1 \xrightarrow{a_i(d,e_i)} 2 \xrightarrow{W} 0 \xrightarrow{R} 1$ etc.

In the linearized version of the implementation, we view everything globally. The state of the system as a whole will be described by the following parameters:

- A – the set of pairs (the database), the parameter of process S.
- $\vec{m} \in N^n$ – the vector of "moments"; m_i = the parameter of process X_i, the moment when X_i will be activated next.
- $\vec{l} \in \{0,1,2\}^n$ – the vector of locations (l_i is the current location of component i); as explained above, a location can be only 0,1,2
- $\vec{d} \in D^n$ – 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 $\vec{d}$ as parameter, instead of just d.

For the initial state,

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

The fact that all components X_i from (2.2) are independent allows us to obtain the linearized version of (2.2) by just summing their separate behaviors in the interaction with SPLICE(A):

$$
\begin{aligned}
&\mathit{Impl}(A,\vec{l},\vec{m},\vec{d}) = \sum_{i=0}^{n-1}(\\
&\quad \sum_{y} R(m_i,y).\,\mathit{Impl}(A,\vec{l}[\,l_i := 1\,],\vec{m},\vec{d}[\,d_i := y\,])\\
&\qquad\qquad \triangleleft\ l_i = 0 \ \wedge\ (m_i,y) \in A\ \triangleright\ \delta\\
&\quad + W(m_i+1,d_i).\,\mathit{Impl}(\,A \cup \{(m_i+1,d)\},\vec{l}[\,l_i := 0\,],\vec{m}[\,m_i := m_i+n\,],\vec{d})\\
&\qquad\qquad \triangleleft\, l_i = 1 \triangleright \delta\\
&\quad + \sum_{e_i:E_i} (a_i(f_i(d_i,e_i)).\,\mathit{Impl}(A,\vec{l}[\,l_i := 2\,],\vec{m},\vec{d})\\
&\qquad\qquad \triangleleft\, l_i = 1 \wedge b_i(d_i,e_i) \triangleright \delta)\\
&\quad + W(m_i+1,d_i).\,\mathit{Impl}(A \cup \{(m_i+1,g_i(d_i,e_i))\},\vec{l}[\,l_i := 0\,],\vec{m}[\,m_i := m_i+n\,],\vec{d})\\
&\qquad\qquad \triangleleft\, l_i = 2 \triangleright \delta)
\end{aligned}
\tag{3.1}
$$

The use of focus points method [10] to prove the equivalence between a specification and an implementation requires that the specification shouldn't contain τ-steps and that the implementation should be convergent (without infinite τ-loops). This means that the equivalence of our specification and implementation cannot be immediately proved because of the infinite τ-loops that appear in the implementation when we abstract from R and W.

Therefore we consider an intermediate specification Y, in which we abstract only from R's and the second W (call it W_2), while keeping the other W (W_1) as a visible action - but renamed to an action without arguments v, because in Y we also drop the database A.

When we distinguish between the two W, (3.1) becomes:

$$
\begin{aligned}
&Impl(A, \vec{l}, \vec{m}, \vec{d}) = \sum_{i=0}^{n-1} (\\
&\quad \sum_{y} R(m_i, y).\, Impl(A, \vec{l}[\, l_i := 1\,], \vec{m}, \vec{d}[\, d_i := y\,]) \\
&\qquad\qquad \lhd\ l_i = 0\ \wedge\ m_i, y) \in A\ \rhd\ \delta \\
&\quad + W_1(m_i + 1, d_i).\, Impl(\, A \cup \{(m_i + 1, d)\}, \vec{l}[\, l_i := 0\,], \vec{m}[\, m_i := m_i + n\,], \vec{d}) \\
&\qquad\qquad \lhd l_i = 1 \rhd \delta \\
&\quad + \sum_{e_i : E_i} (a_i(f_i(d_i, e_i)).\, Impl(A, \vec{l}[\, l_i := 2\,], \vec{m}, \vec{d}) \\
&\qquad\qquad \lhd l_i = 1 \wedge b_i(d_i, e_i) \rhd \delta) \\
&\quad + W_2(m_i + 1, d_i).\, Impl(A \cup \{(m_i + 1, g_i(d_i, e_i))\}, \vec{l}[\, l_i := 0\,], \vec{m}[\, m_i := m_i + n\,], \vec{d}) \\
&\qquad\qquad \lhd l_i = 2 \rhd \delta)
\end{aligned}
\tag{3.2}
$$

So, the claim of theorem 1 rewrites to:

$$\tau.Spec(d) = \tau.\tau_{\{R, W_1, W_2\}} \partial_{\{Read, Write, read, write\}} Impl(\{(0, d)\}, \vec{0}, (0, 1, \cdots, n-1), \vec{0})$$

Next, we will proceed in 2 steps:

- *pre-abstraction*: define the intermediate specification $Y(\mathbf{c}, \mathbf{d})$ and prove, using the focus points method, that

  $$\tau_{\{R, W_2\}} \partial_{\{Read, Write, read, write\}} Impl(\{(0, d)\}, \vec{0}, (0, 1, \cdots, n-1), \vec{0})[W_1 \longrightarrow v] = Y(0, d)$$

  (sections 3.2, 3.3, 3.4)

- *abstraction*: prove by fair abstraction that

  $$\tau.\tau_{\{v\}} Y(0, d) = \tau.Spec(d)$$

  (section 3.5)

3.2 Pre-abstraction

We define the intermediate specification Y as follows:

$$
\begin{aligned}
&Y(\mathbf{c},\mathbf{d}) = \sum_{i=0}^{n-1}(\\
&\quad v.\,Y((i+1) \bmod n, \mathbf{d}) \\
&\qquad\qquad \lhd\, i = \mathbf{c} \,\rhd\, \delta \\
&\quad + \sum_{e_i:E_i} (a_i(f_i(d,e_i)).\,Y((i+1) \bmod n, g_i(d,e_i)) \\
&\qquad\qquad \lhd\, i = \mathbf{c} \wedge b_i(\mathbf{d},e_i) \,\rhd\, \delta)\ )
\end{aligned}
\tag{3.3}
$$

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

We aim to show, by using an appropriate state mapping, that this process is equivalent to the process $\tau_{\{R,W_2\}}Impl[W_1 \longrightarrow v]$, which we will call $X(A,\vec{l},\vec{m},\vec{d})$:

$$
\begin{aligned}
&X(A,\vec{l},\vec{m},\vec{d}) = \sum_{i=0}^{n-1}(\\
&\quad \sum_{y} \tau.\,X(A,\,\vec{l}[l_i := 1],\vec{m},\vec{d}[d_i := y]\,) \\
&\qquad\qquad \lhd\ l_i = 0 \wedge (m_i,y) \in A\ \rhd\ \delta \\
&\quad + v.\,X(\,A \cup \{(m_i+1,d)\},\,\vec{l}[l_i := 0],\,\vec{m}[m_i := m_i+n],\vec{d}) \\
&\qquad\qquad \lhd l_i = 1 \rhd \delta \\
&\quad + \sum_{e_i:E_i} (a_i(f_i(d,e_i)).\,X(A,\vec{l}[\,l_i = 2\,],\vec{m},\vec{d}[d_i := g_i(d,e_i)]) \\
&\qquad\qquad \lhd l_i = 1 \wedge b_i(d,e_i) \rhd \delta) \\
&\quad + \tau.\,X(\,A \cup \{(m_i+1,d_i)\},\,\vec{l}[l_i := 0],\vec{m}[m_i := m_i+n],\vec{d}) \\
&\qquad\qquad \lhd l_i = 2 \rhd \delta)
\end{aligned}
$$

The state mapping must relate equivalent states of X and Y. To insure this, the focus points method [10] requires that certain *matching criteria* should be satisfied. In order to prove these, some properties that hold in all X's states will be necessary (*invariants*).

We continue with stating and proving a few invariants (section 3.3), then we will give the state mapping $h : States(X) \longrightarrow States(Y)$ (section 3.4) and check the required matching criteria.

3.3 Invariants

In the following, $\star$ denotes any instance of data from D.
We will need the function

$$\textbf{active} : States(X) \longrightarrow \{0 \cdots n-1\},\ \textbf{active}(\,\langle A,\vec{l},\vec{m},\vec{d}\rangle\,) = \textit{the } x \text{ for which} (m_x,\star) \in A$$

Lemma 1 *The following invariants hold for the implementation X:*

1. $(\forall t)$ *there is at most one pair* $(t, \star) \in A$
2. $\exists! i\,(0 \leq i < n)$ *s.t.* $(m_i, \star) \in A$ *(i.e.,* **active** *is well defined). And* $m_i = \max\limits_{(t,d)\in A} t$.
3. *if* $x = \mathbf{active}(\langle A, \vec{l}, \vec{m}, \vec{d}\rangle)$ *then*
   $\forall i \in \{0 \cdots n-1\}$ *s.t.* $i \neq (x-1)$ *mod* n, $m_{(i+1)\ mod\ n} = m_i + 1$ *and* $m_x = m_{(x-1)\ mod\ n} + 1 - n$.
   (in other words: in each state we have an arithmetic progression $m_x < m_{x+1} < \cdots m_n < m_1 < \cdots m_{x-1}$*, with step 1)*
4. $\vec{l} = \vec{0}$
   or ($\exists i (0 \leq i < n) : l_i = 1$ *and* $\forall j \neq i\ \ l_j = 0$
   or ($\exists i (0 \leq i < n) : l_i = 2$ *and* $\forall j \neq i\ \ l_j = 0$*).*
5. *if* $l_i > 0$ *then* $\mathbf{active}(\langle A, \vec{l}, \vec{m}, \vec{d}\rangle) = i$.

Proof .
We first prove that these invariants hold for X's initial state
$\langle A_0, \vec{l_0}, \vec{m}_0, \vec{d_0}\rangle = \langle\{(0,d)\}, \vec{0}, (0, 1, \cdots, n-1), \vec{0}\rangle$:

1. immediate.
2. i=0.
3. $\mathbf{active}(A_0, \vec{l_0}, \vec{m}_0, \vec{d_0}) = 0$.
   $(\forall i : 0 \leq i \leq n-2)\ {m_0}_{i+1} = {m_0}_i + 1 = ({m_0}_i + 1) \bmod n$.
   And ${m_0}_0 = 0 = (n-1) + 1 - n = {m_0}_{n-1} + 1 - n = {m_0}_{(-1) \bmod n} + 1 - n$.
4. $l_0 = \vec{0}$.
5. $l_0 = \vec{0}$.

Now, supposing that they hold for an arbitrary state $\langle A, \vec{l}, \vec{m}, \vec{d}\rangle$, we will prove that they're still true for X's next state $\langle A', \vec{l'}, \vec{m}', \vec{d'}\rangle$. We have to analyze the following possibilities (extracted from the description of process X):

- *for some* k, $l_k = 0 \wedge (m_k, \star) \in A$ *and a* τ*-step happens.*
  Then $\langle A', \vec{l'}, \vec{m}', \vec{d'}\rangle = \langle A,\ \vec{l}[l_k := 1], \vec{m}, \vec{d}[d_k := y]\rangle$.
  In this case, (1),(2),(3) remain true because state components that occur in these properties (database A and moments' array $\vec{m}$) haven't changed.
  $(m_k, \star) \in A \overset{inv.(2)}{\rightsquigarrow} \mathbf{active}(A, \vec{l}, \vec{m}, \vec{d}) = k \rightsquigarrow j \neq \mathbf{active}(A, \vec{l}, \vec{m}, \vec{d}), \forall j \neq k \overset{inv.(5)}{\rightsquigarrow} l_j = 0 \forall j \neq k$.
  But $l_k = 0$ too, so $\vec{l} = \vec{0}$, which means that $\vec{l'} = (0 \cdots 1 \cdots 0)$,i.e. (4) is true.
  The only index i for which $l'_i > 0$ is k ($l'_k = 1$). $k = \mathbf{active}(A', \vec{l'}, \vec{m}', \vec{d'})$ (because $A' = A$ and $\vec{m}' = \vec{m}$), so (5) holds too.
- *for some* k, $l_k = 1$ *and a* v *action happens.*
  Then $\langle A', \vec{l'}, \vec{m}', \vec{d'}\rangle = \langle A \cup \{(m_k + 1, d)\},\ \vec{l}[l_k := 0],\ \vec{m}[m_k := m_k + n], \vec{d}\rangle$
  1. We have to prove that the newly added pair $(m_k + 1, d)$ hasn't disturbed this property, i.e. there is no $(m_k + 1, \star)$ already in A. This is true, since $m_k + 1 > m_k \overset{inv.(2)}{=} \max\limits_{(t,d)\in A} t$.
  2. If $n > 1$, the unique i is $(k+1) \bmod n$, because $m'_{(k+1) \bmod n} = m_k + 1$ (inv. (3)) and $(m_k + 1, d) \in A'$. Uniqueness is given by inv. (1).
     If $n = 1$, the active process 0 remains active ($(m_0 + 1, d_0) \in A'$ and $m'_0 = m_0 + 1$).

3. $x' = \mathbf{active}(A', \vec{l'}, \vec{m}', \vec{d'}) = (k+1) \bmod n$. Notice that $(x'-1) \bmod n = k$.
For $i \in \{0 \cdots n-1\}, i \neq k, i \neq (k-1) \bmod n$, the property remains true, as $m' = m$ for the values involved.
It remains to be shown that $m'_k = m'_{(k-1) \bmod n} + 1$ and that $m'_{x'} = m'_k + 1 - n$.
$m'_k = m_k + n \overset{inv.(3)}{=} (m_{(k-1) \bmod n} + 1 - n) + n = m_{(k-1) \bmod n} + 1 = m'_{(k-1) \bmod n} + 1$
$m'_{x'} = m_{x'} \overset{inv.(3)}{=} m_k + 1 = (m'_k - n) + 1$.
4. $l'_i = l_i = 0, \forall i \neq k$ and $l'_k = 0$. So, $\vec{l'} = \vec{0}$.
5. $\vec{l'} = \vec{0}$, by invariant (4).

- *for some k and some $e_k \in E_k$, $l_k = 1 \wedge b_k(d_k, e_k)$ and an a_k action happens.*
Then $\langle A', \vec{l'}, \vec{m}', \vec{d'} \rangle = \langle A, \vec{l}[\,l_k = 2\,], \vec{m}, \vec{d}[d_k := g_k(d_k, e_k)] \rangle$.
(1),(2),(3) hold because the state components involved are not changed by this step.
$l_k = 1 \overset{inv.(4)}{\leadsto} l_i = 0 \ \forall i \neq k$. So, since and $l'_i = l_i = 0 \forall i \neq k$ and $l'_k = 2$, (4) holds in the current state too.
And, finally, (5) holds too, as the active index did not change (it's still k).

- *for some k, $l_k = 2$ and a τ-action happens.*
Then $\langle A', \vec{l'}, \vec{m}', \vec{d'} \rangle = \langle\, A \cup \{(m_k + 1, d_k)\}, \vec{l}[l_k := 0], \vec{m}[m_k := m_k + n], \vec{d} \rangle$.
All the invariants are shown to hold by a reasoning similar to the second case ("for some k, $l_k = 1$ and a v action happens").

We proved (invariant 1) 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 $\mathbf{data}(A, t)$.

Notice that $\mathbf{data}(A, m_{\mathbf{active}(\,\langle A, \vec{l}, \vec{m}, \vec{d} \rangle\,)})$ is defined for all reachable states in $States(X)$.

Lemma 2 *If $l_i = 1$ then $\mathbf{data}(A, m_i) = d_i$.*

Proof . If $l_i = 1$ then the most recent step in X was a τ-step (a read from the database). This could happen only if the guard was true: $l_i = 0 \wedge (m_i, y) \in A$, for some y; by definition, $y = \mathbf{data}(A, m_i)$. The changes that occur in the state $\langle A, \vec{l}, \vec{m}, \vec{d} \rangle$ as a result of this step are $l_i := 1$ and $d_i := y$. That is, $d_i := \mathbf{data}(A, m_i)$.

3.4 State mapping

We define the state mapping $h : \text{States}(X) \longrightarrow \text{States}(Y)$ as follows:

Let $x = \mathbf{active}(\,\langle A, \vec{l}, \vec{m}, \vec{d} \rangle\,)$.

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

Proving bisimilarity between X (which plays the role of the implementation) and Y (the specification) turns to proving that h satisfies **the matching criteria** [10]. For this, we heavily rely on the invariants discussed above.

Lemma 3 *For X, Y and h described above, the matching criteria hold:*

1. *(a) for all i,*
$(\forall y \in D) \ \ l_i = 0 \wedge (m_i, y) \in A \longrightarrow h(\,\langle A, \vec{l}, \vec{m}, \vec{d} \rangle\,) = h(\,\langle A, \vec{l}[l_i := 1], \vec{m}, \vec{d}[d := y] \rangle\,)$

(b) *for all* i,
$l_i = 2 \longrightarrow h(\ \langle A, \vec{l}, \vec{m}, \vec{d}\rangle\ ) = h(\ \langle A \cup \{(m_i + 1, d_i)\}, \vec{l}[l_i := 0], \vec{m}[m_i := m_i + n], \vec{d}\rangle\ )$

(internal steps in X *don't change the mapped state in* Y*)*

2. (a) *for all* i, $l_i = 1 \longrightarrow \mathbf{c} = i$ *(*v *enabled in* $X \Rightarrow v$ *enabled in* Y*)*

   (b) *for all* i, $(\forall e_i : E_i)\ \ l_i = 1 \wedge b_i(d_i, e_i) \longrightarrow \mathbf{c} = i \wedge b_i(\mathbf{d}, e_i)$
   (X *can do* $a_i(f_i(\star, e_i)) \Rightarrow Y$ *can do* $a_i(f_i(\star, e_i))$*)*

   (soundness: in each state, for each external action, if X *can do it* $\Rightarrow Y$ *can do it, too)*
   The external actions that X *can do are* v *(that can happen when* $l_i = 1$*) and* $\{a_i(f_i(d, e_i))\}$ *(that are enabled when* $l_i = 1 \wedge b_i(d_i, e_i)$ *is true).*

3. $FC(\langle A, \vec{l}, \vec{m}, \vec{d}\rangle) : \exists i (1 \leq i \leq n)$ *s.t.* $l_i = 1$ *and* $l_j = 0 \forall j \neq i$, *i.e.* $\vec{l} = (0, \cdots, 0, 1, 0, \cdots, 0)$.

   (a) *for all* i, $FC(\langle A, \vec{l}, \vec{m}, \vec{d}\rangle) \wedge i = \mathbf{c} \longrightarrow l_i = 1$

   (b) *for all* i, $(\forall e_i : E_i) FC(\langle A, \vec{l}, \vec{m}, \vec{d}\rangle) \wedge i = \mathbf{c} \wedge b_i(\mathbf{d}, e_i) \longrightarrow l_i = 1 \wedge b_i(d_i, e_i)$

   (completeness: Y *can do a step* $\Rightarrow X$ *can do that step, too - eventually after a number of internal steps)*

4. *for all* i, $(\forall e_i : E_i) b_i(d_i, e_i) \longrightarrow f_i(d_i, e_i) = f_i(\mathbf{d}, e_i)$
   (the data labels on the visible actions coincide).

5. (a) $l_i = 1 \ \longrightarrow\ h(\langle A \cup \{m_i + 1, d_i\}, \vec{l}[l_i := 0], \vec{m}[m_i := m_i + n], \vec{d}\rangle) = \langle (i + 1)\ mod\ n, \mathbf{d}\rangle$

   (b) $l_i = 1 \wedge b_i(d_i, e_i) \ \longrightarrow\ h(\langle A, \vec{l}[l_i := 2], \vec{m}, \vec{d}[d_i := g_i(d_i, e_i)]\rangle) = \langle (i + 1)\ mod\ n, g_i(\mathbf{d}, e_i)\rangle$

   (every visible action takes related states to related states, i.e. if the initial states in X *and* Y *are h-mapped, then the states after executing the action are also h-mapped)*

Proof .

1. Let $\langle A, \vec{l}, \vec{m}, \vec{d}\rangle$ be X's state before the τ-step, $(\mathbf{c}, \mathbf{d})$ the state in Y mapped from it, $\langle A', \vec{l'}, \vec{m}', \vec{d'}\rangle$ X's state after the τ-step and $(\mathbf{c}', \mathbf{d}')$ its h-mapped Y state. We have to prove that $(\mathbf{c}, \mathbf{d})$ and $(\mathbf{c}', \mathbf{d}')$ are equal.

   (a) After the τ-step, x and m_x are not changed (because A and $\vec{m}$ didn't change). The only different value is of l_x, but this doesn't affect h's definition since $l_x := l_x + 1 = 1$ is still in $\{0, 1\}$. So, $\langle \mathbf{c}', \mathbf{d}'\rangle = \langle x, \mathbf{data}(A', m'_x)\rangle = \langle x, \mathbf{data}(A, m_x)\rangle = \langle \mathbf{c}, \mathbf{d}\rangle$.

   (b) $l_i = 2 \overset{inv.(5),\text{def. } h}{\leadsto} \langle \mathbf{c}, \mathbf{d}\rangle = \langle (i + 1) \bmod n, d_i\rangle$.
   If $n > 1$ then $i = x \neq (x-1) \bmod n \overset{inv.(3)}{\leadsto} m_{(i+1) \bmod n} = m_i + 1 \leadsto \mathbf{active}(\langle A', \vec{l'}, \vec{m}', \vec{d'}\rangle) = (i + 1) \bmod n$. In the new state, $\vec{l} = 0 \leadsto l_{(i+1) \bmod n} = 0$.
   $\langle \mathbf{c}', \mathbf{d}'\rangle \overset{\text{def. } h}{=} \langle (i + 1) \bmod n, \mathbf{data}(A, m_{(i+1) \bmod n})\rangle = \langle (i + 1) \bmod n, d_i)\rangle = \langle \mathbf{c}, \mathbf{d}\rangle$.
   If $n = 1$ then $i = x = (x + 1) \bmod n$, so $\langle \mathbf{c}, \mathbf{d}\rangle = \langle i, d_i\rangle$. The active process $(i = x = 1)$ remains active in the new state (inv. (2)). $\langle \mathbf{c}', \mathbf{d}'\rangle \overset{l'_i=0,\text{def. } h}{=} \langle i, \mathbf{data}(A', m'_i)\rangle$. $m'_i = m_i + n$, i.e. $m_i + 1$ and in A' there is $(m_i + 1, d_i) \leadsto \mathbf{data}(A', m'_i) = d_i \leadsto \langle \mathbf{c}', \mathbf{d}'\rangle = \langle i, d_i\rangle = \langle \mathbf{c}, \mathbf{d}\rangle$.

2. Let $\langle A, \vec{l}, \vec{m}, \vec{d}\rangle$ denote always the current state.

   (a) $l_i = 1 \overset{inv.(5)}{\leadsto} i = \mathbf{active}(\langle A, \vec{l}, \vec{m}, \vec{d}\rangle) \overset{def.\ h}{\leadsto} \mathbf{c} = i$.

   (b) True, because $\mathbf{d} = d_i$ from lemma 2 and $l_i = 1 \longrightarrow \mathbf{c} = i$ was shown at (2(a)).

3. (a) Let i_0 be the i from FC ($l_{i_0} = 1$). Then, from inv. (5) and definition of h, $\mathbf{c} = i_0$. But $\mathbf{c} = i$, also, so $i = i_0$. This means that $l_i = l_{i_0} = 1$.

   (b) $l_i = 1$ is shown with the same reasoning as in 3(a). $b_i(d_i, e_i)$ is true because $b_i(\mathbf{d}, e_i)$ is true and $\mathbf{d} = d_i$ (lemma 2).

4. The conditions $b_i(d, e_i)$ are evaluated when $l_i = 1$. By lemma 2 and definition of h, we get $\mathbf{d} = d_i$, so $f_i(d_i, e_i) = f_i(\mathbf{d}, e_i)$.

5. Again, we will denote $\langle A, \vec{l}, \vec{m}, \vec{d}\rangle$ and $\langle A', \vec{l}', \vec{m}', \vec{d}'\rangle$ the states of X before and after executing the discussed action. Similarly, $\langle \mathbf{c}, \mathbf{d}\rangle$ and $\langle \mathbf{c}', \mathbf{d}'\rangle$ are the corresponding states in Y.

   (a) $l_i = 1 \overset{inv.(5),lemma\ 2}{\rightsquigarrow} \langle \mathbf{c}, \mathbf{d}\rangle = \langle i, d_i\rangle$. $\mathbf{active}(\langle A', \vec{l}', \vec{m}', \vec{d}'\rangle) = (i+1) \bmod n$, by a reasoning similar to 1(b). Because $l'_{(i+1) \bmod n} = 0$, we have $\mathbf{c}' = (i+1) \bmod n$ and $\mathbf{d}' = d_i$. But according to lemma 2, $\mathbf{d} = d_i$. So, $\langle \mathbf{c}', \mathbf{d}'\rangle$ is indeed $\langle (i+1) \bmod n, \mathbf{d}\rangle$.

   (b) $l'_i = 2 \overset{inv.(5)}{\rightsquigarrow} \mathbf{active}(\langle A', \vec{l}', \vec{m}', \vec{d}'\rangle) = i \overset{\text{def. } h, l'_i=2}{\rightsquigarrow} \langle \mathbf{c}', \mathbf{d}'\rangle = \langle (i+1) \bmod n, d'_i\rangle =$ $\langle (i+1) \bmod n, g_i(d_i, e_i)\rangle$.

So far we have proved that
$\tau_{\{R,W_2\}}\partial_{\{r,w,Read,Write\}}(\ \|_i X_i(i) \| \mathsf{SPLICE}(\{(0,d)\})\ )\,[W_1 \longrightarrow v] \ = \ X(\{(0,d)\}, \vec{0}, (0,1,\cdots,n-1), \vec{0})$
and
$X(\{(0,d)\}, \vec{0}, (0,1,\cdots,n-1), \vec{0}) \ = \ Y(0,d)$.

It's left now to establish: $\tau.\tau_{\{v\}}Y(0,d) = \tau.Spec(d)$

3.5 Abstraction

The definition (3.3) says that

$$Y(\mathbf{c},\mathbf{d}) = v.\,Y((\mathbf{c}+1) \bmod n, \mathbf{d}) + \sum_{e_\mathbf{c}:E_\mathbf{c}} (a_\mathbf{c}(f_\mathbf{c}(d, e_\mathbf{c})).\,Y((\mathbf{c}+1) \bmod n, g_\mathbf{c}(d, e_\mathbf{c})) \triangleleft b_\mathbf{c}(\mathbf{d}, e_\mathbf{c}) \triangleright \delta)$$

(for $\mathbf{c} \in \{0 \cdots n-1\}$)
i.e.,

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

$Y(0,d) \cdots Y(n-1,d)$ form a $\{v\} - cluster$, with exits

$$\{a_i(f_i(d,e_i)).\,Y((i+1) \bmod n, g_i(d,e_i)) \triangleleft b_i(d,e_i) \triangleright \delta,\ \forall i\ :\ 0 \le i < n \text{ and } \forall e_i \in E_i\}$$

CFAR [7] (Cluster Fair Abstraction Rule) states that in a fair execution, one of the exits will eventually be taken. In our case, this means:

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

and similarly

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

From this it follows that $\tau.\tau_{\{v\}}Y(0,d) = \tau.\tau_{\{v\}}Y(i,d)$, for $i \in \{1\cdots n-1\}$ (transitivity). So, we can write

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

By applying RSP we see that $\tau.\tau_{\{v\}}Y(0,d)$ is a solution of $\tau.Spec(d)$ (1.1)

This ends the proof of theorem 1.

4. Conclusions

We have proved that, from a functional point of view, the architecture Basic Splice is very expressive: any μCRL requirements specification has a distributed implementation on it. This extends a result in [12] which states that all finite processes can be distributedly implemented on Basic Splice.

Further work includes investigating other distribution schemes, based on different criteria. We should 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 Basic Splice model or to consider weaker equivalences between specification and implementation.

Acknowledgments. Thanks to Bas Luttik and Jaco van de Pol for valuable discussions and suggestions.

References

1. 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*, volume 1, pages 149–155, 2000.
2. Maarten Boasson. Control systems software. *IEEE Transactions on Automatic Control*, 38(7):1094–1106, July 1993.
3. M. Bonsangue, J. Kok, M. Boasson, and E. de Jong. A software architecture for distributed control systems and its transition system semantics. In *Proceedings of the 1998 ACM Symposium on Applied Computing (SAC'98)*.
4. M. Bonsangue, J. Kok, and G. Zavattaro. Comparing software architectures for coordination languages. In P. Ciancarini and A.L. Wolf, editors, *Proceedings of Coordination'99*, volume 1594 of *Lecture Notes in Computer Science*, pages 150–165. Springer Verlag.
5. Antonio Brogi and Jean-Marie Jaquet. On the expressiveness of coordination models. In P. Ciancarini and L. Wolf, editors, *Coordination languages and models: Third International Conference*, volume 1594 of *Lecture Notes in Computer Science*, pages 134–149, Amsterdam, 1999. Springer-Verlag.
6. 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.
7. Wan Fokkink. *Introduction to Process Algebra.* Texts in Theoretical Computer Science, An EATCS Series. Springer-Verlag, 2000.
8. Rob J. Van Glabbeek and W. Peter Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, May 1996.
9. Jan Friso Groote, Alban Ponse, and Yaroslav Usenko. Linearization in parallel pcrl. Report SEN-R0019, CWI, July 2000.
10. Jan Friso Groote and Jan Springintveld. Focus points and convergent process operators. a proof strategy for protocol verification. Report CS-R9566, CWI, 1995.
11. Rom Langerak. *Transformations and Semantics for LOTOS.* PhD thesis, Twente University, 1992.
12. Jaco van de Pol. Expressiveness of Basic Splice. Report SEN-R0033, CWI, December 2000.