Expressiveness of Basic Splice

Jaco van de Pol
Centrum voor Wiskunde en Informatica
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
Email: Jaco.van.de.Pol@cwi.nl

ABSTRACT
We study a simple software architecture, in which application processes are coordinated by writing into and reading from a global set. This architecture underlies Splice, which is developed and used at the company Hollandse Signaalapparaten. Our approach is distinguished by viewing the architecture as a component itself, described formally by means of process algebra.

Two results are proved. First a distributed implementation of the architecture is given, in which each component maintains a local set and data items are exchanged between these local sets. The implementation is proved to be behaviourally equivalent to the conceptual view of having one global set. Next we show that every requirements specification expressible as a finite process has a distributed implementation on this architecture.

2000 Mathematics Subject Classification: 68M14, 68N30, 68Q85
Keywords and Phrases: Splice, shared data space, software architecture, distributed systems, expressiveness
Note: Research carried out in SEN 2 "Specification and analysis of embedded systems"

1. Introduction

The complexity of designing distributed systems is generally managed by introducing an *architecture*, defining how application processes are coordinated. In this way two separate tasks emerge. First, the architecture must be implemented on a distributed network. Second, application processes have to be designed that 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 may demand much overhead from the distributed implementation, or even make the architecture unrealizable.

In this paper we study the core of Splice [4] (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. Application processes can publish data and receive data to which they have subscribed. Each process is accompanied by an agent, which stores data items locally, and forwards these to agents of subscribed processes. Recent research papers propose to view Splice conceptually as a shared data space, i.e. a set of data common to all processes [3, 10].

The advantage of the Splice architecture is that the application processes are loosely coupled, thus increasing the amount of fault tolerance [4]. The data is present at several locations, making replication of processes relatively easy. 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 processes [10].

In this paper, the coordination primitives between application processes are restricted to writing and reading. Informally speaking, $write(v)$ adds value v to the global set; if v is present already this action has no effect. The other primitive, $read(v)$ denotes a non-destructive, blocking read. That is, it waits until it actually finds v in the global set, and then proceeds.

As an example, consider the very simple system specification $a.c$, indicating the sequential composition of the action a followed by the action c. In a distributed implementation, actions a and c might occur in different application processes, e.g. $a.write(0) \mid\mid read(0).c$, where $\mid\mid$ denotes parallel composition, and 0 is just an arbitrary value. 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 $a.write(0).read(0).c$. If we hide the communication actions *read* and *write*, we indeed get the desired system behaviour $a.c$.

1.1 The Formal Approach

Any formal semantics of the architecture must make sure somehow that $read(0)$ doesn't occur before $write(0)$. A common theme has been to embed the primitives in a host language, and give semantics to the coordination language thus obtained. Some related work provides operational [5, 6] or compositional [3] semantics for the global set with read/write. These papers involve a redefinition of the constructs of the host language, especially the parallel composition operator. As an alternative, we propose to view the architecture as a separate component, put in parallel with the application processes.

To this end we define the architecture, called *Basic Splice*, by the following recursive specification, parameterized with the current set A of values of sort D (the type of the values D is intentionally left unspecified):

$$\begin{aligned} S(A:Set) &= \sum_{v:D} Write(v).S(A \cup \{v\}) \\ &+ \sum_{v:D} Read(v).S(A) \lhd v \in A \rhd \delta \end{aligned}$$

This definition uses μCRL-notation [15], which is an extension of the standard process algebra ACP [1] with algebraic data types. Here + denotes non-deterministic choice between processes, $\sum$ denotes alternative choice over a data type, and $x \lhd b \rhd y$ denotes the process "x if b, else y"; δ denotes deadlock, the unit of +. 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. Absence of data cannot be tested.

Note that application processes use the actions *read* and *write* to request the primitives, whereas the architecture S uses actions *Read* and *Write*, to denote the actual occurrence of these primitives. Of course, these actions should synchronize (cf. function calls or method invocations). This is captured in the formalism by providing a communication function: $Read \mid read = R$ and $Write \mid write = W$. As usually in μCRL, the unsynchronized actions are *encapsulated* by the $\partial_{\{Read,read,Write,write\}}$ 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 is now captured formally by the following μCRL-expression:

$$\tau_{\{R,W\}}(\partial_{\{Read,Write,read,write\}}(S(\emptyset) \mid\mid a.write(0) \mid\mid read(0).b))$$

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

In this way, the formal semantics of the coordination primitives is given by a two-line definition of S. The semantics of application processes is obtained by putting them in parallel with S. This approach makes the architecture quite explicit: it is just another process. As an advantage we mention that we can now use existing theory and tools for process algebra to reason about application processes on Basic Splice. Another advantage of this approach is that an implementation of Basic Splice can be obtained by refining S to a distributed process with equivalent behaviour, whereas [5, 3] must give a new semantics to the coordination language when defining a distributed implementation.

1.2 Results

From the separate tasks we mentioned in the first lines of the introduction, two natural questions arise, which apply to any architecture:

1. Realizability: does Basic Splice itself have an efficient distributed implementation?
2. Expressiveness: does Basic Splice support the distributed implementation of system requirements?

Section 3 addresses the first question, by defining a distributed implementation of Basic Splice in which every component has its own local set. Data items are exchanged between these local sets. So a write is not a single atomic operation. Nevertheless, we show that this implementation is behaviourally equivalent to S. The proof simplifies and slightly generalizes the work of [3]. The result is also conform one of the results in [5, 6]. In Section 4 the other question is addressed, by presenting a positive expressiveness result. In particular we show that any requirements specification represented as finite process has a distributed implementation on Basic Splice. This main result is not comparable with existing results on expressiveness, as far as we know.

Both results are proved in a formal manner. The proofs are sufficiently detailed in order to be checked mechanically. In Section 2 recaptures and develops the necessary proof apparatus. In Section 5 we show some implications of our theorems for the design of architectures and coordination languages. We also discuss related previous work and possible future work.

Acknowledgments I like to thank Roel Bloo, Marcello Bonsangue, Paul Dechering, Jozef Hooman and Izak van Langevelde, for fruitful discussions on their papers. Furthermore, discussions with Jan Bergstra, Rashindra Manniesing, Sjouke Mauw and Simona Orzan stimulated this research. Special thanks go to Wan Fokkink for his cooperation in the development of the fairness rule used in this paper. Finally, this research is stimulated by STW/Progress project CES.5009 on distributed shared data spaces.

2. Preliminaries: Process Algebra with Data

For a recent introduction to process algebra see [12]. We will present and prove our ideas using the formalism μCRL [15], which is a combination of the standard process algebra ACP [1] with abstract data types. In the introduction we shortly reviewed the needed process operators. We use sequential, alternative and parallel composition ($.$, $+$, $||$) and encapsulation and hiding (∂_H, τ_I) from ACP. From μCRL we borrow parameterized actions (like $read(v)$), summation over data ($\sum$) and the conditional ($x \lhd b \rhd y$).

Process algebra supports the notion of a correct refinement in the following way. Typically, processes *Spec* and *Impl* are given, representing the high-level specification of a system, and its distributed implementation as a number of communicating processes, respectively. The fact that *Impl* correctly refines *Spec* is expressed as the equality $\tau_I(\mathit{Impl}) = \mathit{Spec}$, where I contains the set of non-observable, internal communications between the components of the implementation.

As equivalence relation between processes we use branching bisimulation [14], which is slightly finer than weak bisimulation. Hence our results also apply to weak bisimulation. In [15] branching bisimulation on μCRL 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* (LPE). In [16] it is demonstrated that a large class of μCRL can be transformed to this format.

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:

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

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 τ, denoting hidden steps. An LPE is *convergent*, if it doesn't admit infinite sequences of τ-steps. In [2] the principle CL-RSP (Recursive Specification Principle for Convergent LPEs) is postulated, which states that a convergent LPE has a unique solution. That paper also introduces the notion of *invariant*. A predicate $I(d)$ is an *invariant* if and only if it is preserved by all transitions, formally iff the following conjunction holds:

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

In [17] the *focus and cones* method is developed 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 τ-loops exist, we need a fairness assumption on executions in order to ensure that eventually an exit from the τ-loop is chosen. To this end, a new fairness rule for non-convergent LPEs will be introduced in Section 2.2.

2.1 State mappings, Cones and Focus Points

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

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

Note that the specification must not contain τ-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 τ-steps are possible. The focus points can be easily characterized by the focus condition: $FC(d) = \bigwedge\!\!\!\bigwedge_{i\in Int} \neg\exists(e_i:E_i).\, b_i(d,e_i)$.

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

1. for each $i \in Int$, $\forall(e_i:E_i).\, b_i(d,e_i) \to h(d) = h(g_i(d,e_i))$
   i.e. internal steps don't change the related state.

2. for each $i \in Ext$, $\forall(e_i:E_i).\, b_i(d,e_i) \to b'_i(h(d),e_i)$
   i.e. the specification can mimic all external steps of the implementation (soundness).

3. for each $i \in Ext$, $\forall(e_i:E_i).\, b'_i(h(d),e_i) \wedge FC(d) \to b_i(d,e_i)$
   i.e. each external step of the specification can be mimicked in the related focus points of the implementation (completeness).

4. for each $i \in \mathit{Ext}$, $\forall(e_i : E_i).\, b_i(d, e_i) \to f_i(d, e_i) = f'_i(h(d), e_i)$
i.e. the data labels on the external transitions coincide.

5. for each $i \in \mathit{Ext}$, $\forall(e_i : E_i).\, b_i(d, e_i) \to h(g_i(d, e_i)) = g'_i(h(d), e_i)$
i.e. the next states after a visible transition are related.

Theorem 1 (from [17]) *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 $\forall(d : D).\, I(d) \to MC_h(d)$, we have*

$$\mathit{Spec}(d) \lhd FC(d) \rhd \tau.\mathit{Spec}(d) \quad = \quad \mathit{Impl}(h(d)) \lhd FC(d) \rhd \tau.\mathit{Impl}(h(d))$$

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 Fair abstraction

The focus and cones method only works for convergent LPEs. But we will encounter τ-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. As far as we know, the only fair abstraction rule formulated on recursive specifications with data parameters in the literature [17] is KFAR$_1$ (Koomen's Fair Abstraction Rule for τ-loops of length 1). In branching bisimulation, KFAR$_n$ for $n > 1$ cannot be proved from this. Therefore, we will now formulate a general fairness rule for LPEs (joint with Wan Fokkink). This is an adaptation of the Cluster Fair Abstraction Rule for process algebra [21].

L-CFAR is based on a cluster function CF, which identifies the clusters (= points in the same τ-loop). Let J be the index set, divided into $J = \mathit{Int} \cup \mathit{Ext}$, where $\mathit{Int} \subseteq A$ is the set of actions that will be hidden, including τ. Given an LPE of the form

$$X(d : D) = \sum_{i \in J} \sum_{e_i : E_i} a_i(f_i(d, e_i)).X(g_i(d, e_i)) \lhd b_i(d, e_i) \rhd \delta,$$

a *cluster function* CF is a function $CF : D \to D'$ to some new domain D', such that for all $d, e : D$, if $CF(d) = CF(e)$ then $X(d) \to^*_{\mathit{Int}} X(e)$. (i.e. d and e are in the same cluster). In fact, given an invariant I, this can be weakened into: $\forall(d, e : D).\, I(d) \wedge I(e) \wedge CF(d) = CF(e) \Rightarrow X(d) \to^*_{\mathit{Int}} X(e)$.

After abstraction from actions in Int, all states in a cluster can be identified, and only the exits from the cluster are preserved. So from a given state d', we can perform all transitions from any d in the corresponding cluster, that are either externally visible or exit the cluster. So the abstracted process will be:

$$\begin{aligned} X_{CF}(d' : D') \quad = \quad & \sum_{i \in \mathit{Int}} \sum_{d:D} \sum_{e_i:E_i} \tau.X_{CF}(CF(g_i(d, e_i))) \\ & \qquad \lhd b_i(d, e_i) \wedge CF(d) = d' \wedge CF(g_i(d, e_i)) \neq d' \rhd \delta \\ + \quad & \sum_{a \in \mathit{Ext}} \sum_{d:D} \sum_{e_i:E_i} a_i(f_i(d, e_i)).X_{CF}(CF(g_i(d, e_i))) \\ & \qquad \lhd b_i(d, e_i) \wedge CF(d) = d' \rhd \delta \end{aligned}$$

Theorem 2 *If CF is a cluster function, then $\tau.\tau_{\mathit{Int}}(X(d)) = \tau.X_{CF}(CF(d))$.*

This theorem will be proved in a separate paper.

3. Distributed Implementation

In this section a distributed implementation of Basic Splice is defined and a correctness proof is given. We will use a number of standard datatypes with the usual operations: *Bool* (booleans), *Nat* (natural numbers), D (arbitrary data values), *Set* (finite sets over D) and *List*, (finite lists over *Set*). Besides the normal mathematical functions we introduce the following: For $m, x : Nat$, $m > 0$, we define $x \bmod m$ as the representant $0 \leq p < m$, such that $p = x \pmod{m}$. For $A : Set$ and $v : D$, $A + v$ denotes $A \cup \{v\}$.

In μCRL such data types are defined by algebraic data type specifications. It is routine to specify these types algebraically. Here we only provide the definition of *List*. It has constructors ϵ (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 $L_0, \ldots, L_{i-1}, L_i + v, L_{i+1}, \ldots, L_n$. When necessary, $L[i : +v]$ extends L with empty sets to have length at least i, and adds v to L_i.

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

In order to define the distributed implementation of Basic Splice, we need the location where a read or write occurs. To this end, we introduce actions $Read(i : Nat, v : D)$ and $Write(i : Nat, v : D)$, where i denotes the service access point and v the datum. The location is also incorporated in the specification, for comparison with the distributed variants. So we slightly modify S into:

$$\begin{array}{rcl} S_1(A : Set) & = & \sum_{i:Nat} \sum_{v:D} Write(i, v).S_1(A + v) \\ & + & \sum_{i:Nat} \sum_{v:D} Read(i, v).S_1(A) \lhd v \in A \rhd \delta \end{array}$$

Thus, S_1 maintains a set A, into which any application process i can write an element v unconditionally, or read an existing value. In the distributed version S_2, 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 S_2 has as parameters the lists K and L and is defined as follows:

$$\begin{array}{rcl} S_2(K, L : List) & = & \sum_{i:Nat} \sum_{v:D} Write(i, v).S_2(K[i : +v], L) \\ & + & \sum_{i:Nat} \sum_{v:D} Read(i, v).S_2(K, L) \lhd v \in L_i \rhd \delta \\ & + & \sum_{v:D} \sum_{i,j:Nat} Send(i, v, j).S_2(K, L[j : +v]) \lhd v \in K_i \setminus L_j \rhd \delta \end{array}$$

According to S_2, 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 $S_1(\emptyset) = \tau_{\{Send\}}(S_2(\epsilon, \epsilon))$.

Proof. We view S_1 as a specification and $\tau_{\{Send\}}(S_2)$ as its implementation; the latter equals S_2 with $Send(i, v, j)$ replace by τ. 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 $Inv = \forall i. L_i \subseteq \bigcup K$, which can be checked easily. The focus condition $FC(K, L)$ 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 $\tau_{\{Send\}}(S_2)$ the number $\sum_i \sum_j \#(K_i \setminus L_j)$ decreases with each τ-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) $(v \in \bigcup K \cup \bigcup L) \wedge (\forall j.\, L_j = \bigcup K) \rightarrow (v \in L_i)$

(5) $(\bigcup K \cup \bigcup L) + v = \bigcup K[i : +v] \cup \bigcup L$

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 1. ☒

In fact this means that S_1 and $\tau_{\{Send\}}(S_2)$ are indistinguishable. This generalizes [5, 3], 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 [17].

4. Expressiveness

In this section we will investigate the expressiveness of Basic Splice. We study this from a system engineering point of view: given the requirements specification of a system under design, can a distributed implementation on Basic Splice be constructed? We now define our terminology.

A *requirements specification* is a process X over some alphabet a_1, a_2, ... An *implementation of X on Basic Splice* is defined to be a number of application processes $P_1, \ldots, P_n$, such that

$$X = \tau_{\{R,W\}} \partial_{\{read, Read, write, Write\}} (S(\emptyset) \,||\, P_1 \,||\, \cdots \,||\, P_n).$$

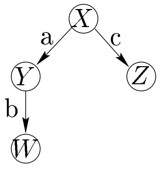
An implementation is called *distributed* if for each i, P_i only uses the actions $\{read, write, a_j\}$ for some j. This implies that each action occurs at a different location, and that processes cannot directly communicate with each other, but only via the coordination primitives.

In the sequel we will construct P_i for specifications of the form $\tau.X.\delta$, where X is a finite process, i.e. a term built from $a_1, \ldots$ using finitely many applications of '+' and '.'. The restriction to finite specifications is used in our construction of P_i and in the correctness proof, but it is probably not needed for the result to hold.

4.1 Translation

Let a finite process X, the specification, be given. Such processes can always be written as a "basic term" of the form $X = a_1.X_1 + \cdots + a_n.X_n$, where a_i are atomic actions and each X_i is again a basic term, or absent in case of termination. Consider this process as a tree with root X. With V we denote the nodes in this tree, and with E we denote the set of edges. An edge can be represented by the pair (α, β) of its source and destination, and the label on it is called $\lambda_{\alpha\beta}$. The outgoing edges from a node are considered ordered; with μ_α we denote the number of outgoing edges from α and for $i < \mu_\alpha$, $\alpha[i]$ denotes the i-th successor of α (counting from 0). Furthermore, ν_β denotes the index of node β among α's children (so $\alpha[\nu_\beta] = \beta$, for $(\alpha, \beta) \in E$). With $\alpha \sqsubset \beta$ we denote that α is a predecessor of β (so $\sqsubset$ is the transitive closure of E).

Example 4 *The graph of $a.b + c$ can be depicted as follows*



$V = \{X, Y, Z, W\}$
$E = \{(X, Y), (X, Z), (Y, W)\}$

$\lambda_{XZ} = c$
$\mu_X = 2$
$X[1] = Z$
$\nu_Z = 1$

The idea for the distributed implementation is as follows. Each edge $(\alpha, \beta) \in E$ will correspond to an application process $P_{\alpha\beta}$. These coordinate by writing certain data to the shared data space, encoding the current state. Each process is blocked, until it succeeds to read a certain value.

For instance, if P_{XY} (Example 4) executes a, it will write the value that unblocks P_{YW}. The main problem is to resolve the choices, for instance between a and c. The solution is that P_{XY} and P_{XZ} pass around some token. Due to the fact that values in the data space cannot be removed, the token is implemented by a strictly increasing sequence number. The process in possession of the current token decides to either execute its action, or pass the token around. In every fair execution, a choice is made eventually.

So we instantiate the actual type of values in the data space (called D earlier) to $Nat \times V$, pairs of sequence numbers and the current state according to the specification. A pair (m, α) means that edge $(\alpha, \alpha[m \bmod \mu_\alpha])$ has the token.

For each $(\alpha, \beta) \in E$, we now define process $P_{\alpha\beta}$ recursively (with parameter m) as follows:

$$P_{\alpha\beta}(m : Nat) = read(m, \alpha).\big(\lambda_{\alpha\beta}.write(0, \beta) + write(m+1, \alpha).P_{\alpha\beta}(m + \mu_\alpha)\big)$$

These processes are initialized with their index, ν_β. The complete distributed implementation of X on Basic Splice consists of the processes $P_{\alpha\beta}(\nu_\beta)$ for $\alpha\beta \in E$. For Example 4, one of the processes is (with initial value $m = 0$):

$$P_{XY}(m) = read(m, X).\big(a.write(0, Y) + write(m+1, X).P_{XY}(m+2)\big)$$

Note that some internal activity takes place before the first action is executed. After the process has finished, the data space is still waiting for more read and write requests, so the implementation doesn't terminate. This motivates the form of the correctness criterion:

Theorem 5 (Main theorem) *Let $H = \{read, write, Read, Write\}$ and $I = \{R, W\}$, and let $\omega \in V$ be the root node in the graph associated with X. Then*

$$\tau.X.\delta = \tau.\tau_I(\partial_H(S(\{(0,\omega)\}) \,||\, (\mathop{||}_{\alpha\beta \in E} P_{\alpha\beta}(\nu_\beta))))$$

The subsequent sections describe the proof of the main theorem. This proceeds via various stages. First we transform the implementation to an LPE (P) by standard computations. Then a state mapping from an abstract version of P (P') to a process Y is defined, where the global set is eliminated and the sequence numbers are wrapped around using modulo arithmetic. This introduces τ-cycles of arbitrary length, which are eliminated by an application of the fair abstraction rule L-CFAR, leading to Z. Finally, Z appears to be the linearized form of X. The full proofs, including a number of invariants, can be found in the Appendix.

4.2 Linearization

In this paragraph we provide the linearization of the implementation and the specification. The main tool of linearization is making the process state explicit. To this end, we introduce the following four locations for each $P_{\alpha\beta}$: ℓ_0 (ready to read), ℓ_1 (unblocked), ℓ_2 (action performed) or ℓ_3 (finished). The state is completely determined by the pair (m, s), where m is the sequence number and s the location. Due to the asynchronous communication via read and write, all processes $P_{\alpha\beta}$ are independent. Thus

Basic Splice simplifies the linearization greatly. The implementation can be linearized as follows:

$$\begin{array}{ll}
\multicolumn{2}{l}{P(A,\dots,m_{\alpha\beta},s_{\alpha\beta},\dots) =} \\
\multicolumn{2}{l}{\sum_{\alpha\beta\in E}(} \\
 & R(m_{\alpha\beta},\alpha).P[s_{\alpha\beta} := \ell_1] \\
 & \lhd (m_{\alpha\beta},\alpha) \in A \land s_{\alpha\beta} = \ell_0 \rhd \delta \\
+ & \lambda_{\alpha\beta}.P[s_{\alpha\beta} := \ell_2] \\
 & \lhd s_{\alpha\beta} = \ell_1 \rhd \delta \\
+ & W(0,\beta).P[A := A + (0,\beta), s_{\alpha\beta} := \ell_3] \\
 & \lhd s_{\alpha\beta} = \ell_2 \rhd \delta \\
+ & W(m+1,\alpha).P[A := A + (m_{\alpha\beta}+1,\alpha), m_{\alpha\beta} := m_{\alpha\beta} + \mu_\alpha, s_{\alpha\beta} := \ell_0] \\
 & \lhd s_{\alpha\beta} = \ell_1 \rhd \delta)
\end{array}$$

The LPE is parameterized with the set A, and with the sequence numbers $m_{\alpha\beta}$ and locations $s_{\alpha\beta}$, for each $\alpha\beta \in E$. With the notation $P[d := k]$ we denote the recursive call to P where parameter d is set to k and all other parameters remain unchanged. Correctness of linearization follows from [16], yielding the following:

Proposition 6 *Let $H = \{write, read, Write, Read\}$ and ω the root node. Then*

$$\partial_H(S(\{(0,\omega)\}) \,||\, (\mathop{||}_{\alpha\beta\in E} P_{\alpha\beta}(\nu_\beta))) = P(\{(0,\omega)\},\dots,\nu_\beta,\ell_0,\dots)$$

Also, the specification X needs to be linearized, in order to apply the focus and cones method. This is done by introducing a parameter $a \in V$, which indicates the current state of X. There is a summand for each edge $\alpha\beta \in E$. Thus we define:

$$X(a) = \sum_{\alpha\beta\in E} \lambda_{\alpha\beta}.X(\beta) \lhd a = \alpha \rhd \delta$$

Proposition 7 *Let ω be the root node in V. Then $X.\delta = X(\omega)$*

Note that $X(\omega)$ doesn't have the option to terminate. This makes the definition simpler, and termination is spoiled already by S, as we have noticed before.

4.3 State Mapping

Note that abstracting in P from all R- and W-actions would introduce infinite τ-sequences. The resulting process would not be convergent, hence the focus and cones method would not be applicable. Therefore, we introduce a process P', which is a pre-abstraction of P. In P', actions R and $W(\beta,0)$ are hidden, but the actions $W(\alpha,m+1)$, giving rise to τ-divergence, are renamed to a single action i, which keeps their occurrence externally visible. In Section 4.4 the visible action i will be abstracted too, by using the fair abstraction principle L-CFAR to get rid of the i-loops.

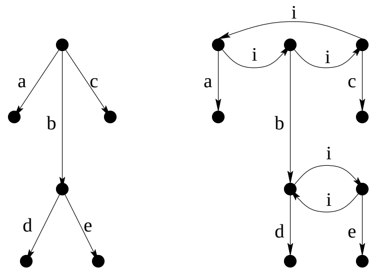
$$\begin{array}{ll}
\multicolumn{2}{l}{P'(A,\dots,m_{\alpha\beta},s_{\alpha\beta},\dots) =} \\
\multicolumn{2}{l}{\sum_{\alpha\beta\in E}(} \\
 & \tau.P'[s_{\alpha\beta} := \ell_1] \\
 & \lhd (m_{\alpha\beta},\alpha) \in A \land s_{\alpha\beta} = \ell_0 \rhd \delta \\
+ & \lambda_{\alpha\beta}.P'[s_{\alpha\beta} := \ell_2] \\
 & \lhd s_{\alpha\beta} = \ell_1 \rhd \delta \\
+ & \tau.P'[A := A + (0,\beta), s_{\alpha\beta} := \ell_3] \\
 & \lhd s_{\alpha\beta} = \ell_2 \rhd \delta \\
+ & i.P'[A := A + (m_{\alpha\beta}+1,\alpha), m_{\alpha\beta} := m_{\alpha\beta} + \mu_\alpha, s_{\alpha\beta} := \ell_0] \\
 & \lhd s_{\alpha\beta} = \ell_1 \rhd \delta)
\end{array}$$

Proposition 8 *For any $\vec{d}$, we have* $\tau_{\{R,W\}}(P(\vec{d})) = \tau_{\{i\}}(P'(\vec{d}))$

Due to hiding, the process P' has internal activity (τ) which can be reduced modulo branching bisimulation. To this end we introduce an intermediate process $Y(a,m)$ without internal activity. We also change the representation of the state considerably. In Y, the global set A is eliminated and the sequence numbers are wrapped around using modulo arithmetic. A parameter a tells in which corresponding state of X we are, and m tells which child of a in X has the token.

$$\begin{array}{rcl} Y(a,m) & = & \sum_{\alpha\beta\in E} \lambda_{\alpha\beta}.Y(\beta,0) \triangleleft a=\alpha \wedge \nu_\beta = m \triangleright \delta \\ & + & \sum_{\alpha\beta\in E} i.Y(a,(m+1) \bmod \mu_\alpha) \triangleleft a=\alpha \wedge \nu_\beta = m \triangleright \delta \end{array}$$

Example 9 *Given $X = a + b.(d+e) + c$ (on the left below), process Y is depicted on the right as follows:*



We will prove that Y is equivalent to P', by giving an appropriate state mapping h. So h should compute the pair (a,m) corresponding to the state vector $\vec{d} = (A,\ldots,m_{\alpha\beta},s_{\alpha\beta},\ldots)$. This mapping is defined in two stages. First, the newest pair (σ,p) is found in the data space, corresponding to some edge $(\sigma,\tau) \in E$.

$$H(A) = (\sigma,p,\tau), \text{ where } \left\{ \begin{array}{l} \sigma = \max_{\sqsupset}\{x \mid (0,x) \in A\}, \\ p = \max_{>}\{p \mid (p,\sigma) \in A\}, \\ \tau = \left\{ \begin{array}{l} \sigma[p \bmod \mu_\sigma], \text{ if } \mu_\sigma > 0 \\ \bot, \text{ otherwise} \end{array} \right. \end{array} \right.$$

In many cases, $(\sigma, p \bmod \mu_\sigma)$ is the pair (a,m) that we look for. However, note that in case $s_{\sigma\tau} = \ell_2$, the action $\lambda_{\sigma\tau}$ has been executed already. In this case the current state is in fact τ (except when σ is a leaf, i.e. $\mu_\sigma = 0$). This explains the case distinction in the following definition of the state mapping h:

$$h(\vec{d}) = (a,m), \text{ where } \left\{ \begin{array}{l} (\sigma,\rho,\tau) = H(A) \\ a = \left\{ \begin{array}{l} \tau, \text{ if } \tau \neq \bot \text{ and } s_{\sigma\tau} = \ell_2 \\ \sigma, \text{ otherwise} \end{array} \right. \\ m = \left\{ \begin{array}{l} 0, \text{ if } \tau = \bot \text{ or } s_{\sigma\tau} = \ell_2 \\ p \bmod \mu_\sigma, \text{ otherwise} \end{array} \right. \end{array} \right.$$

Proposition 10 *Let ω be the root node of V. Then*
$\tau.P'(\{(0,\omega)\},\ldots,\nu_\beta,\ell_0,\ldots) = \tau.Y(\omega,0)$

Proof (sketch). The proof consists of checking certain invariants (Lemma 14–17), and checking the matching criteria (Lemma 19), which can be computed from the state mapping h by the scheme given in Section 2. The proposition then follows directly from Theorem 1, with P' as *Impl* and Y as *Spec.* ⊠

4.4 Applying Fair Abstraction

We now apply L-CFAR to the process $Y(a,m)$ from the previous section, in order to eliminate the clusters of τ-loops. Note that Y takes its parameters from $V \times Nat$. The parameter m travels along the clusters, so we eliminate this parameter by choosing as the target type V. The cluster function is defined as: $F(a,m) = a$. We now have to show that F is indeed a cluster function.

Lemma 11 *F is a cluster function*

Proof. We first need the invariant $J(a,m) := 0 \leq m < \mu_a$. Invariance of J is easy to check. Now let $F(a,m) = F(a',m')$; then $a = a'$. Now assume $J(a,m)$ and $J(a,m')$; we must show $Y(a,m) \rightarrow_i^* Y(a,m')$. This can be proved by induction on $(m'-m)$ mod μ_a. ⊠

Now applying abstraction as in Theorem 2 we immediately get the following LPE Z, with the connected proposition:

$$\begin{aligned} Z(a') &= \sum_{\alpha\beta\in E}\sum_{a,m}\lambda_{\alpha\beta}.Z(\beta) \triangleleft a=\alpha \wedge \nu_\beta = m \wedge a' = a \triangleright \delta \\ &+ \sum_{\alpha\beta\in E}\sum_{a,m}\tau.Z(a) \triangleleft a=\alpha \wedge a' = a \wedge a \neq a' \triangleright \delta \end{aligned}$$

Proposition 12 *Let ω be the root node in V. Then $\tau.\tau_{\{i\}}(Y(\omega,0)) = \tau.Z(\omega)$.*

Proposition 13 *$Z(\omega) = X(\omega)$, where ω is the root node in V.*

Finally, propositions 7, 13, 12, 10, 8 and 6 can be combined into a proof of the Main Theorem 5, indicating that Basic Splice is sufficiently expressive from a functional point of view.

5. Conclusion

We have studied the architecture Basic Splice, 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.

Basic Splice 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 [9]. However, these don't come for free. Either, we have to give up the global view, as shown in [5, 6], or complicated protocols are needed in order to guarantee global consistency, as the two-phase-commit protocol in JavaSpaces$^{\text{tm}}$ [13]. The former compromises ease of application program construction and analysis, the latter might comprise performance on a different level.

5.1 Future work

There are many possibilities for future work. It seems possible to extend the expressiveness results to specifications in full μCRL-specifications, instead of finite processes only. Furthermore, the distributed implementation might be further refined to incorporate agents as in [4, 11].

It would also be interesting to describe a larger part of Splice, or other architectures like JavaSpaces, in μCRL. This would make automatic verification of programs communicating via such architectures by means of model checkers possible.

As final direction we mention the study of fault tolerance. Our translation scheme is not sufficient in the presence of faults. It would be interesting to investigate which coordination primitives are needed to admit a fault tolerant implementation of distributed commitment, along the lines of Lynch [19].

5.2 Related Work

In [11] a more detailed description of a Splice fragment is given, at the level of agents communicating on an Ethernet network. However, an abstract specification of this fragment is not given. Instead the authors verify that a number of frequent scenarios satisfy certain desired temporal logic properties.

The distributed implementation that we give is at the same level of abstraction as in [3, 5, 6]. This is sufficient to show that for read/write primitives a global set is equivalent to a number of local sets. In [5, 6] 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 [8]. In [3] 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 application processes may use recursion and internal choice. This combination has not been considered in [3, 5, 6]. 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 [17].

In [3] 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 S_2. 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 [7], where it is shown that additional primitives, like the test-for-absence, are needed to get Turing completeness. There, application processes 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.

Our construction has similarities with transformations in [18], where a requirements specification is split in parallel parts communicating via message passing, and [20], where an encoding of choice in the a-synchronous π-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.A. Bergstra and J.W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37(1):77–121, 1985.
2. 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.
3. 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)*, volume 1, pages 149–155. ACM, 2000.
4. M. Boasson. Control systems software. *IEEE Transactions on Automatic Control*, 38(7):1094–1106, July 1993.
5. 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.
6. 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.
7. N. Busi, R. Gorrieri, and G. Zavattaro. On the Turing equivalence of Linda coordination primitives. In *Proceedings of Express '97*, volume 6 of *Electronic Notes in Theoretical Computer Science*, 1997.
8. 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.
9. N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.
10. 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.

11. P.F.G. Dechering and I.A. van Langevelde. The verification of coordination. In A. Porto and Roman C., editors, *Proceedings of the Fourth International Conference on Coordination Models and Languages*, number 1906 in LNCS, Limassol, Cyprus, 2000. Springer-Verlag.
12. W. Fokkink. *Introduction to Process Algebra.* Texts in Theoretical Computer Science (EATCS). Springer-Verlag, 2000.
13. E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces principles, patterns, and practice.* Addison-Wesley, Reading, MA, USA, 1999.
14. 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.
15. J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In C. Verhoef A. Ponse and S.F.M. van Vlijmen, editors, *Algebra of Communicating Processes*, Workshops in Computing, pages 26–62. Springer-Verlag, 1994.
16. J.F. Groote, A. Ponse, and Y.S. Usenko. Linearization in parallel pCRL. Technical Report SEN-R0019, CWI, Amsterdam, The Netherlands, 2000.
17. J.F. Groote and J.S. Springintveld. Focus points and convergent process operators — A proof strategy for protocol verification. Report CS-R9566, CWI, Amsterdam, November 1995.
18. R. Langerak. *Transformations and Semantics for LOTOS.* PhD thesis, Department of Computer Science, University of Twente, November 1992.
19. Nancy Lynch. *Distributed Algorithms.* Morgan Kaufmann, San Francisco, CS, 1996.
20. 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.
21. F.W. Vaandrager. Verification of two communication protocols by means of process algebra. Technical Report CS-R8608, CWI, Amsterdam, 1986.

A. Full Proofs

We will often use a number of basic facts about the graph of Section 4, such as for all edges $\alpha\beta \in E$, we have $\alpha \sqsubset \beta$, $\mu_\alpha \geq 1$, $\nu_\beta < \mu_\alpha$ and $\alpha[\nu_\beta] = \beta$.

A.1 Invariants

The full proof needs a number of invariants of the process P'. Note that in the proof methodology we use, the invariants are motivated by the proof of the matching criteria, which can be computed automatically from the state mapping. With (I)–(IV) we enumerate the summands in P'.

Lemma 14 *The following properties are invariants of P' (for all $\alpha\beta \in E$):*

1. $m_{\alpha\beta} \bmod \mu_\alpha = \nu_\beta$
2. $s_{\alpha\beta} \in \{\ell_0, \ell_1, \ell_2, \ell_3\}$
3. $s_{\alpha\beta} = \ell_3 \to \exists\gamma \sqsupset \alpha.\, (0, \gamma) \in A$
4. $(m_{\alpha\beta}, \alpha) \notin A \Rightarrow s_{\alpha\beta} = \ell_0$

Proof. These four invariants can be proved separately. We have to check that for each summand, $I(d) \wedge b(d, e) \to I(g(d, e))$. We here only give the main argument.

1. $m_{\alpha\beta}$ is only changed by adding μ_α to it.
2. $s_{\alpha\beta}$ is never set to a different value.
3. If $s_{\alpha\beta}$ is set to ℓ_3, then at the same moment $(0, \beta)$ is added to A. Note that for each $\alpha\beta \in E$, $\beta \sqsupset \alpha$. On the other hand, elements of A are never removed.
4. There are two dangerous transitions: (1) If $s_{\alpha\beta}$ goes from ℓ_0 to ℓ_1, but this only happens when $(m_{\alpha\beta}, \alpha)$ is in A. (2) If $m_{\alpha\beta}$ changes, $s_{\alpha\beta}$ is reset to ℓ_0. For all other transitions, it is important that A only grows. ⊠

Lemma 15 *The following properties are invariants of P (for each $\alpha\beta \in E$):*

1. $\forall x.\, (x, \alpha) \in A \to (0, \alpha) \in A$
2. $(0, \alpha) \notin A \Rightarrow m_{\alpha\beta} = \nu_\beta$
3. $\forall x.\, (x, \alpha) \in A \to x \leq m_{\alpha\beta}$
4. *For each γ with $\alpha\gamma \in E$: $(m_{\alpha\beta}, \alpha) \in A \to m_{\alpha\gamma} = m_{\alpha\beta} + (\nu_\gamma - \nu_\beta) \bmod \mu_\alpha$*
5. *for each $\gamma \sqsupseteq \beta$: $(0, \alpha) \in A \wedge (0, \gamma) \in A \to s_{\alpha\beta} = \ell_3$.*

Proof. The first can be proved inductively using one of the invariants of the previous lemma. The second is proved inductively, using the first. The last three are proved by simultaneous induction.

1. A only changes in the last two summands. In the third, $(0, \beta)$ is added, so the property is still true. For the last summand, assume that $\forall x.\, (x, \alpha) \in A \to (0, \alpha) \in A$ (induction hypothesis) and $s_{\alpha\beta} = \ell_1$ (guard). Now $(m_{\alpha\beta} + 1, \alpha)$ is added to A. By Lemma 14.4, $(m_{\alpha\beta}, \alpha) \in A$, so by the induction hypothesis, $(0, \alpha) \in A$.
2. A only grows. In step IV, we may assume that $s_{\alpha\beta} = \ell_1$, then by Lemma 14.4, $(m_{\alpha\beta}, \alpha) \in A$, hence by (1), $(0, \alpha) \in A$, so nothing remains to be proved.

3. A and $m_{\alpha\beta}$ are not changed in the first two summands. The property is invariant for the third summand as $0 \leq m_{\beta\gamma}$ trivially holds. So assume summand IV does a (σ, τ) step; assume the guard: $s_{\sigma\tau} = \ell_1$. The induction hypothesis is:

   $$\forall x.(x, \alpha) \in A \Rightarrow x \leq m_{\alpha\beta} \quad \text{IH (3)}$$

   Assume for arbitrary x, $(x, \alpha) \in A + (m_{\sigma\tau} + 1, \sigma)$.

   - Case $\alpha\beta = \sigma\tau$. In this case we have to prove: $x \leq m_{\sigma\tau} + \mu_\sigma$. Either $(x, \alpha) \in A$, in which case by induction hypothesis, $x \leq m_{\alpha\beta}$, hence $x \leq m_{\alpha\beta} + \mu_\alpha$. Or, $(x, \alpha) = (m_{\sigma\tau} + 1, \sigma)$. In this case, note that $x = m_{\sigma\tau} + 1 \leq m_{\sigma\tau} + \mu_\sigma$.
   - Case $\alpha\beta \neq \sigma\tau$. In this case we have to prove $x \leq m_{\alpha\beta}$. Again, either $(x, \alpha) \in A$, or $(x, \alpha) = (m_{\sigma\tau} + 1, \sigma)$. In the former case, the induction hypothesis applies. In the latter case, first note that $s_{\sigma\tau} = s_{\alpha\tau} = \ell_1$, so by Lemma 14.4, $(m_{\alpha\tau}, \alpha) \in A$, hence by the induction hypothesis (4), $m_{\alpha\beta} = m_{\alpha\tau} + (\nu_\beta - \nu_\tau) \bmod \mu_\alpha$. So we finish the proof by observing that $m_{\alpha\tau} + 1 \leq m_{\alpha\tau} + (\nu_\beta - \nu_\tau) \bmod \mu_\alpha$. (use that $\beta \neq \tau$, hence $\mu_\alpha > 1$).

4. Steps of type I and II don't change A or $m_{\alpha\beta}$.

   - First assume that a (σ, τ)-step of type III occurs, so $s_{\sigma\tau} = \ell_2$. Assume $(m_{\alpha\beta}, \alpha) \in A + (0, \tau)$. Then either $(m_{\alpha\beta}, \alpha) \in A$, in which the induction hypothesis does the job. Or, $(m_{\alpha\beta}, \alpha) = (0, \tau)$. In this case, we must prove: $m_{\tau\gamma} = m_{\tau\beta} + (\nu_\gamma - \nu_\gamma) \bmod \mu_\tau$. By 15.5, as $s_{\sigma\tau} \neq \ell_3$, $(0, \tau) \notin A$, so by (2) (using $\alpha = \tau$), $m_{\alpha\beta} = \nu_\beta$ and $m_{\alpha\gamma} = \nu_\gamma$. Note that $\nu_\beta = 0$, so $\nu_\gamma = \nu_\beta + (\nu_\gamma - \nu_\beta) \bmod \mu_\alpha$.
   - Next assume a (σ, τ) step of type IV. Assume $s_{\sigma\tau} = \ell_1$, then by 14.4, $(m_{\sigma\tau}, \sigma) \in A$. We also assume the induction hypothesis, i.e.

     $$(m_{\alpha\beta}, \alpha) \in A \rightarrow m_{\alpha\gamma} = m_{\alpha\beta} + (\nu_\gamma - \nu_\beta) \bmod \mu_\alpha \quad \text{(for all } \alpha\beta \in E \text{ and } \alpha\gamma \in E)$$

     - Case $\alpha \neq \sigma$. To prove: $(m_{\alpha\beta}, \alpha) \in A + (m_{\sigma\tau} + 1, \sigma) \rightarrow m_{\alpha\gamma} = m_{\alpha\beta} + (\nu_\gamma - \nu_\beta) \bmod \mu_\alpha$. Let $(m_{\alpha\beta}, \alpha) \in A + (m_{\sigma\tau} + 1, \sigma)$, then $m_{\alpha\beta} \in A$; now the induction hypothesis applies.
     - Case $\alpha = \sigma$, $\beta = \tau$. Assume $(m_{\sigma\tau} + \mu_\sigma, \sigma) \in A + (m_{\sigma\tau} + 1, \sigma)$. Then either $(m_{\sigma\tau} + \mu_\sigma, \sigma) \in A$, or $\mu_\sigma = 1$. The former leads to a contradiction: by IH (3), we obtain $(m_{\sigma\tau} + \mu_\sigma \leq m_{\sigma\tau}$, but $\mu_\sigma \geq 1$. Hence $\mu_\sigma = 1$, so $\beta = \gamma = \tau$. In this case we have to prove $m_{\sigma\tau} + \mu_\sigma = m_{\sigma\tau} + \mu_\sigma + (\nu_\tau - \nu_\tau) \bmod \mu_\sigma$, which is obvious.
     - Case $\alpha = \sigma$, $\beta \neq \tau$. Assume $m_{\sigma\beta} \in A + (m_{\sigma\tau} + 1, \sigma)$. Then either $m_{\sigma\beta} \in A$, or $m_{\sigma\beta} = m_{\sigma\tau} + 1$. The first leads to a contradiction: by induction hypothesis, $m_{\sigma\tau} = m_{\sigma\beta} + (\nu_\tau - \nu_\beta) \bmod \mu_\sigma$; similarly, as $m_{\sigma\tau} \in A$, $m_{\sigma\beta} = m_{\sigma\tau} + (\nu_\beta - \nu_\tau) \bmod \mu_\sigma$. From these two we derive $\beta = \tau$, contradicting one of the assumptions. Hence, it must be the case that $m_{\sigma\beta} = m_{\sigma\tau} + 1$. We again distinguish two cases:
       - $*$ case $\gamma = \tau$. Now we have to prove: $m_{\sigma\tau} + \mu_\sigma = \mu_{\sigma\beta} + (\nu_\tau - \nu_\beta) \bmod \mu_\sigma$. By induction hypothesis, we obtain

         $$m_{\sigma\beta} = m_{\sigma\tau} + (\nu_\beta - \nu_\tau) \bmod \mu_\sigma.$$

         Using the law of modulo arithmetic $(x - y) \bmod z = z - (y - x) \bmod z$, we obtain:

         $$m_{\sigma\beta} = m_{\sigma\tau} + \mu_\sigma - (\nu_\tau - \nu_\beta) \bmod \mu_\sigma,$$

         and we are done.

- case $\gamma \neq \tau$. In this case we must prove: $m_{\sigma\gamma} = m_{\sigma\beta} + (\nu_\gamma - \nu_\beta) \bmod \mu_\sigma$. We now use the induction hypothesis twice (for β and γ):

$$\begin{aligned} m_{\sigma\gamma} &= m_{\sigma\tau} + (\nu_\gamma - \nu_\tau) \bmod \mu_\sigma \\ m_{\sigma\beta} &= m_{\sigma\tau} + (\nu_\beta - \nu_\tau) \bmod \mu_\sigma \end{aligned}$$

Subtraction leads to the desired result:

$$\begin{aligned} m_{\sigma\gamma} - m_{\sigma\beta} &= (\nu_\gamma - \nu_\tau) \bmod \mu_\sigma - (\nu_\beta - \nu_\tau) \bmod \mu_\sigma \\ &= (\nu_\gamma - \nu_\beta) \bmod \mu_\sigma \end{aligned}$$

5. Let $\alpha\beta \in E$ and $\gamma \sqsupseteq \beta$ be given. Assume the induction hypothesis: $(0,\alpha) \in A$ and $(0,\gamma) \in A$ implies $s_{\alpha\beta} = \ell_3$. In summand I and II, A is not changed, and in summand IV no pair of the form $(0,x)$ is added to A, so in these cases the induction hypothesis immediately applies and yields that $s_{\alpha\beta} = \ell_3$. Two cases can be distinguished:

   - $\alpha\beta = \sigma\tau$: Then $s_{\sigma\tau} = \ell_3$, which contradicts the fact that we took summand I, II or IV.
   - $\alpha\beta \neq \sigma\tau$: then $s_{\alpha\beta}$ is unchanged, and remains ℓ_3 in the transition.

   Next, assume that a (σ,τ)-step of type III is taken. We can assume the guard: $s_{\sigma\tau} = \ell_2$. Furthermore, assume that $(0,\alpha) \in A + (0,\tau)$ and $(0,\gamma) \in A + (0,\tau)$. We distinguish cases:

   - Case $(0,\alpha) \in A$ and $(0,\gamma) \in A$. This is similar to steps I,II and IV.
   - Case $(0,\alpha) \in A$ and $\gamma = \tau$: Notice that either $\beta \sqsubseteq \sigma$, or $a = \sigma$. If $a = \sigma$ we have to prove $\ell_3 = \ell_3$. Otherwise, if $\beta \sqsubseteq \sigma$, we must prove that $s_{\alpha\beta} = \ell_3$. Note that via previously proved invariants we obtain $(0,\sigma) \in A$, so by the induction hypothesis, $s_{\alpha\beta} = \ell_3$.
   - Case $a = \tau$ and $(0,\gamma) \in A$. Then $s_{\sigma\alpha} = \ell_2$, hence by previous invariants $(0,\sigma) \in A$, so by the induction hypothesis, $s_{\sigma\alpha} = \ell_3$, contradiction.
   - $\alpha = \tau$ and $\gamma = \tau$. This is impossible, because $\alpha \sqsupset \gamma$, and E is supposed to be a-cyclic. $\boxtimes$

The following invariant reuses notation of Section 4.3. In particular, let $\vec{d} = (A, \ldots, m_{\alpha\beta}, s_{\alpha\beta}, \ldots)$. Furthermore, let $(\sigma, p, \tau) = H(A)$.

Lemma 16 *The following is an invariant of P': $\tau = \bot$ or $p = m_{\sigma\tau}$.*

Proof. $s_{\alpha\beta} \neq \ell_0$, hence by 14.4, $(m_{\alpha\beta}, \alpha) \in A$. Hence by 15.1, also $(0,\alpha) \in A$. As $s_{\alpha\beta} \neq \ell_3$, by 15.5 we know that there exists no $x \sqsupset \alpha$ with $(0,x) \in A$. Hence $\sigma = \alpha$. By 15.3, there is no $y > m_{\alpha\beta}$ such that $(y,\alpha) \in A$. Hence $p = m_{\alpha\beta}$. Now using 14.1, $\tau = \alpha[m_{\alpha\beta} \bmod \mu_\alpha] = \alpha[\nu_\beta] = \beta$. $\boxtimes$

We now prove that all invariants hold in the initial state of P'. So let $\vec{d} = (\{(0,\omega)\}, \ldots, \nu_\beta, \ell_0, \ldots)$ be the initial state of P'.

Lemma 17 *All invariants of Lemma 14, 15 and 16 hold on $\vec{d}$.*

Proof.

Invariant of Lemma 14: Let $\alpha\beta \in E$.

1. $\nu_\beta \bmod \mu_\alpha = \nu_\beta$, because $\nu_\beta < \mu_\alpha$.
2. $\ell_0 \in \{\ell_0, \ell_1, \ell_2, \ell_3\}$.
3. This holds as $s_{\alpha\beta} = \ell_0 \neq \ell_3$.
4. This holds because $s_{\alpha\beta} = \ell_0$.

Invariant of Lemma 15: Let $\alpha\beta \in E$.

1. If $(x, \alpha) \in A$, then $x = 0$ because $(0, \omega)$ is the only element in the initial state, hence $(0, \alpha) \in A$.
2. This holds because initially $m_{\alpha\beta} = \nu_\beta$.
3. If $(x, \alpha) \in A$, then $x = 0$ hence $x \leq m_{\alpha\beta}$.
4. If $(m_{\alpha\beta}, \alpha) \in A$, then $m_{\alpha\beta} = 0$, so $\nu_\beta = 0$. Let $\alpha\gamma \in E$, then indeed $\nu_\gamma = m_{\alpha\beta} + (\nu_\gamma - \nu_\beta)$.
5. For no $\gamma \sqsupseteq \beta \sqsupset \alpha$, $(0, \gamma) \in A$, so this item is vacuously true.

Invariant of Lemma 16: Note that $H(\vec{d}) = (\sigma, p, \tau)$. So $\sigma = \omega$ and $p = 0$. If $\mu_0 = 0$, then $\tau = \perp$ (finished). Otherwise, $\tau = \omega[0]$, and $m_{\sigma\tau} = 0 = p$. ⊠

A.2 Matching Criteria

Next, we have to compute and prove the matching criteria, as introduced in Section 2. Here P' plays the role of the implementation, and Y serves as the specification. We write $\vec{d}$ for the state vector of P': $(A, \ldots, m_{\alpha\beta}, s_{\alpha\beta}, \ldots)$. We reuse the abbreviations introduced for the state mapping in Section 4.3. Let $(\sigma, p, \tau) = H(A)$ and $(a, m) = h(\vec{d})$, where h is the state mapping. Then the matching criteria (for any $\alpha\beta \in E$) are:

1. (a) $(m_{\alpha\beta}, \alpha) \in A \wedge s_{\alpha\beta} = \ell_0 \Rightarrow h(\vec{d}) = h(\vec{d}[s_{\alpha\beta} := \ell_1])$
   (b) $s_{\alpha\beta} = \ell_2 \Rightarrow h(\vec{d}) = h(\vec{d}[A := A + (0, \beta), s_{\alpha\beta} := \ell_3])$
2. If $s_{\alpha\beta} = \ell_1$ then $a = \alpha$ and $m = \nu_\beta$.
3. If FC, $a = \alpha$ and $\nu_\beta = m$ then $s_{\alpha\beta} = \ell_1$.
5. (a) If $s_{\alpha\beta} = \ell_1$ then $h(\vec{d}[s_{\alpha\beta} := \ell_2]) = h(\vec{d})[a, m := \beta, 0]$.
   (b) If $s_{\alpha\beta} = \ell_1$ then $h(\vec{d}[A := A + (m_{\alpha\beta} + 1, \alpha), s_{\alpha\beta := \ell_0}]) = h(\vec{d})[m := (m + 1) \bmod \mu_\alpha]$.

In order to prove these criteria, we need an auxiliary lemma:

Lemma 18 *Let $(\alpha\beta) \in E$, $\vec{d} = (A, \ldots, m_{\alpha\beta}, s_{\alpha\beta}, \ldots)$. Define $(\sigma, p, \tau) = H(A)$. If $s_{\alpha\beta} \in \{\ell_1, \ell_2\}$ then $(\sigma, p, \tau) = (\alpha, m_{\alpha\beta}, \beta)$*

Proof. $s_{\alpha\beta} \neq \ell_0$, hence by 14.4, $(m_{\alpha\beta}, \alpha) \in A$. Hence by 15.1, also $(0, \alpha) \in A$. As $s_{\alpha\beta} \neq \ell_3$, by 15.5 we know that there exists no $x \sqsupset \alpha$ with $(0, x) \in A$. Hence $\sigma = \alpha$. By 15.3, there is no $y > m_{\alpha\beta}$ such that $(y, \alpha) \in A$. Hence $p = m_{\alpha\beta}$. Now using 14.1, $\tau = \alpha[m_{\alpha\beta} \bmod \mu_\alpha] = \alpha[\nu_\beta] = \beta$. ⊠

Lemma 19 *Given the invariants of 14, 15 and 16, the matching criteria hold.*

Proof.

1. (a) A and $m_{\alpha\beta}$ don't change, and $s_{\alpha\beta} \neq \ell_2$ before and after the transition, hence none of σ, p, τ, a, m change.
   (b) Assume $s_{\alpha\beta} = \ell_2$. Then by 18, $(\sigma, p, \tau) = (\alpha, m_{\alpha\beta}, \beta)$. Hence $h(\vec{d}) = (\beta, 0)$. Let $\vec{d'} = d[A := A + (0, \beta); s_{\alpha\beta} := \ell_3]$. Define $(\sigma', p', \tau') = H(\vec{d'})$. Then $\sigma' = \beta$, $p' = 0$. Now distinguish cases on $\mu_\alpha > 1$:
   - $\tau' = \perp$. Then $h(\vec{d'}) = (\beta, 0) = h(\vec{d})$.
   - $\tau' = \beta[0]$. As $(0, \beta) \notin A$ (α was maximal), $s_{\beta\tau'} = \ell_0$ by 14.4, so $h(\vec{d'}) = (\beta, 0) = h(\vec{d})$.
2. Let $s_{\alpha\beta} = \ell_1$. Then by 18, $(\sigma, p, \tau) = (\alpha, m_{\alpha\beta}, \beta)$. Then $a = \sigma = \alpha$, and $m = m_{\alpha\beta} \bmod \mu_\alpha = \nu_\beta$ by 14.1.

3. Define the focus condition $FC = \forall \alpha, \beta : (s_{\alpha\beta} \neq \ell_0 \vee (m_{\alpha\beta}, \alpha) \notin A) \wedge s_{\alpha\beta} \neq \ell_2$. Assume FC, $a = \alpha$ and $\nu_\beta = m$. By FC, $s_{\alpha\beta} \neq \ell_2$, hence by definition, $\alpha = \sigma$ and $m = p \bmod \mu_\sigma$. Then $\tau = \alpha[m] = \alpha[\nu_\beta] = \beta$. If $s_{\alpha\beta} = \ell_3$, by 14.3, $(0, \beta) \in A$, but this contradicts the fact that σ is the maximal such node by definition. Hence $s_{\alpha\beta} \neq \ell_3$. Now assume $s_{\alpha\beta} = \ell_0$. Then $(m_{\alpha\beta}, \alpha) \notin A$, but by Lemma 16, $(m_{\sigma\tau}, \sigma) = (p, \sigma)$, which is in A by definition of p. Hence, using 14.2, $s_{\alpha\beta} = \ell_1$.

5. (a) Let $s_{\alpha\beta} = \ell_1$. Then by 18, $(\sigma, p, \tau) = (\alpha, m_{\alpha\beta}, \beta)$. Hence $h(\vec{d}[s_{\alpha\beta} := \ell_2]) = (\beta, 0) = h(\vec{d})[a, m := \beta, 0]$.

   (b) Let $s_{\alpha\beta} = \ell_1$, then by 18, $(\sigma, p, \tau) = (\alpha, m_{\alpha\beta}, \beta)$. Let $\vec{d'} = \vec{d}[A := A + (m_{\alpha\beta} + 1, \alpha), m_{\alpha\beta} := m_{\alpha\beta} + \mu_\alpha, s_{\alpha\beta} := \ell_0]$. Let $H(\vec{d'}) = (\sigma', p', \tau')$. Then $\sigma' = \alpha$, $p' = m_{\alpha\beta} + 1$ and $\tau' = \alpha[(m_{\alpha\beta} + 1) \bmod \mu_\alpha]$. We distinguish cases:

   - $\mu_\alpha = 1$. Then $h(\vec{d'}) = (\alpha, 0) = h(\vec{d})[m := (m+1) \bmod \mu_\alpha]$.
   - $\mu_\alpha > 1$. As $(m_{\alpha\beta}, \alpha) \in A$ by 14.4, Lemma 15.4 can be applied, yielding $m_{\alpha\tau'} = m_{\alpha\beta} + (\nu'_\tau - \nu_\beta) \bmod \mu_\alpha = m_{\alpha\beta} + 1$. Hence $m_{\alpha\tau'} > p$, so by maximality of p, $(m_{\alpha\tau'}, \alpha) \notin A$, hence by 14.4, $s_{\alpha\tau'} = \ell_0$. But then we may conclude: $h(\vec{d'}) = (\sigma', p' \bmod \mu_{\sigma'}) = (\alpha, (m_{\alpha\beta} + 1) \bmod \mu_\alpha) = (\alpha, (m+1) \bmod \mu_\alpha) = h(\vec{d})[m := (m+1) \bmod \mu_\alpha]$. ⊠

A.3 Proofs of Other Equivalences

Proof of Proposition 8. Note that P' is convergent, because in each τ-step, the number $\#\{\alpha\beta \mid s_{\alpha\beta} \in \{\ell_0, \ell_2\}\}$ decreases. As a consequence, CL-RSP can be used on P', which states that convergent LPEs have a unique solution.

Consider the following generalized renaming (acting on actions rather than action labels):

$$\rho = \left\{ \begin{array}{rcl} R(m, a) & \mapsto & \tau \\ W(0, \alpha) & \mapsto & \tau \\ W(m+1, \alpha) & \mapsto & i \end{array} \right.$$

Observe that $\rho(P)$ is a solution for P' (this is checked by substituting $\rho(P)$ in the defining equation for P' and using the fact that renamings distribute over $+$ and $\cdot$). Next, by CL-RSP, $\rho(P) = P'$. Now consider the renaming $\rho' : i \mapsto \tau$. Now using a generalized alphabet law,

$$\tau_{\{i\}}(P') = \tau_{\{i\}}(\rho(P)) = \rho'(\rho(P)) = (\rho' \circ \rho)(P) = \tau_{\{R,W\}}(P)$$

⊠

Proof of Proposition 10. We proved that the properties of Lemma 14, 15 and 16 are invariant, and that these invariants hold in the initial state (17). Note that P' can do a τ-step in the initial state, so the focus condition doesn't hold. From the cones and focus theorem, we obtain for any $\vec{d}$ not satisfying the focus condition: $\tau.Y(h(d)) = \tau.P'(d)$. Note that $H(\{(0, \omega)\}, \ldots, \nu_\beta, \ell_0, \ldots) = (\omega, 0, \omega[0])$, hence $h(\{(0, \omega)\}, \ldots, \nu_\beta, \ell_0, \ldots) = (\omega, 0)$. Hence indeed, $\tau.P'(\{(0, \omega)\}, \ldots, \nu_\beta, \ell_0, \ldots) = \tau.Y(\omega, 0)$. ⊠

Proof of Proposition 13.

$$\begin{aligned}
Z(a') &= \sum_{\alpha\beta\in E}\sum_{a,m}\lambda_{\alpha\beta}.Z(\beta) \triangleleft a=\alpha \wedge \nu_\beta = m \wedge a'=a \triangleright \delta \\
&\quad + \sum_{\alpha\beta\in E}\sum_{a,m}\tau.Z(a) \triangleleft F \triangleright \delta \\
&= \sum_{\alpha\beta\in E}\sum_{a,m}\lambda_{\alpha\beta}.Z(\beta) \triangleleft a=\alpha \wedge \nu_\beta = m \wedge a'=a \triangleright \delta \\
&= \sum_{\alpha\beta\in E}\lambda_{\alpha\beta}.Z(\beta) \triangleleft a'=\alpha \triangleright \delta \\
&= X(a')
\end{aligned}$$

In the first equality we used some data identities; then we used some basic laws, like $x \triangleleft F \triangleright y = y$, $x+\delta = x$ and $\sum\delta = \delta$. In the third step we have used the sum-elimination theorem: $\sum_{d:D} p(d) \triangleleft d = e \wedge b(d) \triangleright \delta = p(e) \triangleleft b(e) \triangleright \delta$. In the last step we use RSP, which says that the defining equation for X has a unique solution. ⊠

Proof of Main Theorem 5. The proof concatenates the results of Proposition 7, 13, 12, 10, 8 and 6. We also use that $\tau.\tau_I(x) = \tau_I(\tau.x)$ and some basic reasoning about the alphabet of processes. Let ω be the root symbol in V.

$$\begin{aligned}
\tau.X.\delta &= \quad \{\text{linearization: Proposition 7}\} \\
&\tau.X(\omega) \\
&= \quad \{\text{RSP: Proposition 13}\} \\
&\tau.Z(\omega) \\
&= \quad \{\text{L-CFAR: Proposition 12}\} \\
&\tau.\tau_{\{i\}}(Y(\omega,0)) \\
&= \quad \{\text{state mapping: Proposition 10}\} \\
&\tau.\tau_{\{i\}}(P'(\{(0,\omega)\},\ldots,\nu_\beta,\ell_0,\ldots)) \\
&= \quad \{\text{alphabet: Proposition 8}\} \\
&\tau.\tau_{\{R,W\}}(P(\{(0,\omega)\},\ldots,\nu_\beta,\ell_0,\ldots)) \\
&= \quad \{\text{linearization: Proposition 6}\} \\
&\tau.\tau_{\{R,W\}}(\partial_{\{Read,Write,read,write\}}(S(\{(0,\omega)\}) \,||\, (\mathop{||}_{\alpha\beta\in E} P_{\alpha\beta}(\nu_\beta))))
\end{aligned}$$

⊠