Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

Miniford: a Kernel for a Manifold-like Coordination Language

E.P.B.M. Rutten

Computer Science/Department of Interactive Systems

CS-R9252 1992

MINIFOLD: a Kernel for a MANIFOLD-like Coordination Language

E.P.B.M. RUTTEN

CWI
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

Abstract

MINIFOLD is a kernel for a coordination language, following the MANIFOLD model. This model focuses on the coordination of processes, separated from their computation functionality. Processes are considered as black boxes, and their behavior is abstracted to their communications.

MINIFOLD provides constructs to build up an environment of concurrent processes and to manage the communication between them. On the one hand, a data-flow mechanism can be used to build networks of streams, linking input and output ports of the processes, and carrying the units exchanged between them. On the other hand, an event broadcasting mechanism provides control on the dynamical modification of the data-flow network. MINIFOLD is introduced in a constructive and incremental way. It is provided with an operational semantics, a model of its execution based on automata is proposed, and illustrated by simple classical example.

The purpose of the study of this very simplified instance of the MANIFOLD concept is to explore models of its behavior, and to give a formalization of its bare essentials. It is intended that the MANIFOLD language can take advantage of this, as guidelines for formalisms underlying practical tools for program analysis, clarification of its structure and as a basis for the comparison with other models.

1991 Mathematics Subject Classification: *68N15 [Software]: Programming Languages; 68Q05, 68U30.*

1991 CR Categories: *C.1.2, C.1.3, C.2.m, D.1.3, D.3.1, F.1.2, I.1.3.*

Key Words and Phrases *Formal specifications, parallel computing, models of computation, programming language semantics, coordination languages.*

Note *Author's present address: IRISA/INRIA, F-35042 Rennes, France; rutten@irisa.fr*

Contents

1 Introduction **3**

2 MINIFOLD: a kernel of MANIFOLD **4**
2.1 Atomic processes 4
2.2 Data-flow connections 6
2.2.1 Streams: connecting ports of processes 6
2.2.2 Data-flow networks: sets of streams 7
2.3 Event-driven state change of the network 9
2.3.1 States: associating networks and events 9
2.3.2 Coordinator: set of states 9
2.4 Single coordinator applications 10
2.4.1 Applications 10
2.4.2 A first complete example 10
2.5 Concurrent coordinators applications 11
2.5.1 Applications 11
2.5.2 An example with concurrent coordinators 12
2.6 MINIFOLD and its complete grammar 13

3 Formal model of MINIFOLD **13**
3.1 States of an application 14
3.2 Construction of the state of a program 16
3.3 Transitions 17
3.3.1 Unit exchange between ports and streams 18
3.3.2 Event exchange between atomic and coordinator processes 19
3.3.3 Termination of processes and networks 21
3.3.4 Relation between event level and unit level 24
3.4 An alternative model: automaton of an application 24
3.4.1 Automaton of a coordinator 25
3.4.2 Operations for combining interacting automata 27
3.4.3 Automaton of an application 29

4 Examples **31**
4.1 The Fibonacci series 31
4.2 The sieves of Eratosthenes 32
4.2.1 Arrays of processes 33
4.2.2 The example of the sieves of Eratosthenes 34

5 Miscellaneous ideas and open problems **35**

6 Conclusion **37**

1 Introduction

We present MINIFOLD, a kernel for a coordination language, following the MANIFOLD model [1]. As such, MINIFOLD can also be seen as an abstraction of the MANIFOLD parallel programming language. The focus of this language is on the coordination of processes, and on their communication; it is *not* on the computations performed by some of the processes. These latter are considered as black boxes, the behavior of which is abstracted to their input and output. This work is in the area of coordination languages [7], of which LINDA can be seen as a different instance [6]. Communication is supported by two mechanisms: data-flow streams, and event broadcasting. Thus it is an approach to data-flow languages [8], originated from and motivated by practical problems in dataflow hardware realization, rather than theoretical considerations. MANIFOLD is a practical and experimental language, defined in a detailed informal specification [1], and for which an implementation is being finalized [2].

A formal specification is given of a sub-language [10], in the form of an operational semantics focusing on the transitions of the event-driven mechanism and the representation of connection states of the data-flow networks. It is intended to clarify formally the structures and behaviors of the model, while keeping most of the programming language, and has been implemented in the ASF+SDF environment [11]. The complexity of the result was mainly due to the representation of features that are not of a primary significance. Hence the need for a still more abstract model.

In this paper, we perform a reconstruction of only very essential features of MANIFOLD, into the kernel language MINIFOLD. However, fundamental assumptions of the model are kept: data-flow and event communication co-exist, some processes are there to coordinate the communication between others, while atomic processes are those not decomposable as a coordinator process, and are the only ones responsible for the computations.

An important difference between MINIFOLD and MANIFOLD is a simplification concerning the event communication. The event mechanism of MINIFOLD is deterministic, and leads to simpler models in terms of states and transitions. The goal of this work is to propose formal models for some of the concepts of MANIFOLD-like languages: in order to keep a clear understanding of the behavior of these models, it is preferable to keep them small, which is a motivation for the simplification. As such, this model does not meet the choices of full asynchrony and non-determinism made for MANIFOLD, which correspond to its practical and real-world motivations. But it constitutes a set of clarified concepts, and a base for possible extensions in these direction.

MINIFOLD provides constructs to build up an environment of concurrent processes and to manage the communications between them. On the one hand, a data-flow mechanism allows to build networks of streams, linking input and output ports of the processes, and carrying the units exchanged between them. On the other hand, an event broadcasting mechanism provides control on the dynamical modification of the data-flow network.

In the following section, we introduce one by one the basic features of MINIFOLD, illustrating them with examples, and explicitly describing their inter-relations: we introduce atomic processes, streams connecting these processes, data-flow networks made of several of

these streams, states associating such a network with an event, coordinators made of a set of such states, and applications grouping concurrent coordinators. In section 3, we give a formal model of the MINIFOLD language and its constructs, and rules describing the possible transitions between the states of an application. We also describe MINIFOLD applications in terms of finite state deterministic automata. In section 4, the two classical examples of Fibonacci and prime numbers by Eratosthenes method are given in MINIFOLD. We discuss open issues in section 5, and conclude in section 6.

2 MINIFOLD: a kernel of MANIFOLD

MINIFOLD is defined as a configuration language where *atomic processes*, characterized only by their input and output, are connected through *streams* attached to their *ports*. The streams together form a data-flow *network*, and a change of *state* of the dynamic data-flow network is made by a *coordinator process* on the reception of an *event occurrence*, raised by one of the atomic processes[1] . The atomic processes, the coordinators, the streams, and the ports constitute the environment in which events are broadcasted.

2.1 Atomic processes

Atomic processes are external, and *atomic* in the sense that they are considered as *black boxes*, of which no internal feature or behavior is known. This is justified by the fact that MINIFOLD is a *configuration* language, meant to manage the communication between processes, but not the computations performed inside them. Thus, at the level of MINIFOLD, they cannot be decomposed further than their input and output channels, hence they are said to be atomic.

The atomic processes communicate only using *units* (input in or output from *ports*, where the connections will be attached), and *events* (raised and broadcast in the surrounding environment). An atomic process can perform the following actions:

- it can *raise an event*,
- one of its input ports can *take a unit in* from *a* stream[2] to which it is connected,
- one of its output ports can *put a unit out* to *all* the streams[3] to which it is connected.

Seen from MINIFOLD, atomic processes can terminate on their own, without condition; their ports are then not accessible anymore, and their events cannot be raised.

The syntax for the definition of such a process is as follows:

[1] In MINIFOLD, coordinator processes are not given the possibility of raising events.

[2] A port might be attached to several streams, but it accepts units form only one of them at a time, *merging* them non-deterministically.

[3] This means that each unit put out is *duplicated* for each of the streams.

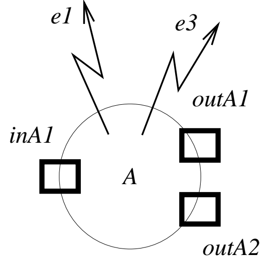

Figure 1: An atomic process.

$$\begin{array}{rcl}\langle atomic \rangle & ::= & \texttt{atomic}\ \langle process \rangle\ \langle ports\ in \rangle\ \langle ports\ out \rangle\ \langle events \rangle \\ \\ \langle ports\ in \rangle & ::= & \texttt{in}\ \langle port \rangle\ [\ \texttt{,}\ \langle port \rangle]^*\ \mid \varepsilon \\ \langle ports\ out \rangle & ::= & \texttt{out}\ \langle port \rangle\ [\ \texttt{,}\ \langle port \rangle]^*\ \mid \varepsilon \\ \langle events \rangle & ::= & \texttt{event}\ \langle event \rangle\ [\ \texttt{,}\ \langle event \rangle]^*\ \mid \varepsilon \end{array}$$

The names ⟨*process*⟩ of the process, ⟨*port*⟩ of ports in the lists, and ⟨*event*⟩ of the events in the list, are identifiers. The empty word is designated by ε.

From outside the atomic processes, at the global level of an application considered further, the name of the process will be used to build absolute names of its ports, in the form of a composition using the dot: "`.`".

A port ⟨*port*⟩ of a process ⟨*process*⟩ will have the absolute name:

$$\langle port\ name \rangle ::= \langle process \rangle . \langle port \rangle$$

In the same way, event *occurrences*, can be given global names, by mentioning their *source* i.e., the port or process that raised them; an ⟨*event*⟩ raised by a process ⟨*process*⟩ will have the absolute name:

$$\langle event\text{-}occ \rangle \quad ::= \quad \langle event \rangle . \langle process \rangle$$

An example of atomic process is the process `A`, with one input port named `inA1` and two output ports named `outA1` and `outA2`. It can raise the events `e1` and `e3`. This process is defined in the statement:

```
atomic A  in inA1
          out outA1 , outA2
          event e1 , e3
```

This example is illustrated graphically in fig. 1.

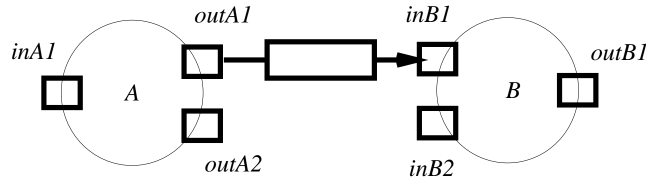

Figure 2: The stream: `A.outA1 -> B.inB1` .

2.2 Data-flow connections

2.2.1 Streams: connecting ports of processes

The streams are connections between ports of processes. They are attached to two ports: one *source* port (which is an output port of its owner process) and one *sink* port (which is an input port of its owner process).

Streams carry *units* unidirectionally, from the source to the sink port. They behave like a *first-in first-out* link, without loss of units. There is no assumption whatsoever about the contents or meaning of units: this is left to computations in atomic processes.

A stream can perform the following actions:

- *take a unit in* from its source port,
- *put a unit out* to its sink port.

The effect of a process disappearing is that all the streams involving one of its ports are terminated i.e., broken.

The syntax to denote a stream between two ports is as follows:

$$\langle stream \rangle ::= \langle port\ name \rangle \texttt{ -> } \langle port\ name \rangle$$

where the left $\langle port\ name \rangle$ is the source, and the one on the right is the sink.

An example of stream is to link the processes `A`, defined previously, and `B`, defined by:

```
atomic B in inB1, inB2 out outB1 event e2, e3
```

with a stream going from the port `outA1` of process `A` to the port `inB1` of process `B`, with the statement:

```
A.outA1 -> B.inB1
```

as illustrated graphically in fig. 2.

2.2.2 Data-flow networks: sets of streams

A set of streams between processes defines a communications *network*. In a network, all member streams are simply acting concurrently. A network of which all streams are broken is broken also. In cases where only some of several streams in a network are broken, there are two ways of grouping streams into sub-networks:

- *pipe-lines* are sets of streams such that a pipe-line breaks if *at least one* of its members breaks;
- *groups* are sets of pipe-lines such that a group breaks if *all* of its members break.

For groups we use an addition-like notation `+` i.e., for two streams s_1 and s_2, s_1 `+` s_2. For pipe-lines, we use a multiplication-like $*$ i.e., for two streams s_1 and s_2, $s_1 * s_2$[4].

The intuitive reason for the choice of these notations is simply that if a broken stream is interpreted as 0, then[5]:

- a group s `+` $0 = s$, which can be interpreted as: one of the members of the group disappears, but others continue to exist.
- a pipe-line $s * 0 = 0$: when one member of a pipe-line breaks, then the whole pipe-line is broken.

The pipe-line operator has a higher priority than the group operator: n_1 `+` $n_2 * n_3$ means n_1 `+` $(n_2 * n_3)$. Such an expression is to be interpreted in the context of an application, and a stream $P.p$ `->` $P'.p'$ is 0 if either process P, or process P' is terminated. Rules for evaluating such networks are: n `+` $0 = 0$ `+` $n = n$ and $n * 0 = 0 * n = 0$.

The syntax for networks defines them as groups of pipe-lines of streams:

$$\begin{array}{lcl} \langle \mathit{network} \rangle & ::= & \langle \mathit{pipe\text{-}line} \rangle \ \texttt{+} \ \langle \mathit{network} \rangle \mid \langle \mathit{pipe\text{-}line} \rangle \\ \langle \mathit{pipe\text{-}line} \rangle & ::= & \langle \mathit{stream} \rangle * \langle \mathit{pipe\text{-}line} \rangle \mid \langle \mathit{stream} \rangle \end{array}$$

In the case where several streams share the same port as source, as in the network: p `->` p' `+` p `->` p'', illustrated in fig. 3 (a), the units are duplicated to all the streams at ounce. In the case where several streams share the same port as a sink, as in the network: p' `->` p `+` p'' `->` p, illustrated in fig. 3 (b), their outcoming units are accepted by the port and merged in a non-deterministic order.

We illustrate this in our example, extended with a third process `C` defined by:

```
atomic C in inC out outC event e3
```

A network between these three processes is:

```
A.outA1 -> B.inB1 + C.outC -> A.inA1 + B.outB1 -> C.inC
```

as illustrated in fig. 4.

[4] Other notations could be chosen, like different styles of parentheses, as in MANIFOLD [10, 11]: `(...)` and `[...]`.

[5] Another way of noting this is to say that, if a breaking stream is interpreted as *false*, then noting pipe-lines with a conjunction $\wedge$ and groups with a disjunction $\vee$ means that: the pipe-line $s \wedge false$ is $false$ i.e., terminates; the group $s \vee false$ is s i.e., behaves like s.

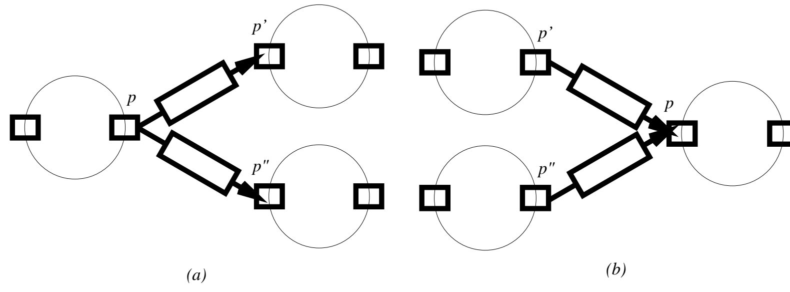

Figure 3: The networks: (a): p`->`p' `+` p`->`p'', (b): p'`->`p `+` p''`->`p.

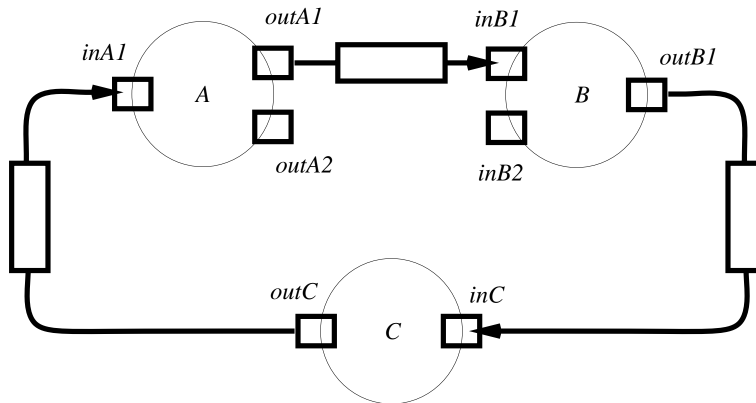

Figure 4: A network.

2.3 Event-driven state change of the network

2.3.1 States: associating networks and events

The MINIFOLD language is about changing states of a data-flow network, in reaction to an occurrence of an event raised by a source.

A new construct of the language associates a data-flow network expression with one or several event occurrences, forming a label. It is called a *state*, and has the following syntax:

$$\begin{aligned} \langle \textit{state} \rangle &::= \langle \textit{label} \rangle \texttt{:} \quad \langle \textit{network} \rangle \ \texttt{.} \\ \langle \textit{label} \rangle &::= \langle \textit{event-occ} \rangle \ [\ \texttt{,}\ \langle \textit{event-occ} \rangle\ ]^* \end{aligned}$$

Here is its intuitive semantics: when the event is raised by an atomic process, the previous state of the network is preempted, its corresponding network is dismantled, and the new state is the network associated with the label featuring the event occurrence.

The termination of a state means that this state can not be reached anymore, and its network can not be installed. This can be defined from its network, and from the event occurrences in its label. A state terminates when one of the two following is verified:

- the network is broken;
- all sources of event occurrences in the label of the state are terminated processes; this means that the state is not reachable anymore, even if the network is unbroken.

As long as it is not terminated, a state is maintained up to date during execution of an application, meaning that events from terminated sources are removed from its label, and broken pipe-lines are removed from the network.

For example, the network of the previous example, illustrated in fig. 4, can be associated with the event `e1`, raised by process `A`, into the state:

```
e1.A : A.outA1 -> B.inB1 + C.outC -> A.inA1 + B.outB1 -> C.inC
```

A coordinator goes into that state when reacting to the raising of event `e1` by process `A`.

2.3.2 Coordinator: set of states

A *coordinator* is a process, that might have input and output ports itself, and consists of a collection of states. It coordinates the communications between processes, by installing different configurations of the data-flow network. It is defined following the syntax:

$$\begin{aligned} \langle \textit{coordinator} \rangle ::= \ & \texttt{coordinator} \ \langle \textit{process} \rangle \\ & \langle \textit{ports in} \rangle \\ & \langle \textit{ports out} \rangle \\ & \{ \ \langle \textit{state} \rangle^{+} \ \} \end{aligned}$$

Its execution consists of the transition to the corresponding state on reception of an event raised by a process. The states in the body are required to correspond to events in an exclusive manner, in order to keep the transition to the next state deterministic. In other terms, the labels have to be disjoint.

In each of these states, the network is built between processes declared in the declarations of the application. A coordinator can access its own ports in a way different from the others' ports: it can use its own input ports as sources in streams, and its output ports as sinks. These ports of the coordinator processes give a means of structuring the network by encapsulating sub-networks: in the case of concurrency introduced further, connections to the coordinator's ports can be managed by other processes without taking into account how the coordinator manages its sub-network. This point is illustrated in the example of section 2.4.2, in figs. 5 and 6, where the ports `input` and `output` of the coordinator are used this way.

A coordinator terminates when it has nothing left to coordinate i.e., when all its states are terminated. This means in particular that the coordinator has nothing to install anymore; for a coordinator having k states $l_i : n_i.$, $i \in [1..k]$, this could be written with the `+` notation: $\sum_{i\in[1..k]} n_i = 0$. During execution, the set of states is updated by removing terminated states.

2.4 Single coordinator applications

2.4.1 Applications

An application is made of at least one coordinator process and atomic processes. Its behavior consists of reacting to events raised by the atomic processes by changing the state of the data-flow network, following the coordinator.

All processes are started together at the start of the application, and the coordinator has an initial state featuring an empty network.

Each terminating process disappears from the application. An application terminates when the coordinator is terminated. If there are remaining unterminated atomic processes, they are not coordinated any more, thus they do not comprise an application: therefore the application termination forces their termination.

We can introduce a first approximation of MINIFOLD with a single coordinator. Its syntax is:

$$\langle application \rangle ::= \langle atomic \rangle^+ \ \langle coordinator \rangle$$

We can now give a first complete example, representing the language in one of its simplest expressions.

2.4.2 A first complete example

A simple but complete example, featuring the previous partial examples, is shown in table 1.

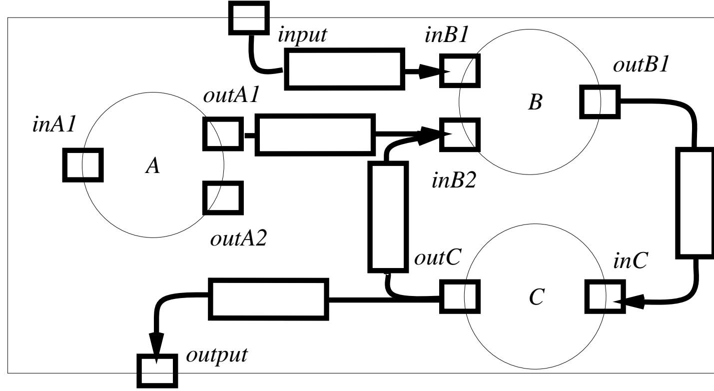

Figure 5: Single-coordinator example: the state s_2.

The application is in the state illustrated in fig. 4, and that we will call s_1, if the event `e1` is raised by process `A`:

```
e1.A : A.outA1->B.inB1 + C.outC->A.inA1 + B.outB1->C.inC .
```

If the event `e2` is raised by `B`, the previous state of the network is dismantled, and instead the new state $s_2$is installed, as shown in fig. 5:

```
e2.B : A.outA1->B.inB2 + C.outC->B.inB2
                  + B.outB1->C.inC + main.input->B.inB1
                                          + C.outC-> main.output .
```

If the event `e3` is raised, by process `A` or `C`, the previous state of the network is dismantled, and instead the new state s_3 is installed, as shown in fig. 6:

```
e3.A , e3.C : A.outA1->B.inB1 + C.outC->A.inA1
                  + A.outA2->C.inC + main.input->A.inA1
                                          + B.outB1-> main.output .
```

2.5 Concurrent coordinators applications

2.5.1 Applications

The network between the ports of the processes can be structured by dividing it into several sub-networks, each changing states according to its own rules. These sub-networks need not be disjoint: they might involve common processes, the same ports of these processes, or even feature common streams. The global network is defined as the union of the local networks.

The way to specify this is to have several concurrent coordinators, each managing its own sub-network. They evolve independently, and add up their behaviors to define the evolution of the global network. The only difference with the single coordinator case is that there are

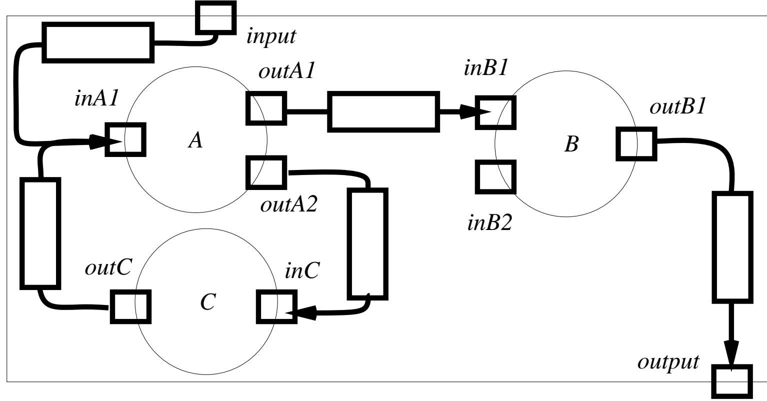

Figure 6: Single-coordinator example: the state s_3.

several coordinators, each of them behaving exactly the same way as the one defined earlier. When an event is raised by a process, they all receive it: this event occurrence is *broadcast*, and each coordinator reacts according to its set of states. All coordinators that can react to an event occurrence, do so simultaneously: the resulting global state of the application is composed of the states of all individual processes.

Such an application terminates when all coordinators have terminated.

One atomic process terminates at a time, just like one atomic process raises an event at a time. In the cases of event exchange, all coordinators that can, do react at the same time; in the case of termination, the termination of one atomic process can cause the termination of one or more coordinators at the same time, which can in turn cause the termination of further processes, in a cascading effect.

The syntax of this concurrent coordinators version is thus simply:

$$\begin{array}{rcl} \langle application \rangle & ::= & \langle atomic \rangle^+ \\ & & \langle coordinator \rangle^+ \end{array}$$

2.5.2 An example with concurrent coordinators

In table 2, the program is composed of two atomic processes `A1` and `A2`, and of two coordinators: `C1` with states s_1 and s_2, and `C2` with states s'_1, s'_2 and s'_3.

Each state of each individual coordinator corresponds to the subnetworks illustrated in fig. 7. When the coordinators act concurrently, their sub-networks merge into a global network. The states the application can be in, depend on the way the different coordinators react to event occurrences. In our example, when event occurrence `e1.A1` is raised, `C1` is in state s_1 and `C2` in state s'_1: i.e., the application, on the raising of `e1.A1`, will be in a state "merging" s_1 and s'_1: let us call it $s_1s'_1$. When event occurrence `e1.A2` is raised, `C1` is necessarily in state s_2, and `C2` in state s'_2: the application is in state $s_2s'_2$.

When event occurrence `e2.A2` is raised, the coordinator `C2` goes into state s'_3, but `C1` is not affected, because it has no corresponding state. Thus the state of the application can

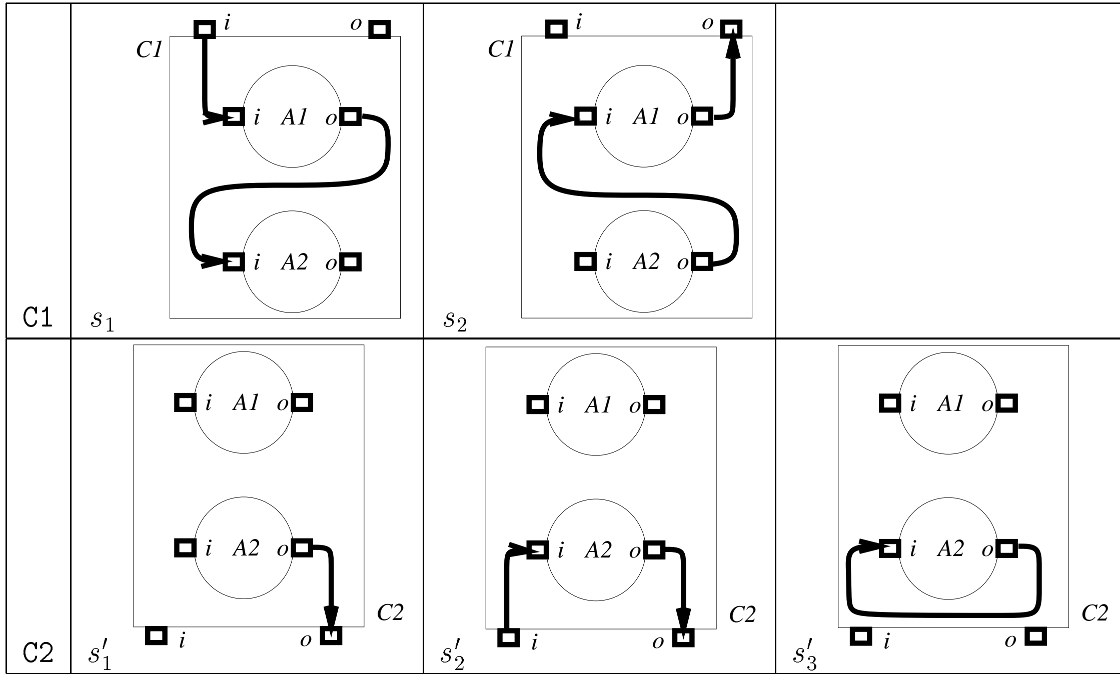

Figure 7: Two-coordinators example: subnetworks for C1 and C2.

be, if the previous state of C1 was s_1: $s_1s'_3$; or, if the previous state of C1 was s_2: $s_2s'_3$. These states $s_1s'_1$, $s_2s'_2$, $s_1s'_3$, and $s_2s'_3$, are illustrated in figure 8.

2.6 Minifold and its complete grammar

The features introduced until now already provide us with the basic constructs of Minifold.

Augmenting them with other features is possible: this is discussed in section 5. Also, notational facilities can be introduced, as we do in section 4 by defining arrays of processes. However, the basic elements that we wanted to integrate in this language kernel i.e., processes, ports, streams, and events, are present, even if they are in their simplest form.

The grammar of the language is given in table 3. As said earlier, the non-terminals ⟨*process*⟩, ⟨*port*⟩ and ⟨*event*⟩ are identifiers.

3 Formal model of Minifold

In this section, we define the operational semantics of Minifold, using transition systems. We first give the structures with which to build a model of an application, and to give its states; we give the rules for the translation of an application into these structures, and then give the rules describing the transitions from one state to another, for the different actions that can be taken by an application. We also introduce an alternative representation of coordinators as automata, and of applications as the synchronized product of these automata.

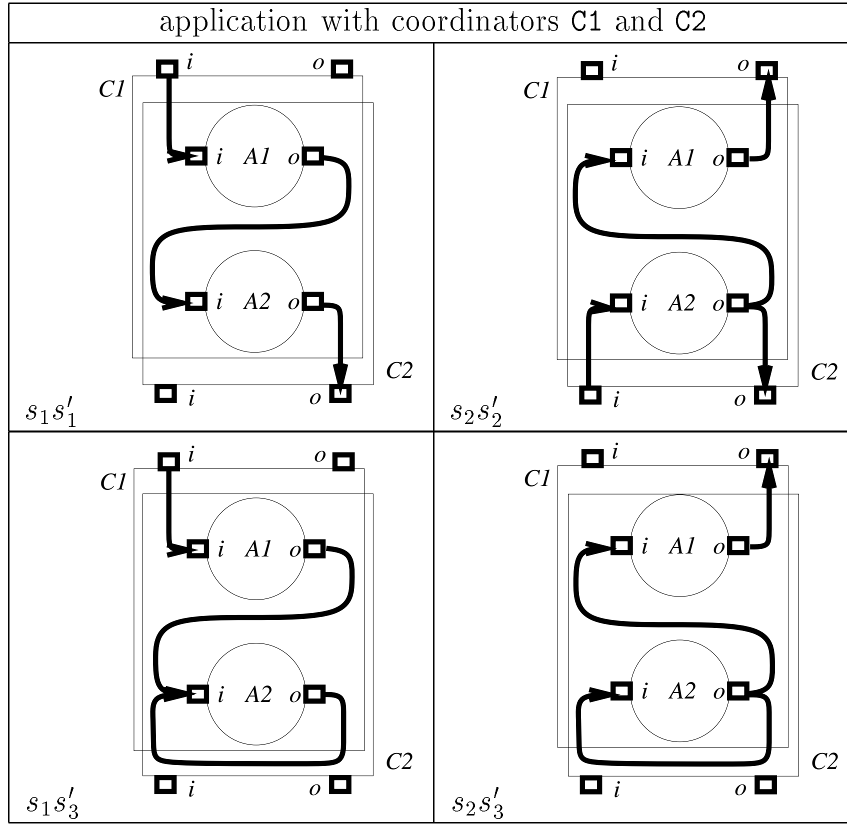

Figure 8: Two-coordinators example: networks for the application.

3.1 States of an application

In this section we define a formal model of MINIFOLD[6]. The states of applications will be defined in terms of the states of their components i.e., ports, atomic and coordinator processes, and streams.

Atomic processes. They are characterized by:

- a name P,
- a set E of events e that can be raised by P (occurrences ϵ will have the form $e.P$).

These aspects are formulated in a tuple: $\langle P,E \rangle$. The terminal state of an atomic process is noted $\bot$.

Coordinators. They are characterized by:

- their name P,

[6] We could call it $\mu\nu$FOLD (To be read: MUNUFOLD).

- their set of states S, each state being of the form $\langle L_s, N_s \rangle$, where:
  - L_s is the label of that state i.e., a set of event occurrences,
  - N_s is a network, i.e. a set of pairs $\langle so,si \rangle$, where so and si are the names of the source port (so) and the sink port (si).
- N is the current network of the coordinator process, of the form described above.

This is formulated in a tuple: $\langle P,S,N \rangle$. The terminal state of a coordinator process is noted $\perp$.

The units queues. The sets of units are *first-in first-out* queues, of unbounded size. They are written as lists: $[u_1, \ldots u_n]$. The empty list is $[\,]$. Operations on these lists of units are:

- $empty = [\,]$,
- $get([u_1, \ldots u_n]) = u_1$,
- $put(u, [u_1, \ldots u_n]) = [u_1, \ldots u_n, u]$,
- $rest([u_1, u_2, \ldots u_n]) = [u_2, \ldots u_n]$.

The cases: $get([\,])$ and $rest([\,])$ are undefined.

Ports. They are defined by

- a name p of the form $\langle port\ name \rangle$ i.e., $\langle process \rangle.\langle port \rangle$,
- contents pc: a set of units. The set of units is defined to have a *first-in first-out* behavior.

This is noted as a pair: $\langle p,pc \rangle$.

Streams. They are defined by:

- the name of their *source* port so,
- the name of their *sink* port si,
- the set of names of the coordinators which installed them: sl;
- their contents sc: a set of units.

i.e. the tuple: $\langle so,si,sl,sc \rangle$.

A newly installed stream has an empty content: $\langle so,si,\{p\},[\,] \rangle$, where p is the name of the process installing it.

Application state. An application is defined as a tuple $\langle \mathcal{A}, \mathcal{C}, \mathcal{P}, \mathcal{S} \rangle$ where:

- $\mathcal{A}$ is the set of atomic processes,
- $\mathcal{C}$ is the set of coordinator processes,
- $\mathcal{P}$ is the set of ports,
- $\mathcal{S}$ is the set streams.

Terminated components of an applications disappear from the set to which they belong. The terminal state of an application is noted $\perp$.

3.2 Construction of the state of a program

This section describes the translation from MINIFOLD to the formal model.[7]

An application is translated into a tuple $\langle \mathcal{A}, \mathcal{C}, \mathcal{P}, \mathcal{S} \rangle$, computed on an empty tuple by the closure of the transition relation:

$$\langle\ \langle application \rangle,\ \langle \emptyset, \emptyset, \emptyset, \emptyset \rangle\ \rangle \longrightarrow^{*} \langle \mathcal{A}, \mathcal{C}, \mathcal{P}, \mathcal{S} \rangle$$

An atomic process is translated into a pair $\langle P,E \rangle$, featuring its name P and the set of its raisable events E, which is added to $\mathcal{A}$, while its ports, the input ones as well as the output ones, are added to $\mathcal{P}$:

$$\frac{\langle\ \langle events \rangle,\ \emptyset\ \rangle \longrightarrow^{*} E,\quad \langle\ \langle ports\ in \rangle,\ \mathcal{P}\ \rangle \longrightarrow^{*} \mathcal{P}',\quad \langle\ \langle ports\ out \rangle,\ \mathcal{P}'\ \rangle \longrightarrow^{*} \mathcal{P}''}{\begin{array}{l}\langle\ \texttt{atomic}\ \langle process \rangle\ \langle ports\ in \rangle\ \langle ports\ out \rangle\ \langle events \rangle\ \langle element \rangle^{+},\ \langle \mathcal{A}, \mathcal{C}, \mathcal{P}, \mathcal{S} \rangle\ \rangle \\ \qquad\qquad \longrightarrow\ \langle\ \langle element \rangle^{+},\ \langle \mathcal{A} \cup \{ \langle\ \langle process \rangle,\ E\ \rangle \}, \mathcal{C}, \mathcal{P}'', \mathcal{S} \rangle\ \rangle\end{array}}$$

The list of events is transformed into a set containing all the event occurrences (we show here only the general case, leaving the management of the end of the list to the reader's imagination):

$$\langle \texttt{event}\ \langle event \rangle \texttt{,} \langle events \rangle\ ,E \rangle \longrightarrow\ \langle \texttt{event}\ \langle events \rangle ,E \cup \{ \langle event \rangle \} \rangle$$

The lists of ports are transformed into a single set containing for each port a pair $\langle p,pc \rangle$ where p is the name of the port, and pc is its contents: this latter is initially empty (we show here also only the general cases, leaving the management of the end of the lists to the reader's imagination):

$$\langle \texttt{in}\ \langle port \rangle \texttt{,} \langle ports \rangle ,\mathcal{P} \rangle \longrightarrow\ \langle\ \texttt{in}\ \langle ports \rangle ,\mathcal{P} \cup \{ \langle\ \langle port \rangle,\ empty\ \rangle \}\ \rangle$$

$$\langle \texttt{out}\ \langle port \rangle \texttt{,} \langle ports \rangle ,\mathcal{P} \rangle \longrightarrow\ \langle\ \texttt{out}\ \langle ports \rangle ,\mathcal{P} \cup \{ \langle\ \langle port \rangle,\ empty\ \rangle \}\ \rangle$$

[7] Actually, the introduction of $\langle element \rangle$ makes this translation more liberal than MINIFOLD, but as it is a strict superset of MINIFOLD that is recognized, this does not impair the obtained result.

In any case, the empty word ε is translated into nothing, leaving a set $\mathcal{E}$ unaffected in this terminal rule:

$$\langle \varepsilon, \mathcal{E} \rangle \longrightarrow \ \mathcal{E}$$

A coordinator process is translated into a tuple $\langle P,S,N \rangle$, of which P is its name, S is its set of states, and N is its current network (i.e., in a sense, state): this latter is initially empty. This tuple is added to the set $\mathcal{C}$. Its processes, the input ones as well as the output ones, are added to $\mathcal{P}$:

$$\frac{\langle\ \langle \textit{ports in} \rangle,\ \mathcal{P}\ \rangle \longrightarrow^* \mathcal{P}', \quad \langle\ \langle \textit{state} \rangle^+,\ \emptyset\ \rangle \longrightarrow^* S, \quad \langle\ \langle \textit{ports out} \rangle,\ \mathcal{P}'\ \rangle \longrightarrow^* \mathcal{P}''}{\begin{array}{r}\langle\ \texttt{coordinator}\ \langle \textit{process} \rangle\ \langle \textit{ports in} \rangle\ \langle \textit{ports out} \rangle\ \{\ \langle \textit{state} \rangle^+\ \}\ \langle \textit{element} \rangle^*, \\ \langle \mathcal{A}, \mathcal{C}, \mathcal{P}, \mathcal{S} \rangle\ \rangle \\ \longrightarrow\ \ \langle\ \langle \textit{element} \rangle^*,\ \langle\ \mathcal{A}, \mathcal{C} \cup \{\langle\ \langle \textit{process} \rangle,\ S,\ \emptyset\ \rangle\}, \mathcal{P}'', \mathcal{S}\ \rangle\end{array}}$$

Each state is translated into a pair $\langle L,N \rangle$ where L is the set of event occurrences translating the label, and N is the set of port-pairs (representing the ports of a stream) translating the network:

$$\frac{\langle\ \langle \textit{label} \rangle,\ \emptyset\ \rangle \longrightarrow L, \qquad \langle\ \langle \textit{network} \rangle,\ \emptyset\ \rangle \longrightarrow N}{\langle\ \langle \textit{label} \rangle\ :\ \langle \textit{network} \rangle\ .\ \langle \textit{states} \rangle^*,\ S\ \rangle \longrightarrow\ \ \langle\ \langle \textit{states} \rangle^*,\ S \cup \{\langle L,N \rangle\}\ \rangle}$$

A label is translated into a set of event occurrences:

$$\langle\ \langle \textit{event-occ} \rangle, \langle \textit{event-occs} \rangle\ ,\ E \rangle \longrightarrow\ \ \langle\ \langle \textit{event-occs} \rangle,\ E \cup \{\langle \textit{event-occ} \rangle\}\ \rangle$$

$$\langle\ \langle \textit{event-occ} \rangle\ ,\ E \rangle \longrightarrow\ \ E \cup \{\langle \textit{event-occ} \rangle\}$$

Each stream of a network expression is translated into a pair $\langle so,si \rangle$ where so is the name of its source port, and si is the name of its sink port.

$$\langle\ \langle \textit{port}_1 \rangle\ \texttt{->}\ \langle \textit{port}_2 \rangle\ ,\ N \rangle \longrightarrow\ \ N \cup \{\langle \langle \textit{port}_1 \rangle, \langle \textit{port}_2 \rangle \rangle\}$$

A pipe-line is a set of streams.

$$\langle\ \langle \textit{port}_1 \rangle \texttt{->} \langle \textit{port}_2 \rangle \texttt{+} \langle \textit{pipe-line} \rangle\ ,\ N \rangle \longrightarrow\ \ \langle\ \langle \textit{pipe-line} \rangle,\ N \cup \{\langle \langle \textit{port}_1 \rangle, \langle \textit{port}_2 \rangle \rangle\}\ \rangle$$

A group or network is translated into the set of the pipe-lines translations:

$$\frac{\langle\ \langle \textit{pipe-line} \rangle,\ \emptyset \rangle \longrightarrow\ N'}{\langle\ \langle \textit{pipe-line} \rangle\ \texttt{+}\ \langle \textit{network} \rangle\ ,\ N \rangle \longrightarrow\ \ \langle\ \langle \textit{network} \rangle,\ N \cup \{N'\}\ \rangle}$$

3.3 Transitions

For the different possible actions presented informally in previous sections, transition rules define the changes in the states of the application and of its components.

3.3.1 Unit exchange between ports and streams

We first present the exchange of units between ports and streams: this involves the capability for a stream to receive a unit and to send it out, and the same two capabilities for a port. From these bases, we can define the passing of a unit from a stream to a port, and from a port to streams.

Stream receiving a unit. The transition is labeled by ⟨*port name*⟩?⟨*unit*⟩, where ⟨*port name*⟩ is the name of the source port with which the communication is made, namely: from which the unit is taken in i.e., the port sending the unit. This unit is appended to the stream's contents. The condition for this transition to be made, is that the stream is installed i.e., that its list sl of names of installer processes is not empty:

$$\frac{sl \neq \emptyset}{\langle so,si,sl,sc \rangle \xrightarrow{so?u} \langle so,si,sl,append(u,sc) \rangle}$$

Stream sending a unit. The transition is labeled by ⟨*port name*⟩!⟨*unit*⟩, where ⟨*port name*⟩ is the name of the sink port with which the communication is made, namely: to which the unit is sent i.e., the port receiving the unit. This unit is removed from the stream's non empty contents, if the stream is installed:

$$\frac{sc \neq [\,] \qquad sl \neq \emptyset}{\langle so,si,sl,sc \rangle \xrightarrow{si!get(sc)} \langle so,si,sl,rest(sc) \rangle}$$

Port receiving a unit. The port p can make a transition when it receives a unit u sent explicitly to him, hence the label $p?u$; it appends it in its contents:

$$\langle p,pc \rangle \xrightarrow{p?u} \langle p,append(u,pc) \rangle$$

Port sending a unit. The port p can send the first unit u taken from its non empty contents, and sends it to all the streams connected to it as a source; the label $p!u$ carries the port name:

$$\frac{pc \neq [\,]}{\langle p,pc \rangle \xrightarrow{p!get(pc)} \langle p,rest(pc) \rangle}$$

Passing a unit from a stream to a port. A unit u passes from a stream S to a port $P = \langle p,pc \rangle$ if the stream can make a sending transition for this port and the port can make a receiving transition. We note $\mathcal{E}[e'/e]$ the set $(\mathcal{E} \setminus \{e\}) \cup \{e'\}$ i.e., the set $\mathcal{E}$ where a new element e' *replaces* element e.

Then, the application is modified in its ports set $\mathcal{P}$ and in its streams set $\mathcal{S}$:

$$\frac{\exists S \in \mathcal{S}, S \xrightarrow{p!u} S', \quad \exists P \in \mathcal{P}, P \xrightarrow{p?u} P'}{\langle \mathcal{A}, \mathcal{C}, \mathcal{P}, \mathcal{S} \rangle \xrightarrow{v} \langle \mathcal{A}, \mathcal{C}, \mathcal{P}[P'/P], \mathcal{S}[S'/S] \rangle}$$

The transition is labeled by $v = \langle p, u \rangle$, in order to mark it with the process p and the unit u involved in the exchange.

Passing a unit from a port to streams. A unit u passes from a port $P = \langle p, pc \rangle$ to streams in $\mathcal{S}$ if the port can make a sending transition and some streams in $\mathcal{S}$ can make a receiving transition for this port. Then, the application is modified in its ports set $\mathcal{P}$ and in its streams set $\mathcal{S}$, by the application level transition, labeled by $v = \langle p, u \rangle$.

For this latter set of streams, we need to write that all the streams that can make a transition, do it, while the others remain in the same state. Therefore, we introduce a transition relation between sets, defined in terms of the possible transitions of its elements. For a set $\mathcal{E}$, the transition $\mathcal{E} \longrightarrow_{each} \mathcal{E}'$ means that $\mathcal{E}'$ is the set of elements e' resulting from the application, when possible, of the transition to an element e of $\mathcal{E}$: $e \longrightarrow e'$, or e itself otherwise[8].

$$\frac{\exists P \in \mathcal{P}, P \stackrel{p!u}{\longrightarrow} P', \quad \mathcal{S} \stackrel{p?u}{\longrightarrow}_{each} \mathcal{S}'}{\langle \mathcal{A}, \mathcal{C}, \mathcal{P}, \mathcal{S} \rangle \stackrel{v}{\longrightarrow} \langle \mathcal{A}, \mathcal{C}, \mathcal{P}[P'/P], \mathcal{S}' \rangle}$$

3.3.2 Event exchange between atomic and coordinator processes

The exchange of an event occurrence between an atomic process and coordinators involves the capability of an atomic process to raise an event, the capability of a coordinator to react to en event, and from the point of view of the data-flow network, the modification to its global state must be deduced from the modifications to the individual sub-networks, which are in turn induced by the new state of each coordinator process.

From these bases, we can describe the effects of an event occurrence exchange on the states of the processes and on the state of the data-flow network.

Atomic process raising an event occurrence. An atomic process P can make a transition raising an event occurrence $e.P$ (labeled: "$!e.P$") if e is in the set of raisable events of P. Its state does not change from the point of view of our model:

$$\frac{e \in E}{\langle P, E \rangle \stackrel{!e.P}{\longrightarrow} \langle P, E \rangle}$$

Coordinator process reacting to event occurrence. A coordinator process P can make a transition receiving event occurrence ϵ (labeled "$?\epsilon$"), which is an event occurrence of the form $e.P$, if a state $\langle L, N' \rangle$ belongs to its set of states S such that $\epsilon \in L$; the current network N of the process is then changed into N':

$$\frac{\langle L, N' \rangle \in S, \qquad \epsilon \in L}{\langle P, S, N \rangle \stackrel{?\epsilon}{\longrightarrow} \langle P, S, N' \rangle}$$

[8] In particular, if the transition from e to e' is not deterministic, then $\longrightarrow_{each}$ isn't either, and $\mathcal{E}'$ is one possible outcome of the transition, with each e' being one possible transform of some e in $\mathcal{E}$.

Modifications to the global network from one local sub-network. Each coordinator $\langle P,S,N \rangle$ has a local network of current state N, set of sets of pairs of port names, corresponding to streams installed by P. The distinction between groups and pipe-lines is irrelevant to their installation: the set of all streams must be installed. Therefore, we introduce a flattened network $\mathcal{N}$, the union of N's elements: $\mathcal{N} = \bigcup_{N_p \in N} N_p$. In $\mathcal{S}$, we have the streams $\langle so,si,sl,sc \rangle$. We have several cases:

- if $\langle so,si \rangle$ is in $\mathcal{N}$, and $\langle so,si,sl,sc \rangle$ is in $\mathcal{S}$, then P is added to the set of installers: $sl \cup \{P\}$:

$$\frac{s = \langle so,si \rangle \in \mathcal{N}, \qquad S = \langle so,si,sl,sc \rangle \in \mathcal{S}}{\langle \mathcal{S}, \mathcal{N} \rangle \xrightarrow{P} \langle\ \mathcal{S}[\ \langle so,si,sl \cup \{P\},sc \rangle\ /\ S\ ],\ \mathcal{N} \setminus \{s\}\ \rangle}$$

- if $\langle so,si \rangle$ is in $\mathcal{N}$ and $\langle so,si,sl,sc \rangle$ is *not* in $\mathcal{S}$, then a new stream must be added to $\mathcal{S}$, with P as installer, and empty contents:

$$\frac{s = \langle so,si \rangle \in \mathcal{N}, \qquad \langle so,si,sl,sc \rangle \notin \mathcal{S}}{\langle \mathcal{S}, \mathcal{N} \rangle \xrightarrow{P} \langle\ \mathcal{S} \cup \{\langle so,si,\{P\},empty \rangle\},\ \mathcal{N} \setminus \{s\} \rangle}$$

- if $\langle so,si \rangle$ is *not* in $\mathcal{N}$ and there is a $\langle so,si,sl,sc \rangle$ in $\mathcal{S}$ such that $P \in sl$, then it is a stream that was removed from the current local network of P, thus P must be removed from sl:

$$\frac{S = \langle so,si,sl,sc \rangle \in \mathcal{S}, \qquad P \in sl \qquad \langle so,si \rangle \notin \mathcal{N},}{\langle \mathcal{S}, \mathcal{N} \rangle \xrightarrow{P} \langle\ \mathcal{S}[\ \langle so,si,sl \setminus \{P\},sc \rangle\ /\ S\ ],\ \mathcal{N}\ \rangle}$$

  At this point, if $sl \setminus \{P\} = \emptyset$, then no units flow through the stream. Its contents are kept until some coordinator re-installs the stream[9].

- finally, this rule terminates the transitions:

$$\langle \mathcal{S}, \emptyset \rangle \xrightarrow{P} \mathcal{S}$$

Modifications to the global network from all the local sub-networks. The global network is a combination of all the local networks: each coordinator process $C = \langle P,S,N \rangle$ of $\mathcal{C}$ contributes its local network N (or more easily its flattened form $\mathcal{N} = \bigcup_{N_p \in N} N_p$ introduced before), thus modifying $\mathcal{S}$ into $\mathcal{S}'$, in a transition labeled by the process' name P. The following rules accumulate the modifications for all coordinators in $\mathcal{C}$:

$$\frac{C = \langle P,S,N \rangle \in \mathcal{C}, \qquad \langle \mathcal{S}, \bigcup_{N_p \in N} N_p \rangle \xrightarrow{P}{}^{*} \mathcal{S}'}{\langle \mathcal{S}, \mathcal{C} \rangle \longrightarrow \langle \mathcal{S}', \mathcal{C} \setminus \{C\} \rangle}$$

and this rule for termination of the transition:

$$\langle \mathcal{S}, \emptyset \rangle \longrightarrow \mathcal{S}$$

[9] In MANIFOLD[1], the stream is said to be in a *dormant* state, and an alternative specification was chosen: units are considered to be lost, sc being reset to $\emptyset$. Further versions of MANIFOLD feature both mechanisms.

Raising and reaction to an event occurrence. An application makes a transition by raising and reacting to an event occurrence:

- if some atomic port of $\mathcal{A}$ can make the transition for the raising of event ϵ,
- and the set of coordinators $\mathcal{C}$ can make the transition to $\mathcal{C}'$ corresponding to the transitions of each its elements reacting (or not) to the event occurrence ϵ,
- and the set of streams $\mathcal{S}$ makes transitions of modifications according to the new states of the coordinators.

$$\frac{\exists A \in \mathcal{A}, A \xrightarrow{!\epsilon} A', \quad \mathcal{C} \xrightarrow{?\epsilon}_{each} \mathcal{C}', \quad \langle \mathcal{S}, \mathcal{C}' \rangle \longrightarrow^* \mathcal{S}'}{\langle \mathcal{A}, \mathcal{C}, \mathcal{P}, \mathcal{S} \rangle \xrightarrow{\epsilon} \langle \mathcal{A}[A'/A], \mathcal{C}', \mathcal{P}, \mathcal{S}' \rangle}$$

3.3.3 Termination of processes and networks

Termination of applications, coordinators, networks and atomic processes is defined hierarchically.

Termination of atomic processes. They can terminate without condition. Their state is transformed into $\perp$:

$$\langle A, E \rangle \xrightarrow{\dagger} \perp$$

Termination of networks. The transition system $\longrightarrow_n$ formalizes the termination of networks, evaluating the network of non-terminated streams N' for a network N, in the context of an application with sets of coordinators $\mathcal{C}$ and of atomics $\mathcal{A}$.

A stream is broken if a process owning one of its ports is terminated i.e., if it belongs neither to $\mathcal{A}$ nor to $\mathcal{C}$:

$$\frac{(\langle P,E \rangle \notin \mathcal{A} \wedge \langle P,S,N \rangle \notin \mathcal{C}) \vee (\langle P',E' \rangle \notin \mathcal{A} \wedge \langle P',S',N' \rangle \notin \mathcal{C})}{\langle\ \langle P.p, P'.p' \rangle,\ \mathcal{C},\ \mathcal{A}\ \rangle \longrightarrow_n \perp}$$

otherwise it is kept as it is:

$$\frac{(\langle P,E \rangle \in \mathcal{A} \vee \langle P,S,N \rangle \in \mathcal{C}) \wedge (\langle P',E' \rangle \in \mathcal{A} \vee \langle P',S',N' \rangle \in \mathcal{C})}{\langle\ \langle P.p, P'.p' \rangle,\ \mathcal{C},\ \mathcal{A}\ \rangle \longrightarrow_n \langle P.p, P'.p' \rangle}$$

A pipe-line being a set of streams, it is broken if one of its streams is broken:

$$\frac{\exists s \in N, \langle s, \mathcal{C}, \mathcal{A} \rangle \longrightarrow_n \perp}{\langle N, \mathcal{C}, \mathcal{A} \rangle \longrightarrow_n \emptyset}$$

otherwise it is kept as it is:

$$\frac{\forall s \in N, \langle s, \mathcal{C}, \mathcal{A}\rangle \longrightarrow_n s, s \neq \perp}{\langle N, \mathcal{C}, \mathcal{A}\rangle \longrightarrow_n N}$$

A group being a set of pipe-lines, terminates when all its members are terminated: if one of them terminates, it is removed from N_g:

$$\frac{N_p \in N_g, \langle N_p, \mathcal{C}, \mathcal{A}\rangle \longrightarrow_n \emptyset}{\langle N_g, \mathcal{C}, \mathcal{A}, N'_g\rangle \longrightarrow_n \langle N_g \setminus \{N_p\}, \mathcal{C}, \mathcal{A}, N'_g\rangle}$$

otherwise it is kept in its updated form:

$$\frac{N_p \in N_g, \langle N_p, \mathcal{C}, \mathcal{A}\rangle \longrightarrow_n N'_p, \qquad N'_p \neq \emptyset}{\langle N_g, \mathcal{C}, \mathcal{A}, N'_g\rangle \longrightarrow_n \langle N_g \setminus \{N_p\}, \mathcal{C}, \mathcal{A}, N'_g \cup \{N'_p\}\rangle}$$

This transformation is completed by the following rule:

$$\langle \emptyset, \mathcal{C}, \mathcal{A}, N_g\rangle \longrightarrow_n N_g$$

Termination of states. As said informally before, the termination of a state can involve two reasons: reachability or networks.

Unreachable states : The transition $\longrightarrow_r$ evaluates the set of states for which the label is accessible.

A state $\langle L,N\rangle$ is unreachable with regard to a set of atomic processes $\mathcal{A}$, if no $e.P \in L$ has a source P present in $\mathcal{A}$. It is removed from $\mathcal{A}$:

$$\frac{\langle L,N\rangle \in S, \qquad \forall e.P \in L, \langle P, E\rangle \notin \mathcal{A}}{\langle S, \mathcal{A}, S'\rangle \longrightarrow_r \langle S \setminus \{\langle L,N\rangle\}, \mathcal{A}, S'\rangle}$$

otherwise it is kept, with updated label L' featuring only raisable events:

$$\frac{\langle L,N\rangle \in S, \qquad L' = \{e.P \in L | \langle P, E\rangle \in \mathcal{A}\}, \qquad L' \neq \emptyset}{\langle S, \mathcal{A}, S'\rangle \longrightarrow_r \langle S \setminus \{\langle L,N\rangle\}, \mathcal{A}, S' \cup \{\langle L',N\rangle\}\rangle}$$

This transformation is completed by the following rule:

$$\langle \emptyset, \mathcal{A}, S\rangle \longrightarrow_r S$$

Broken network :

A state $\langle L,N\rangle$ has a broken network with regard to sets $\mathcal{A}$ and $\mathcal{C}$ of atomic and coordinator processes, if all pipe-lines are broken. It is removed from S:

$$\frac{\langle L,N\rangle \in S, \qquad \langle N,\mathcal{C},\mathcal{A},\emptyset\rangle \longrightarrow_n \emptyset}{\langle S,\mathcal{C},\mathcal{A},S'\rangle \longrightarrow_n \langle S\setminus\{\langle L,N\rangle\},\mathcal{C},\mathcal{A},S'\rangle}$$

otherwise, it is kept, its network updated by removal of broken pipe-lines:

$$\frac{\langle L,N\rangle \in S, \langle N,\mathcal{C},\mathcal{A},\emptyset\rangle \longrightarrow_n N', \qquad (N'\neq\emptyset)}{\langle S,\mathcal{C},\mathcal{A},S'\rangle \longrightarrow_n \langle S\setminus\{\langle L,N\rangle\},\mathcal{C},\mathcal{A},S'\cup\{\langle L,N'\rangle\}\rangle}$$

This transformation is completed by the following rule:

$$\langle\emptyset,\mathcal{C},\mathcal{A},S\rangle \longrightarrow_n S$$

Termination of coordinators. A coordinator terminates if all its states are terminated. If its transformation by removal of unreachable states and of broken streams results in a non-empty set of states S'', then $C=\langle P,S,N\rangle$ is updated into $C'=\langle P,S'',N\rangle$:

$$\frac{\langle S,\mathcal{A}\rangle \longrightarrow_r S', \langle S',\mathcal{C},\mathcal{A}\rangle \longrightarrow_n S'', \qquad (S''\neq\bot)}{\langle\ \langle P,S,N\rangle\ ,\mathcal{C}\ ,\mathcal{A}\ \rangle \stackrel{\dagger}{\longrightarrow} \langle P,S'',N\rangle}$$

otherwise it is terminated:

$$\frac{\langle S,\mathcal{A}\rangle \longrightarrow_r S', \qquad \langle S',\mathcal{C},\mathcal{A}\rangle \longrightarrow_n \bot}{\langle\ \langle P,S,N\rangle\ ,\mathcal{C}\ ,\mathcal{A}\ \rangle \stackrel{\dagger}{\longrightarrow} \bot}$$

When a coordinator C terminates, it disappears from the set $\mathcal{C}$, and the whole set is checked again, as other coordinators might terminate in turn:

$$\frac{C\in\mathcal{C}, \langle C,\mathcal{C}\cup\mathcal{C}',\mathcal{A}\rangle \stackrel{\dagger}{\longrightarrow} \bot}{\langle\mathcal{C},\mathcal{A},\mathcal{C}'\rangle \stackrel{\dagger}{\longrightarrow} \langle(\mathcal{C}\cup\mathcal{C}')\setminus\{C\},\mathcal{A},\emptyset\rangle}$$

otherwise it is kept, in its modified form C':

$$\frac{C\in\mathcal{C}, \langle C,\mathcal{C}\cup\mathcal{C}',\mathcal{A}\rangle \stackrel{\dagger}{\longrightarrow} C', \qquad (C'\neq\bot)}{\langle\mathcal{C},\mathcal{A},\mathcal{C}'\rangle \stackrel{\dagger}{\longrightarrow} \langle\mathcal{C}\setminus\{C\},\mathcal{A},\mathcal{C}'\cup\{\mathcal{C}'\}\rangle}$$

3.3.4 Relation between event level and unit level

On the one hand, there is a transition between two states of the application when an event occurrence ϵ is exchanged i.e., raised and reacted to:

$$\langle \mathcal{A}, \mathcal{C}, \mathcal{P}, \mathcal{S} \rangle \xrightarrow{\epsilon} \langle \mathcal{A}', \mathcal{C}', \mathcal{P}, \mathcal{S}' \rangle$$

On the other hand, there is a transition labeled $v = \langle p, u \rangle$ between two states of the application when a unit u is passed through port p (in either direction: from the port to streams, or from a stream to the port). This transition is of the form:

$$\langle \mathcal{A}, \mathcal{C}, \mathcal{P}, \mathcal{S} \rangle \xrightarrow{v} \langle \mathcal{A}, \mathcal{C}, \mathcal{P}', \mathcal{S}' \rangle$$

This transition leaves $\mathcal{A}$ and $\mathcal{C}$ unaffected: this means that process states in our model are not changed by changes in the streams and ports.

The event transitions are thus quite independent of the unit transitions, and can be considered separately. We therefore isolate event-level states, thereby acquiring a higher-level view on the state of an application. The event-level is detached from the circulation of units, and concerns only the states of processes in $\mathcal{A}$ and $\mathcal{C}$. Furthermore, atomic processes in $\mathcal{A}$ do not really change states, as the transition does not modify them: thus the states of coordinators $\mathcal{C}$ are sufficient to define the state of an application at this level. Finally, it can be noted that this coordinator state $\langle P,S,N \rangle$ changes only in N i.e., in its local network.

At this level, one state defined by $\mathcal{C}$ (or actually the sub-networks N of the processes in $\mathcal{C}$) can be seen as the set of all states $\langle \mathcal{A}, \mathcal{C}, \mathcal{P}, \mathcal{S} \rangle$ with the same $\mathcal{C}$. We can recall that there is an initial state for each coordinator, where its sub-network is empty, before reacting to the first event occurrence.

Thus, the states of an application correspond to the different possible configurations of the streams in the network, responding to events arbitrarily raised by the atomic processes. It seems interesting to study this particular aspect of the behavior of applications with a specific model. We therefore propose an alternative model based on finite state automata.

3.4 An alternative model: automaton of an application

Formulated in terms of automata, the event-driven behavior of coordinators and applications corresponds to:

- states, corresponding to the network connection states;
- transitions, leading, for each label, from each state to the state corresponding to that label.

We do not take termination into account, in order to keep the model simple.

We will define such automata for isolated coordinators first, and then combine these automata into an automaton for the application.

3.4.1 Automaton of a coordinator

A coordinator has n states, $i \in [1..n]$, each with a label l_i and a network expression n_i, represented as "$l_i : n_i$.". Each "$l_i : n_i$." corresponds to a pair $\langle L_i, N_i \rangle$ in the formal model in section 3.2, that describes a coordinator as:

`coordinator` $\langle$*process*$\rangle$ `in` $\langle$*ports in*$\rangle$ `out` $\langle$*ports out*$\rangle$ { ... l_i : n_i. ... }

The automaton corresponding to this coordinator is defined by:

- one state s_i for each network N_i, and one state s_0 for the initial state, before any event occurrence has been raised, and corresponding to an empty network ($N_0 = \emptyset$);

- one transition t_{ijk}, $i \in [0..n], j \in [1..n], k \in [1..|L_j|]$ from each state s_i to each state s_j, labeled by each event occurrence ϵ_k in the label L_j (of cardinality $|L_j| = m$ i.e., $L_j = \{\epsilon_1, ..., \epsilon_m\}$) for the destination state s_j.

  We also define null reflexive transitions on each state, representing the fact that when there is no event occurrence to be reacted to, the coordinator remains in the same state, doing nothing. This transition goes from s_i to s_i for each $i \in [1..n]$, and is labeled by the *null event occurrence*, noted ε. In this sense, it is different from the reflexive transitions $t_{iik}, k \geq 1, \epsilon_k \in L_i$. We note the null transitions $t_{ii0}, i \in [1..n]$. Given $k = 0$, an alternative notation can be $\epsilon_0 = \varepsilon$.

  This null event occurrence represents in fact event occurrences for which the coordinator makes no state change: this will be useful when considering asynchronous concurrent coordinators, reacting differently to event occurrences. It enables us to represent that some processes do nothing while others advance.

  In the following, we will explicitly represent ε-transitions only when needed; otherwise, in discussions, figures and tables of the examples, we leave them out of sight, but they are implicitly present.

The automata corresponding to the coordinator processes `C1` and `C2` in the example of section 2.5.2 are illustrated in figure 9. The states correspond to the networks given in figure 7, and the transitions are labeled with the event occurrences that can lead to them.

More formally, we note an automaton following the notations of Arnold [3, 4], because of their adequacy for the combination of automata defined further. A labeled transition system (or automaton) $\mathcal{A}$ is a five-tuple $\langle S, T, \alpha, \lambda, \beta \rangle$ where:

- S is a set of states,

- T is a set of transitions,

- α and β are applications from T in S, associating to each transition t in T the two states $\alpha(t)$ and $\beta(t)$, which are respectively the origin and the goal of the transition t.

- λ is an application from T in the labels alphabet A, associating to each transition t its label $\lambda(t)$.

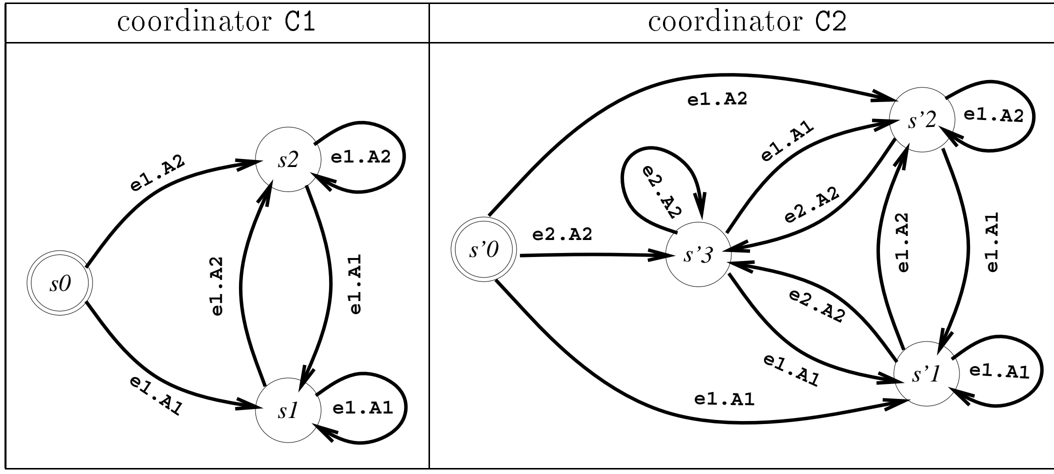

Figure 9: Two-coordinators example: automata for the coordinators C1, C2.

A transition t can then noted: $\alpha(t) \xrightarrow{\lambda(t)} \beta(t)$.

In our framework, we note $\langle S_C, T_C, \alpha_C, \lambda_C, \beta_C \rangle$ the automaton $\mathcal{A}_C$ for a coordinator C modeled by $\langle C,S,N \rangle$, where:

- $S_C = \{s_i | s_0 = \emptyset,\ i \in [1..n] : s_i = N_i \text{ such that } \langle L_i, N_i \rangle \in S\}$,
- $T_C = \{t_{ijk} \mid i \in [0..n], j \in [1..n], k \in [1..|L_j|]\} \cup \{t_{ii0} | i \in [1..n]\}$,
- $\alpha_C(t_{ijk}) = s_i$, such that $s_i \in S_C$,
- $\beta_C(t_{ijk}) = s_j$, such that $s_i \in S_C$,
- $\lambda_C(t_{ijk}) = \begin{cases} \epsilon_k & \text{if } k \geq 1, \text{such that } \epsilon_k \in L_j \text{ of} \langle L_j, N_j \rangle \in S \\ \varepsilon & \text{if } k = 0 \end{cases}$

i.e., the transitions have the form: $t_{ijk} : s_i \xrightarrow{\epsilon_k} s_j$.

The formal language recognized by this automaton $\mathcal{A}_C$ is that of strings on the alphabet $A_C \cup \{\varepsilon\}$, where A_C is in fact $\bigcup_{i \in [1..n]} L_i$. The automaton recognizes the series of event occurrences to which it reacts and of null transitions.

Such an automaton is deterministic here because of the uniqueness of the state of a coordinator corresponding to an event occurrence i.e., the fact that labels denote disjoint sets of event occurrences: $\forall i, j \in [1..n], i \neq j \Rightarrow L_i \cap L_j = \emptyset$. Hence, when in a state s_i, and making a transition on event occurrence ϵ, we never have more than one transition labeled with ϵ, and thus the new state s_j is uniquely defined. Formally, determinism is defined as: $\forall t, t' \in T, \alpha(t) = \alpha(t') \wedge \lambda(t) = \lambda(t') \Rightarrow \beta(t) = \beta(t')$. For t_{ijk} and $t_{i'j'k'}$, we have $\lambda(t_{ijk}) = \lambda(t_{i'j'k'}) = \epsilon \Rightarrow \epsilon \in L_j \wedge \epsilon \in L_{j'}$ i.e., $\epsilon \in L_j \cap L_{j'}$. We saw that labels L_j are disjoint, thus $\exists \epsilon \in L_j \cap L_{j'} \Rightarrow j = j'$ i.e., $L_j = L_{j'}$ and $s_j = s_{j'}$. Hence finally: $\beta(t_{ijk}) = \beta(t_{i'j'k'})$. $\square$

For n states in the body of a coordinator, each with a label $L_i, i \in [1..n]$, the size in number of states of the automaton is $(n+1)$ (including the initial state s_0). For the number

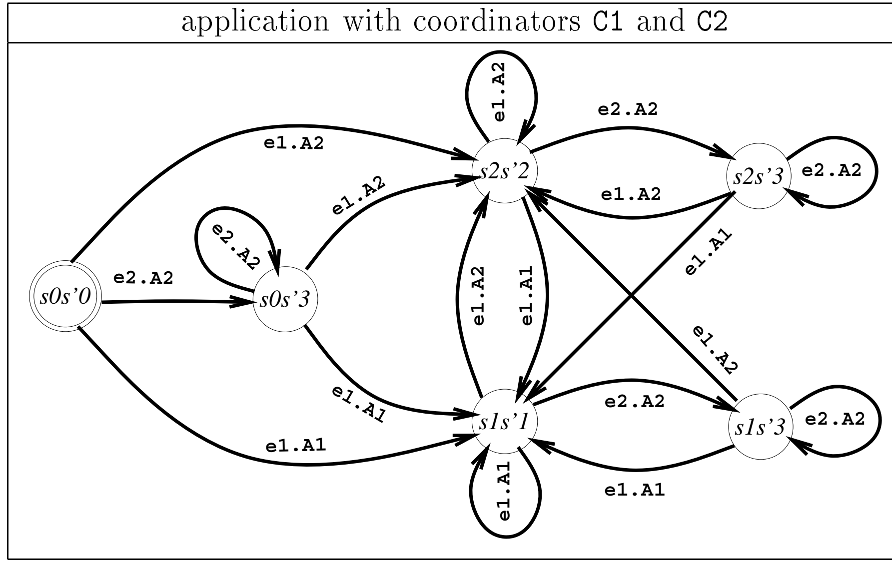

Figure 10: Two-coordinators example: automaton for the application.

of transitions, we have $n+1$ origins, n goals, for each goal $j, j \in [1..n]$, $|L_j|$ ways of getting there and one ε-transition t_{ii0} for each state $s_i, i \in [0..n]$ i.e.: $\sum_{i\in[0..n]}(1 + \sum_{j\in[1..n]} |L_j|)$.

In our example, the coordinators C1 and C2 have the automata (indicated with sizes, not featuring the ε-transitions):

C	C1		C2		
S_C	$\{s_0, s_1, s_2\}$	(3)	$\{s'_0, s'_1, s'_2, s'_3\}$		(4)
T_C	$t_{01} : s_0 \xrightarrow{\texttt{e1.A1}} s_1$ $t_{02} : s_0 \xrightarrow{\texttt{e1.A2}} s_2$ $t_{11} : s_1 \xrightarrow{\texttt{e1.A1}} s_1$ $t_{12} : s_1 \xrightarrow{\texttt{e1.A2}} s_2$ $t_{21} : s_2 \xrightarrow{\texttt{e1.A1}} s_1$ $t_{22} : s_2 \xrightarrow{\texttt{e1.A2}} s_2$	(6)	$t'_{01} : s'_0 \xrightarrow{\texttt{e1.A1}} s'_1$ $t'_{02} : s_0 \xrightarrow{\texttt{e1.A2}} s'_2$ $t'_{03} : s_0 \xrightarrow{\texttt{e1.A2}} s'_3$ $t'_{11} : s'_1 \xrightarrow{\texttt{e1.A1}} s'_1$ $t'_{12} : s'_1 \xrightarrow{\texttt{e1.A2}} s'_2$ $t'_{13} : s'_1 \xrightarrow{\texttt{e2.A2}} s'_3$	$t'_{21} : s'_2 \xrightarrow{\texttt{e1.A1}} s'_1$ $t'_{22} : s'_2 \xrightarrow{\texttt{e1.A2}} s'_2$ $t'_{23} : s'_2 \xrightarrow{\texttt{e2.A2}} s'_3$ $t'_{31} : s'_3 \xrightarrow{\texttt{e1.A1}} s'_1$ $t'_{32} : s'_3 \xrightarrow{\texttt{e1.A2}} s'_2$ $t'_{33} : s'_3 \xrightarrow{\texttt{e2.A2}} s'_3$	(12)

3.4.2 Operations for combining interacting automata

An application is composed of concurrent coordinators. Its behavior is modeled by an automaton combining the automata of its component coordinators. This combination must be defined to correspond to the behavior of applications defined earlier.

As an example, the automaton for the application in the example of section 2.5.2 is illustrated in figure 10. The states correspond to the networks given in figure 8,
and the transitions are labeled with the event occurrences that can lead to them. In particular, we can notice that making a transition on event occurrence e1.A2 from state $s_1s'_1$ leads to state $s_2s'_2$ i.e., the two coordinators C1 and C2 each made a transition simultaneously. However, from the same state $s_1s'_1$, a transition on event occurrence e2.A2 leads to state

$s_1 s'_3$, where only the coordinator `C2` has actually made a transition. Also, there is no state $s_1 s'_2$, because s'_2 is the state accessed in reaction to event occurrence `e1.A2`, and this event occurrence causes `C1` to transit to state s_2: thus, when `C2` is in state s'_2, `C1` can only be in state s_2, not in s_1, which makes an application state $s_1 s'_2$ impossible.

This shows that states of the application automaton are combinations of states of the individual coordinator automata, and that transitions are also combinations of the transitions in the coordinator automata. However, this combination is not just the cross product of the two automata: their interaction is restricted by constraints, that can be used to reduce the size of the resulting automaton.

Therefore, we introduce first the free product of automata, defined by the cross product of its components in the absence of constraint (hence free). Then we introduce the means to express and take into account the interaction constraints.

Free product of automata. Following the definition given by Arnold [4], the free product $\mathcal{A} = \mathcal{A}_1 \times ... \times \mathcal{A}_n$ of automata $\mathcal{A}_i = \langle S_i, T_i, \alpha_i, \lambda_i, \beta_i \rangle$ is defined by:

$$\begin{aligned} \langle S, T, \alpha, \lambda, \beta \rangle &= \prod_{i \in [1..n]} \langle S_i, T_i, \alpha_i, \lambda_i, \beta_i \rangle \\ &= \langle \prod_{i \in [1..n]} S_i \ , \prod_{i \in [1..n]} T_i \ , \langle \alpha_1, ..., \alpha_n \rangle \ , \langle \lambda_1, ..., \lambda_n \rangle \ , \langle \beta_1, ..., \beta_n \rangle \ \rangle \end{aligned}$$

A global state has the form $s = \langle s_1, ..., s_n \rangle$, and it can be changed to the state $s' = \langle s'_1, ..., s'_n \rangle$ by a global transition $t = \langle t_1, ..., t_n \rangle$ such that, in each transition system $\mathcal{A}_i$, there is a transition $t_i : s_i \rightarrow s'_i$. Hence, the origin of a transition t is: $\alpha(t) = \alpha(\langle t_1, ..., t_n \rangle) = \langle \alpha_1(t), ..., \alpha_n(t) \rangle$, its goal: $\beta(t) = \beta(\langle t_1, ..., t_n \rangle) = \langle \beta_1(t), ..., \beta_n(t) \rangle$, and, finally, the label of t is: $\lambda(t) = \lambda(\langle t_1, ..., t_n \rangle) = \langle \lambda_1(t), ..., \lambda_n(t) \rangle$.

The transitions $t = \langle t_1, ..., t_n \rangle$ represent the simultaneity of transitions t_i: this makes the assumption of atomic elementary actions, and is natural for synchronous systems. However, our coordinators are only loosely coupled: while some of them, in an application, might react to an event occurrence ϵ, others, not having any state featuring ϵ in its label, might remain inactive, and stay in the same state. For these cases, and for enabling the representation of an asynchronous behavior, the null transitions, labeled by ε, were introduced: $\forall s \in S, s \xrightarrow{\varepsilon} s$. Then, in the free product, some components can make a transition and change state, while others will remain in the same state, doing nothing under the form of ε.

Synchronized product of automata. The synchronized product is introduced, in order to represent the fact that the possible existence of global transitions depends on the interactions between processes. It restricts the transitions to a sub-set of those in the free product. These interactions entail communication and synchronization constraints, which define the sub-system called a synchronized product.

Such constraints are given in the form of a set SV of possible synchronization vectors, specifying the allowed actions, i.e. labelings of the transitions. I.e., each automaton being labeled on an alphabet A_i, SV is such that: $SV \subseteq A_1 \times ... \times A_n$.

The synchronized product of the $\mathcal{A}_i$ with regard to SV is noted $\langle \mathcal{A}_1, ...\mathcal{A}_n; SV \rangle$, and is the sub-system of the free product that contains only the global transitions $t = \langle t_1, ..., t_n \rangle$ such that $\lambda(t)$ is an element of SV i.e., $\langle \lambda_1(t), ..., \lambda_n(t) \rangle \in SV$.

3.4.3 Automaton of an application

Now that we have these operations at our disposal, we are going to apply them in the framework of the applications.

For an application A with n coordinators $\mathtt{C}_k, k \in [1..n]$:

$$\begin{array}{l} \quad \ldots \\ \mathtt{coordinator}\ C_k \quad \mathtt{in}\ \langle \textit{ports in} \rangle_k\ \mathtt{out}\ \langle \textit{ports out} \rangle_k\ \{\ \ldots\ l_{ki}\ :\ \ n_{ki}\ .\ \ \ldots\ \} \\ \quad \ldots \end{array}$$

we have an automaton with:

- states being vectors of states of the coordinators automata: $s_A = \langle s_1, ..., s_n \rangle$, for $s_k \in S_k$. The actual set of states S_A is a subset of the cross-product of states sets: $S_A \subseteq \prod_{k \in [1..n]} S_k = S_1 \times ... \times S_n$.
- transitions being vectors of transitions of the coordinators automata, in a way similar to the states: $t_A = \langle t_1, ..., t_n \rangle$, for $t_k \in T_k$. Note that the transitions t_k can be ε-transitions, and that a global ε-transition is labeled $\langle \varepsilon, ..., \varepsilon \rangle$. The actual transitions set T_A is also a subset of the cross-product of the T_k: $T_A \subseteq \prod_{k \in [1..n]} T_k = T_1 \times ... \times T_n$.

These sets are restricted in order to respect the constraints on the behavior of the language: when an event is raised, it is received and handled by all coordinators that have a state for it and the others do not change. We will detail this in the remainder of this section.

Product of coordinator automata: case of the example. In our framework of applications of coordinator processes, the constraint is that, as a consequence of the specifications given in section 3.3, one event occurrence is exchanged at a time, and all coordinators that can react to it, do so in the same reaction, the others staying in the same state[10].

In the example of section 2.5.2, the synchronization constraint is that the transitions must be labeled by one the following:

- $\langle$`e1.A1`, `e1.A1`$\rangle$ (that we will note `e1.A1` for a shorthand): when reacting to event occurrence `e1.A1`, both coordinators `C1` and `C2` must make a transition together;
- $\langle$`e1.A2`, `e1.A2`$\rangle$ (that we will note `e1.A2` for a shorthand): when reacting to event occurrence `e1.A2`, in the same way, both coordinators must make a transition together;

[10] This corresponds to the transition on the set $\mathcal{C}$ of coordinators: $\longrightarrow_{each}$ defined in section 3.3.

- $\langle \varepsilon$, e2.A2$\rangle$ (that we will note e2.A2 for a shorthand): only C2 can react to this event occurrence, C1 is unaffected by it: thus C2 makes a transition to react to it, while C1 does nothing i.e., ε.

The synchronized product is obtained by keeping transitions and states from the free product only when they respect the synchronization constraint and are accessible from the initial state $\langle s_0, s'_0 \rangle$. We obtain:

$$\begin{array}{llllll}
s_0s'_0 & \stackrel{\texttt{e1.A1}}{\longrightarrow} s_1s'_1 & s_1s'_1 & \stackrel{\texttt{e1.A1}}{\longrightarrow} s_1s'_1 & s_2s'_2 & \stackrel{\texttt{e1.A1}}{\longrightarrow} s_1s'_1 \\
 & \stackrel{\texttt{e1.A2}}{\longrightarrow} s_2s'_2 & & \stackrel{\texttt{e1.A2}}{\longrightarrow} s_2s'_2 & & \stackrel{\texttt{e1.A2}}{\longrightarrow} s_2s'_2 \\
 & \stackrel{\texttt{e2.A2}}{\longrightarrow} s_0s'_3 & & \stackrel{\texttt{e2.A2}}{\longrightarrow} s_1s'_3 & & \stackrel{\texttt{e2.A2}}{\longrightarrow} s_2s'_3 \\
 & & & & & \\
s_0s'_3 & \stackrel{\texttt{e1.A1}}{\longrightarrow} s_1s'_1 & s_1s'_3 & \stackrel{\texttt{e1.A1}}{\longrightarrow} s_1s'_1 & s_2s'_3 & \stackrel{\texttt{e1.A1}}{\longrightarrow} s_1s'_1 \\
 & \stackrel{\texttt{e1.A2}}{\longrightarrow} s_2s'_2 & & \stackrel{\texttt{e1.A2}}{\longrightarrow} s_2s'_2 & & \stackrel{\texttt{e1.A2}}{\longrightarrow} s_2s'_2 \\
 & \stackrel{\texttt{e2.A2}}{\longrightarrow} s_0s'_3 & & \stackrel{\texttt{e2.A2}}{\longrightarrow} s_1s'_3 & & \stackrel{\texttt{e2.A2}}{\longrightarrow} s_2s'_3
\end{array}$$

Compared to the free product $\mathcal{A}_{free} = \mathcal{A}_{\texttt{C1}} \times \mathcal{A}_{\texttt{C1}}$ of the automata of C1 ($\mathcal{A}_{\texttt{C1}}$: 3 states, 6 transitions) and C2 ($\mathcal{A}_{\texttt{C2}}$: 4 states, 12 transitions), which would have had $3 \times 4 = 12$ states and $6 \times 12 = 72$ transitions, the synchronized product $\mathcal{A}_s$ has 6 states and 18 transitions. The automaton is illustrated in fig. 10.

Product of coordinator automata: general case. In general, an application is composed of coordinators $\texttt{C}_1$, ..., $\texttt{C}_n$; they each have a corresponding automaton

$$\mathcal{A}_{\texttt{C}_i} = \langle S_{\texttt{C}_i}, T_{\texttt{C}_i}, \alpha_{\texttt{C}_i}, \lambda_{\texttt{C}_i}, \beta_{\texttt{C}_i} \rangle$$

The global transitions will be labeled by the synchronization vectors of the form: $v = \langle v_1, ..., v_n \rangle$. For $i \in [1..n]$, each v_i will be the label $\lambda_{\texttt{C}_i}(t_i)$ of some transition $t_i \in T_{\texttt{C}_i}$.

The constraint for these synchronization vectors is that all processes that can react to one event occurrence ϵ do so, while the others do nothing (i.e., a ε-transition). In other terms, the global transition is made on event occurrence ϵ iff, for all processes $\texttt{C}_i$ such that $\exists t \in T_{\texttt{C}_i}, \lambda_{\texttt{C}_i}(t) = \epsilon \neq \varepsilon$, we have: $v_i = \epsilon$, and for the others $\texttt{C}_j$: $v_j = \varepsilon$. We add the vector $\langle \varepsilon, ..., \varepsilon \rangle$ of ε-transitions, that provides a global ε-transition. More formally:

$$SV = \{ \quad v = \langle v_1, ..., v_n \rangle \mid i \in [1..n], t_i \in T_{\texttt{C}_i}, v_i = \lambda_{\texttt{C}_i}(t_i) = \epsilon \neq \varepsilon, \\ j \in [1..n], j \neq i, v_j = \begin{cases} \epsilon & \text{if } \exists t_j \in T_{\texttt{C}_j}, \lambda_{\texttt{C}_j}(t_j) = \epsilon \\ \varepsilon & \text{otherwise} \end{cases} \quad \} \cup \{ \langle \varepsilon, ..., \varepsilon \rangle \}$$

This definition ensures that: $\forall i \in [1..n], v_i \neq \epsilon \Rightarrow v_i = \varepsilon \wedge \nexists t \in T_{\texttt{C}_i}, \lambda_{\texttt{C}_i}(t) = \epsilon$ i.e., only the processes that cannot react don't, and make the ε-transition instead.

We can also note that: $\forall i, j \in [1..n], v_i = v_j \vee v_i = \varepsilon \vee v_j = \varepsilon$, which means that transitions can be distinguished by a label ϵ, as a shorthand of $\langle v_1, ..., v_n \rangle$ where ϵ is the only significant value. In the case of $\langle \varepsilon, ..., \varepsilon \rangle$, the global ε-transition can be labeled by ε.

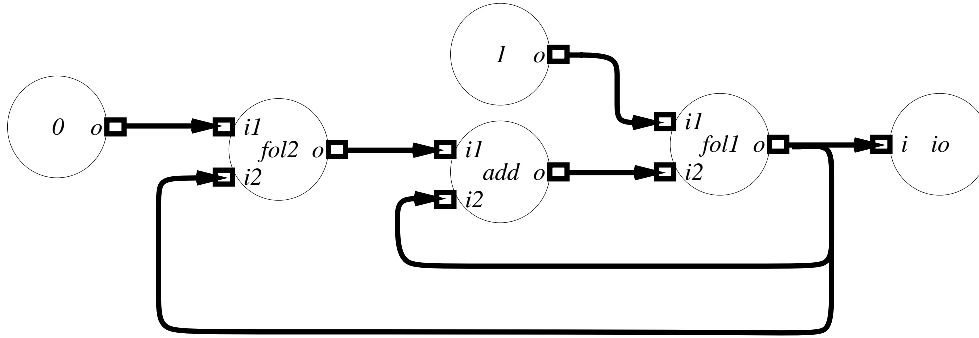

Figure 11: Fibonacci series example: the network of processes.

4 Examples

In this section we treat two classical academic examples for the illustration of the use of programming languages: the computation of the Fibonacci series, and the sieves of Eratosthenes.

4.1 The Fibonacci series

The Fibonacci series consists of the calculation of the numbers $f(n)$, for each positive integer n, such that:

$$\begin{cases} f(0) = 1 \\ f(1) = 1 \\ f(n) = f(n-1) + f(n-2) \end{cases}$$

In terms of a data-flow between processes, this involves essentially an addition process, an input-output interaction process, and a sufficient management so that units to be added are given in the right order.

A new solution to this specific problem is not our primary interest; we get inspiration from that presented by Boussinot [5], and express it our language as presented in table 5, which results in the network illustrated by fig. 11.

The input-output process `io` raises an event `start` when the application must start to produce the Fibonacci series, and takes in the numbers of the series through its input port `i`. The addition process `add` has two input ports `i1` and `i2`, and when it has one unit on each of them, it calculates the sum of their values, and outputs it on its port `o`.

The problem here is to insure that units will be presented at these input ports in an order such that the series of outputs coming out of `o` is the Fibonacci series. For this, we can note that the third equation in the system above defines $f(n)$ as $f(n-1)+f(n-2)$ i.e., in the general case ($n \geq 2$), the unit on port `i1` of `add` must be the unit output by `add` two

additions earlier, and on the input `i1` it must be the result of the former addition; hence, the output of `add` must be reconnected to its inputs.

For the initialization cases ($n = 0$ and $n = 1$), a different management is needed. When $n = 0$, neither $n-1$ nor $n-2$ are defined; thus the result 1 is given directly to process `io`, from the output of the process `const1`, defined to deliver the integer constant `1` on its output port `o`. When $n = 1$, the previous value $f(n-1) = f(0) = 1$, but $n-2$ is still undefined: nevertheless, $f(1)$ is the result of the addition of $f(n-1) = f(0)$ and 0. Hence, the operator `add` is fed on its input `i1` with the integer value 0 output by the process `const0`, and on its input `i2`, with the previous value of the series i.e., the value that was put out by `const1`.

Thus, expressed form the point of view of the series of output values, which is what we are interested in, the results given as input to the input-output process `io` are, first, the output of the process `const1`, followed by the results of the addition, from port `add.o`. The inputs of the addition process `add` are, on its port `i2`: the previous results of the addition (i.e., for the calculation of the n_{th} value of the series, $n \geq 1$, `i2` receives the value of rank $n-1$), and on its input port `i1`: first the output of constant `const0`, followed by the same that `i2` received (i.e., for the calculation of the n_{th} value of the series, $n \geq 2$, `i1` receives the value that `i2` received on the previous operation i.e., the value of rank $n-2$).

In order to program this in MINIFOLD, we must define an operator for the expression "followed by" that we used above. What was meant is that a first element of a series was taken from one source, and all the others from another source. We have two instances `fol1` and `fol2` of such a process, each of them with input ports `i1` and `i2`, and an output port `o`. The first unit output on `o` comes from `i1`, and the subsequent ones come from `i2`. The application calculating the Fibonacci series can be depicted as in fig. 11.

The coordinator process $\texttt{fol}_i$ encoding the "followed by" functionality can be defined with use of atomic processes $\texttt{p}_i$, with one input port `i` and one output port `o`, and raising an event `u` as soon as the first unit arrives in their input port[11]. After having raised the event, the process passes the unit to its output port, as well as all the following units coming on its input. The coordinator $\texttt{fol}_i$ begins by installing a stream between one of its inputs `i1` and the input of $\texttt{p}_i$. This happens when the interaction process `io` raises the `start` event. On reception of the event occurrence $\texttt{u.p}_i$, meaning that the first unit from `i1` has been received, it changes state, breaking the stream from `i1` to $\texttt{p}_i\texttt{.i}$, and installing a stream from $\texttt{fol}_i\texttt{.i2}$ to $\texttt{p}_i\texttt{.i}$, while the stream from $\texttt{p}_i\texttt{.o}$ to $\texttt{fol}_i\texttt{.o}$ remains. The states of this behavior are illustrated by fig. 12.

Compared to the solution presented by Boussinot [5], this one does not feature a notion of `Pre` operator, giving the previous value of a series: this aspect of the problem is taken into account here by the *first-in first-out* behavior of streams.

4.2 The sieves of Eratosthenes

In the previous section, when presenting informally the coordinator $\texttt{fol}_i$, we used an indexed notation that does not belong to the language as it is. We want to introduce here

[11] As such, it is reminiscent of the `guard` pseudo-process in MANIFOLD [1, 10].

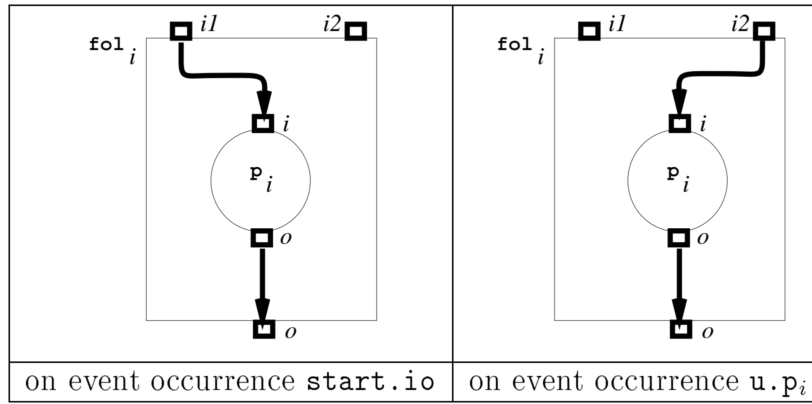

Figure 12: Fibonacci series example: states of the $\texttt{fol}_i$ coordinator.

the possibility to define arrays (uni-dimensional vectors or multi-dimensional matrices) of processes.

4.2.1 Arrays of processes

In the case of $\texttt{fol}_i$ and $\texttt{p}_i$, we might have written:

```
atomic [1..2] p in i out o event u
coordinator [1..2] fol    in i1, i2    out o
{  start.io :  self.i1 -> p[i].i * p[i].o -> self.o .
   u.p[i]  :  self.i2 -> p[i].i * p[i].o -> self.o .     }
```

For an atomic process `p`, this notation means that there are 2 instances, named `p[1]` and `p[2]`, with ports and events of the same name. These will be distinguished by their absolute names: ⟨*event*⟩`.p[i]` for events, and `p[i].`⟨*port*⟩ for ports.

For a coordinator process, it means basically the same thing; for references to indexed processes in its states, the index used is that of the coordinator itself: in the example above, each `fol[i]` coordinates the network around one process `p[i]`, where `i` is the same for `fol` and `p`.

Multi-dimensional arrays of processes follow the same principles: for `P` of 3 dimensions of respective ranges [1..5], [0..10] and [7..9] we note: `P[1..5,0..10,7..9]`. This is just a syntactic augmentation to MINIFOLD, as it can be translated into programs in the previous language by simple extension. It is however an extension to it. The definition of an array or matrix of processes can be multi-dimensional, with one index per dimension.

It follows the syntax:

⟨*range*⟩	::=	`[` ⟨*d-range*⟩ *[* `,` ⟨*d-range*⟩ *]** `]`
⟨*d-range*⟩	::=	⟨*lower bound*⟩ `..` ⟨*upper bound*⟩
⟨*a-array*⟩	::=	`atomic` ⟨*range*⟩ ⟨*process*⟩ `[` ⟨*index*⟩ `]`
		⟨*ports in*⟩ ⟨*ports out*⟩ ⟨*events*⟩
⟨*c-array*⟩	::=	`coordinator` ⟨ *range*⟩ ⟨*process*⟩ `[` ⟨*index*⟩ `]`
		⟨*ports in*⟩ ⟨*ports out*⟩ { ⟨*state*⟩+ }
⟨*application*⟩	::=	*[* ⟨*atomic*⟩ | ⟨*a-array*⟩ *]+* *[* ⟨*coordinator*⟩ | ⟨*c-array*⟩ *]+*

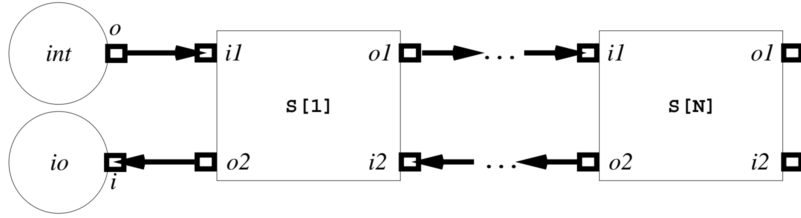

Figure 13: The sieves of Eratosthenes example: the network.

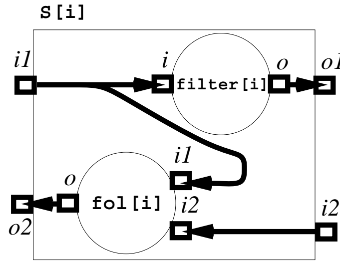

Figure 14: Eratosthenes sieves example: one sieve.

where ⟨*index*⟩ is an identifier (e.g., `i`), and ⟨*lower bound*⟩ and ⟨*upper bound*⟩ are integers.

It is straightforward to translate each array into as much single processes as necessary; thus the semantics need not be extended. Also, in network expressions, indexes of processes are easily interpretable.

4.2.2 The example of the sieves of Eratosthenes

An example suited to this extension is the program calculating the n first prime numbers, following the method of the sieves of Eratosthenes.

A number of processes will share the task of filtering away the integers that are multiples of some already known prime number. For each number passing through this filter, a new filter is given. The program encoding this is given in table 5.

If we name each of these processes `s[i]`, then the application looks like the illustration in fig. 13. A process `int` gives all integers $n \geq 2$ on its output port `o`. The interaction process `io` takes the results in its input port `i`. Each of the sieves `S[i]` takes a series of integers in its input `i1`, outputs those that are not filtered in `o1`, takes in results from further sieves in `i2`, and outputs results in `o2`.

The sieve process `S[i]` itself is illustrated in fig. 14. It coordinates two sub-processes: $\texttt{fol}_i$ is like the one previously described, and allows to output first the prime number for the sieve itself, then those coming from further sieves. The process `filter[i]` is atomic, and, taking integers through its input port, gives out only those that are not a multiple of the

first received (i.e., for the kth unit u_k, it is output if $u_k \bmod u_1 \neq 0$).

The termination of this example can be obtained as follows: when having received `N` units on its input port, the process `io` raises an event `terminate`, and outputs a special unit u_t. The coordinator `main` has a state:

```
terminate.io :  [1..N] ( io.o -> S[i].i2 )
```

The atomic process `p[i]` terminates when receiving u_t.

When each `S[i]` receives u_t on `i2`, it is given through to `p[i]`, which terminates, causing `fol[i]` to terminate, causing `S[i]` to terminate, which, when all of them are terminated, causes `main` to terminate, which causes the application to terminate, thereby causing the termination of `io`, `int` and all the `filter[i]`.

5 Miscellaneous ideas and open problems

Controlling state transitions. In the automata shown in section 3.4 for the examples, there exist transitions between *all* of their states, namely from any state of the application to the state s_i (corresponding to $l_i : n_i$.), labeled with events of the label l_i. The multitude of these transitions makes the automata complex, and compromises the overview on the behavior of a process. Restricting these transitions means controlling the execution path of the program through all the possible transitions.

Ways to moderate this explosion of transitions consist of giving rules restricting which events might cause a transition from a state to an other one. Such rules can define the selection criteria as a function of the current state e.g., restricting transitions to events for which the source is involved in the network of that state: in MANIFOLD this is called *preemptivity* [1, 10].

Coordinators raising events. This possibility could enable coordinators to change state with an internal cause (i.e., with a local event raising), or also interaction between coordinators.

However, even if the coordinators are provided with this possibility, the ultimate source of a state change is still always an atomic process. Indeed, if a coordinator can raise an event (be it locally or externally), it is from one of its state; in order to arrive in this state, the coordinator had to react to an event occurrence. Thus, a coordinator can raise an event only in reaction to another event occurrence, coming either form a coordinator, or from an atomic process. Thus, the cause of a transition will eventually be an event occurrence from an atomic process.

In this sense, the fact that MINIFOLD coordinators do not raise events is a simplification without being a real impoverishment.

Furthermore, the possibility that, in a reaction, several events can be raised and reacted to, possibly by raising other events, implies that one coordinator might receive several event occurrences to which it has to react. A coordinator can be in only one state at a time: these event occurrences must be treated one at a time. The consequence of this is that an

event memory is needed, and that the uncoupling between event reception and treatment introduces asynchrony. Also, the order in which event occurrences are treated must be fixed; in MANIFOLD, this is done non-deterministically.

Network expressions. Networks are graphs, and in MINIFOLD, they are described by the enumeration of their arcs i.e., streams. This is sufficient to describe any graph.

However, having more elaborate constructs would ease the specification of networks. A full graphical language with branching, joining, looping operators could be useful.

From the point of view of the operators `->`, `+` and $*$ that we already introduced, it can also mean the definition of distributivity rules, like:

$$
\begin{aligned}
p_1 \texttt{ -> } p_2 \texttt{ -> } p_3 &= p_1 \texttt{ -> } p_2 \texttt{ * } p_2 \texttt{ -> } p_3 \\
p_1 \texttt{ -> } (p_2 \texttt{ + } p_3) &= p_1 \texttt{ -> } p_2 \texttt{ + } p_1 \texttt{ -> } p_3 \\
p_1 \texttt{ -> } (p_2 \texttt{ * } p_3) &= p_1 \texttt{ -> } p_2 \texttt{ * } p_1 \texttt{ -> } p_3 \\
(n_1 \texttt{ + } n_2) * n_3 &= (n_1 * n_3) \texttt{ + } (n_2 * n_3) \\
n_1 \texttt{ + } (n_2 * n_3) &= (n_1 \texttt{ + } n_3) * (n_2 \texttt{ + } n_3)
\end{aligned}
$$

Producing and using the automata The description of applications in terms of automata in section 3.4 was done in informal accordance with the transition system of the semantics in section 3. Termination states and transitions were left out for simplicity, and unit exchange states and transitions were ignored because of a clear difference of level between the two aspects of the language; for the rest, the event occurrence exchange transitions and the automata describe the same behaviors.

It would be interesting to have rules for the translation of a source program into an automaton (i.e. a compilation into an automaton), following the semantics.

The automata-based model could be useful for a deep and complete analysis of programs. For example, detecting the problematic states or parts of the automaton: unreachable states, states with no outcoming transition, states from which terminal states are unreachable, can be done by simple operations on graphs and automata.

Further, the analysis of applications could benefit from existing results in the area of formal specification and verification of concurrent systems, namely using techniques and concepts as bi-simulation equivalences.

A problem, general to any analysis of an application in MANIFOLD-like languages, is that the semantics of the language does not reflect the behavior of the atomic processes, because they are outside the scope of the language; however it is necessary to know their behavior in order to know the behavior of a whole application. To this end, it should be possible to give partial specifications of their behavior, abstracted to the raising of events and the input and output of units.

Practically, the use of an environment generating tool like ASF+SDF [9] would enable to experiment with the specification of MINIFOLDitself, and to have a whole environment for testing each of its versions by running example programs, and incrementally modify the specification.

6 Conclusion

We have presented MINIFOLD, a kernel for a coordination language, following the MANIFOLD model. We introduced it constructively, illustrated it by examples, and provided it with an operational semantics, as well as a model based on automata. Various extensions are possible, in order to augment the possibilities of the language. The models deserve more attention, in particular automata and the existing concepts in the area of the modeling of concurrent systems might lead to the possibility of formally analyzing the behavior of applications.

The purpose of the study of this very simplified instance of the MANIFOLD concept is to explore models of its behavior, and to give a formalization of its bare essentials. It is intended that the MANIFOLD language can take advantage of this, as guidelines for formalisms underlying practical tools for programs analysis, clarification of its structure and as a basis for the comparison of MANIFOLD with other models.

References

[1] F. Arbab. *Specification of* MANIFOLD. CWI Report, Interactive Systems Dept., CS-R 9220, 1992.

[2] F. Arbab, I. Herman, P. Spilling. *An overview of* MANIFOLD *and its implementation.* CWI Report, Interactive Systems Dept., CS-R 9142, 1991.

[3] A. Arnold. Transition systems and concurrent processes. In *Mathematical problems in Computation theory* (Banach Center Publications, vol. 21), 9 – 20, 1988.

[4] A. Arnold. Systèmes de transitions finis et sémantique des processus communicants. *T.S.I. Technique et Science Informatiques*, vol. 9, no. 3, 1990. (in French)

[5] F. Boussinot. *Réseaux de Processus Réactifs*. Rapport de Recherche, INRIA, Sophia-Antipolis, n^0 1588, Janvier 1992. (in French)

[6] N. Carriero, D. Gelernter. LINDA in context. *Comm. of the ACM*, April 1989, vol. 32, no. 4.

[7] D. Gelernter, N. Carriero. Coordination languages and their significance. *Comm. of the ACM*, February 1992, vol. 35, no. 2.

[8] G. Kahn, D. MacQueen. Coroutines and networks of parallel processes. In *Proceedings of IFIP '77* (A. Finlay, ed.), pp. 993–998, 1977.

[9] P. Klint. *A meta-environment for generating programming environments.* CWI Report, Dept. of Software Technology, to appear, 1992.

[10] E.P.B.M. Rutten, F. Arbab, I. Herman. *Formal Specification of* MANIFOLD*: a Preliminary Study.* CWI Report, Interactive Systems Dept., CS-R 9215, 1992.

[11] E.P.B.M. Rutten, S. Thiébaux. *Formal Semantics of* MANIFOLD*: Specification in ASF+SDF and extensions*. CWI Report, Interactive Systems Dept., to appear, 1992.

List of Figures

1 An atomic process. 5
2 The stream: `A.outA1 -> B.inB1` 6
3 The networks: (a): p->p' + p->p'', (b): p'->p + p''->p. 8
4 A network. 8
5 Single-coordinator example: the state s_2. 11
6 Single-coordinator example: the state s_3. 12
7 Two-coordinators example: subnetworks for `C1` and `C2`. 13
8 Two-coordinators example: networks for the application. 14
9 Two-coordinators example: automata for the coordinators `C1`, `C2`. 26
10 Two-coordinators example: automaton for the application. 27
11 Fibonacci series example: the network of processes. 31
12 Fibonacci series example: states of the $\texttt{fol}_i$ coordinator. 33
13 The sieves of Eratosthenes example: the network. 34
14 Eratosthenes sieves example: one sieve. 34

List of Tables

1 Single-coordinator example: the application. 39
2 Two-coordinators example: the application. 39
3 The grammar of MINIFOLD. 40
4 Fibonacci series example: the application. 40
5 Eratosthenes sieves example: the application. 41

```
atomic A in inA1 out outA1, outA2 event e1, e3

atomic B in inB1, inB2 out outB1 event e2, e3

atomic C in inC out outC event e3

coordinator main
   in input
   out output
{
e1.A : A.outA1->B.inB1
               + C.outC->A.inA1
                             + B.outB1->C.inC .
e2.B : A.outA1->B.inB2 + C.outC->B.inB2
     + B.outB1->C.inC + main.input->B.inB1
                       + C.outC-> main.output .
e3.A , e3.C : A.outA1->B.inB1 + C.outC->A.inA1
     + A.outA2->C.inC + main.input->A.inA1
                      + B.outB1-> main.output .
}
```

Table 1: Single-coordinator example: the application.

```
atomic A1 in i out o event e1

atomic A2 in i out o event e1, e2

coordinator C1    in i    out o
{
e1.A1 :  C1.i->A1.i + A1.o->A2.i .                 s1
e1.A2 :  A2.o->A1.i + A1.o -> C1.o .               s2
}

coordinator C2    in i    out o
{
e1.A1 :  A2.o->C2.o .                              s'1
e1.A2 :  C2.i -> A2.i + A2.o -> C2.o .             s'2
e2.A2 :  A2.o -> A2.i .                            s'3
}
```

Table 2: Two-coordinators example: the application.

⟨*atomic*⟩	::=	`atomic` ⟨*process*⟩ ⟨*ports in*⟩ ⟨*ports out*⟩ ⟨*events*⟩
⟨*ports in*⟩	::=	`in` ⟨*port*⟩ [, ⟨*port*⟩]* | ε
⟨*ports out*⟩	::=	`out` ⟨*port*⟩ [, ⟨*port*⟩]* | ε
⟨*events*⟩	::=	`event` ⟨*event*⟩ [, ⟨*event*⟩]* | ε
⟨*port name*⟩	::=	⟨*process*⟩`.`⟨*port*⟩
⟨*event-occ*⟩	::=	⟨*event*⟩`.`⟨*process*⟩
⟨*stream*⟩	::=	⟨*port name*⟩ `->` ⟨*port name*⟩
⟨*pipe-line*⟩	::=	⟨*stream*⟩ * ⟨*pipe-line*⟩ | ⟨*stream*⟩
⟨*network*⟩	::=	⟨*pipe-line*⟩ `+` ⟨*network*⟩ | ⟨*pipe-line*⟩
⟨*label*⟩	::=	⟨*event-occ*⟩ [, ⟨*event-occ*⟩]*
⟨*state*⟩	::=	⟨*label*⟩ : ⟨*network*⟩ .
⟨*coordinator*⟩	::=	`coordinator` ⟨*process*⟩ ⟨*ports in*⟩ ⟨*ports out*⟩ { ⟨*state*⟩$^+$ }
⟨*application*⟩	::=	⟨*atomic*⟩$^+$ ⟨*coordinator*⟩$^+$

Table 3: The grammar of MINIFOLD.

```
atomic const0 out o

atomic const1 out o

atomic add in i1, i2 out o

atomic io in i event start

atomic p1 in i out o event u

atomic p2 in i out o event u

coordinator fol1    in i1, i2    out o
{  start.io :  fol1.i1 -> p1.i * p1.o -> fol1.o .
   u.p1 :  fol1.i2 -> p1.i * p1.o -> fol1.o .     }

coordinator fol2    in i1, i2    out o
{  start.io :  fol2.i1 -> p2.i * p2.o -> fol2.o .
   u.p2 :  fol2.i2 -> p2.i * p2.o -> fol2.o .     }

coordinator main
 { start.io :  const0.o -> fol2.i1 * fol1.o -> fol2.i2
               * fol2.o -> add.i1 * fol1.o -> add.i2
               * const1.o -> fol1.i1 * add.o -> fol1.i2
               * fol1.o -> io.i .                      }
```

Table 4: Fibonacci series example: the application.

```
atomic int out o

atomic io in i event start terminate

atomic [1 ..  N] p[i] in i out o event u

atomic [1 ..  N] filter[i] in i out o

coordinator [1..N] fol[i]    in i1, i2    out o
 { start.io :  fol[i].i1 -> p[i].i * p[i].o -> fol[i].o .
     u.p[i] :  fol[i].i2 -> p[i].i * p[i].o -> fol[i].o .    }

coordinator [1 ..  N] S[i]    in i1, i2    out o1, o2
 { start.io :  S[i].i1 -> filter[i].i * filter[i].o -> S[i].o1
               * S[i].i1 -> fol[i].i1 * S[i].i2 -> fol[i].i2
                                         * fol[i].o -> S[i].o2 .  }
coordinator main
 {  start.io     :  int.o -> S[1].i1 * S[1].o2 -> io.i
                             * [1..N-1] ( S[i].o1 -> S[i+1].i1
                                    * S[i+1].o2 -> S[i].i2 ) .
    terminate.io :  [1..N] ( io.o -> S[i].i2 ) .                 }
```

Table 5: Eratosthenes sieves example: the application.