Parallel Distributed Algorithm Design through Specification Transformation [1]: The Asynchronous Vision System

Didier BUCHS (+), Daniel MONTEIRO, Fabrice MOURLIN, Denis BRUNET(*)

*(+) : **LGL - DI***
Swiss Federal Institute of Technology
1015 Lausanne
SWITZERLAND
email: Didier.Buchs@di.epfl.ch

(*) : ***CUI***
Université de Genève
24, rue du général Dufour
1211 Genève 4, SUISSE
email: brunet@cui.unige.ch

Abstract:

We present a constructive approach to the realization of a parallel system for the vision processing. This approach follows the classical software engineering principles: analysis, specification, design and implementation. The language CO-OPN (Concurrent Object Oriented Petri Net) is used to discribe the specification and the design of such system, while the implementation has been performed on a transputer machine. The writting of the CO-OPN specifications has been supported by the SANDS environement (Structured Algebraic Nets Development System) which also allow to simulate the behavior of the modelled system in an abstract way. A module, the main piece of the vision system, is presented that can be used for composing a computer vision architecture exploiting asynchronous data flows. This module is general enough to be duplicated and interconnected into a uniform graph. A module works by grouping asynchronous pieces of information coming through a channel, which delays them according to their relevance to the task of the module. The traditional separation between quantitative and symbolic processings vanishes since the module allows them to be implemented in a uniform way. The abstract specification of such principle is proposed, followed by a transformation refining this abstract specification into a concrete specification by introducing a distributed algorithm The concrete specification serves as the reference for the implementation on the distributed machine.

Keywords: System specification and design using algebraic Petri nets, Implementation on parallel machine, Asynchronous image and vision processing.

1. This research is partially supported by the «Fonds National Suisse de la Recherche Scientifique», program SPP IF 5003-034295 and FNRS 21-32'286.91

I Introduction

The main purpose of this paper is to introduce, through the case study of a new class of general approach of vision processing, the use formal techniques to model complex concurrent systems using abstract specification. Moreover, we want to show how to derive implementations on parallel machines by specification transformation. The implementation is derived from the abstract specification by introducing concrete aspects of the realization, for instance the algorithms used to distribute the global knowledge of time, producing a concrete specification. The CO-OPN (Concurrent Object Oriented Petri Nets) language is used to model both specifications, the abstract one and the concrete one. This example proves the wide spectrum of application modelling of this specification language. CO-OPN is supported by a complete environement called SANDS (Structured Algebraic Net Development System) including a graphical editor, a compiler and a simulator. The concrete specification is used for producing the code for the parallel machine, in our case a transputer machine.

The idea of developing a vision system in a modular fashion has already been raised by many authors [1]. Anatomical, physiological and psychological studies show that despite separation and specialization of visual processing, common underlying principles and mechanisms seem to apply. Modularity is also well suited to computer science, as a mean to decompose a large problem into smaller ones. Moreover if modules are casted from the same mold, reusability can significantly save development time.

We therefore attempt to define and formalize an architecture made of similar modules for solving the vision problem of indexing. The goal of this work is not to propose a universal vision system, but rather to investigate promising concepts. One of these is the active use of time [8] in desynchronizing data in a specific way in order to improve the various processings. A second concept is to group pieces of information in more and more elaborated ones [1].

The vision principles lead naturally to a parallel system realization, specified using the CO-OPN language, in which components of the vision system can be implemented on separate processors. The time is logically used to control both direction of the information flow. It is expressed in the first abstract specification as an universal reference. Continous components of the abstract vision model are discretized into an user defined small amount of interaction zones.

The implementation must incorporates more concrete features than the abstract specification, for instance in order to eliminate the explicit reference of time. A distributed algorithm has been defined that distribute the time reference, preserving the logical causallity between the forward and the backward flow of information.

This paper starts with a description of the principles of the asynchronous vision system, continues with a description of the components of the first abstract specification of the vision system, then we introduce the concrete aspects of the implemented algorithms, we show that the concrete specification can be obtained by transforming the abstract specification and we conclude after having presented the key points of the implementation on a transputer machine.

II Asynchronous Visual Processing

II.1 Motivations

II.1.1 Computer vision context

We are addressing the computer vision problem of primal indexing [7] amongst a known collection of man made objects in a cluttered environment. The work presented here only concerns the data processing aspect, and not the reasoning part. A more detailled presentation of the vision aspects can be found in [6]. A key point is that a module does not have to be perfect by itself, but that it has to provide useful information for the indexing task. To this end data at any stage of processing are represented in a symbolic, uniform manner. The architecture is a graph of modules, each module searching for its proper sources of information thus realizing connections to others. Although many directions of this work are inspired from the knowledge of the human visual system, the aim is however to propose new concepts for our vision problem rather than to strictly adhere to anthropomorphism.

II.1.2 Using time

The concept of *asynchronism* [8] consists in delaying information with different priorities instead of simultaneously sending the totality of the information. Therefore a static image, as in our case, leads to time varying flows of data. Priorities between information differ for each module. Computing resources are thus allocated to data that are thought to be the most relevant for the task at hand. This manner of proceeding allows earlier pieces of information to constrain later ones, either by reinforcement or by inhibition. This is done via a special case of feedbacks that spread constraints backwards. Besides, starting with the most relevant information renders possible for the system to reach its solution even without having processed all the information, making it more efficient: a vital characteristic of visual indexing.

II.1.3 Uniform architecture

All modules are instantiated from the same mold, and they exhibit the same basic functionalities, being therefore intrinsically similar. In computer science, splitting a task into smaller ones has great advantages, and the reusability of a component is very practical. Once a prototype of a module is tested, as many new ones as needed can be generated. The effort is then spent only in their connections and the specification of their proper parameters. This could lead to the design of a prototyping platform to test vision architectures with a high level of abstraction. In this case, if a module can be implemented on a MIMD computer, actual powerful prototypes could be built and tested.

II.2 Architectural overview

An example is shown in figure 1, applied to the problem of extracting lines and corners from achromatic images. It is a graph whose nodes are made of processors and whose arcs are channels for interchanging data between processors. A *module* encompasses one *grouping processor* and all the *channels* connected to its left side (a more thorough description is the subject of Section II.3). Processors translate incoming data into new ones with different dimensions, for example from points to line segments. Data flows between processors contain both feedforward and feedback information. At this stage it is possible to see that processors work concurrently according to the stream of data.

In figure 1, the analysis of the image begins in the *On* and *Off* modules. The *Curves* and *Segments* modules group On and Off responses into elementary curve and line segments respectively. These two modules try to group pairs of tokens from the On and Off channel if they could be curves and lines belonging to the objects. The *Lines* module groups colinear line segments to produce lines with any length, and the *Corners* module uses the curve and line segments to produce corners. This example can be extended to more evolved groupings, for example to group two set of parallel lines into a parallelogram.

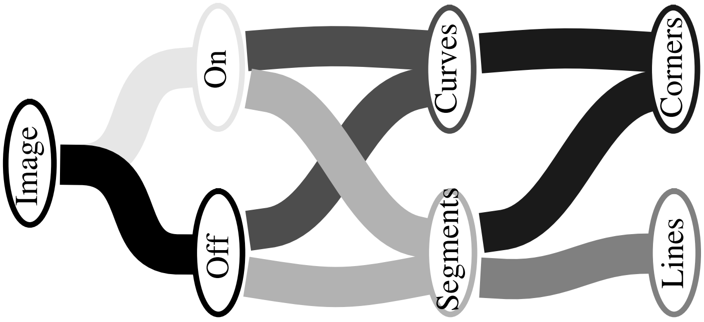


Fig. 1: A simple example made of six modules: one module (in a given grey-level) contains one processor (ellipse) and at least one channel (ribbon) to connect processors together.

II.3 Definition of a single module

II.3.1 General description

A module is the basic component of the vision architecture. It receives data coming from lower (i.e. leftwards in all the figures) modules, and then processes them to produce more abstract entities. In other words, modules transform data via a grouping process. Information is assumed to be represented in a symbolic way even at the earliest stages, i.e. the image. Each symbol will be called a *token*. This guarantees uniformity in data representations and processings.

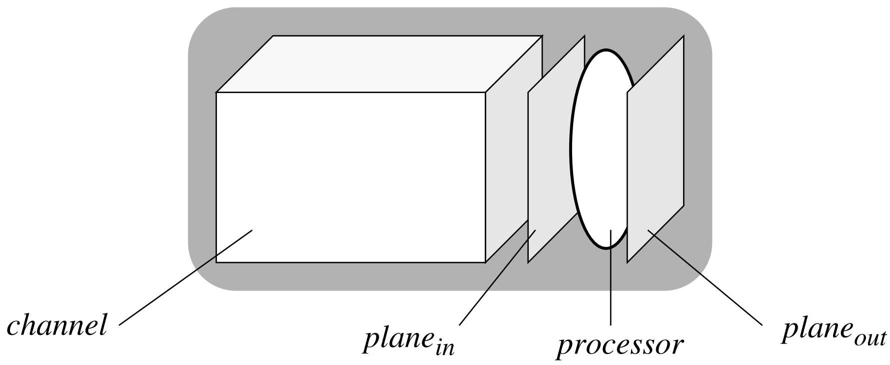


Fig. 2: A prototype of a module is composed of: a) input channel conveying data; b) a plane $plane_{in}$ receiving them; c) a processor and d) a plane $plane_{out}$ for storing the results.

A *module* is described in figure 2. Tokens are first picked by the *channel* when available and then transferred to the $plane_{in}$, accumulating them. On the figure, the channel entry is on the left and the exit on the right. The *processor* consults the $plane_{in}$ and try to produce new groupings with this incomplete view. The resulting groupings are stored in the $plane_{out}$, from which in turn they can be copied by the following module. A module can have more than one channel if needed, in this case it has also as many $plane_{in}$ as channels. The key point here is in the active manner a channel conveys data, as explained in Section II.3.3.

II.3.2 Grouping processor

The goal of the processor (see figure 3) is to transform tokens into more abstract ones. Grouping tokens implies a reduction of the quantity of information transmitted through the channel together with a more accurate possible interpretation (the figure 3 shows three line segments grouped into a single line). If needed, the processor can handle more than one channel, dealing with heterogeneous information, but produces a homogeneous type of results.

The content of the $plane_{in}$ from which the processor receives data is varying in time, as data are not received simultaneously and once received have only a fixed lifetime before to disappear. This constrains the way the processor groups tokens, having to deal with an incomplete view of the scene at any given time. However, it allows contradictory tokens at the same location to be considered at different time, for example different orientations of line segments. It also reduces the number of tokens to be considered in the grouping process.

Successful groupings are placed in the output plane $plane_{out}$ where they can be copied by the followings modules.

Finally, the processor can give rise to feedbacks, either on its own to complete an imperfect grouping or after a request from a connected module. In the first case, when an incomplete grouping has been performed, the processor sends back a specific request for more information (f.ex. the grey line segment in figure 3 could be reinforced). In the second case, a connected module sends a request to the current module (f.ex. a request for a specific line to the module that groups line segments to lines). The module then tries to produce the appropriate grouping with the available data in $plane_{in}$, and if it fails to do so it can in turn generate some requests (f.ex. for some missing line segments).

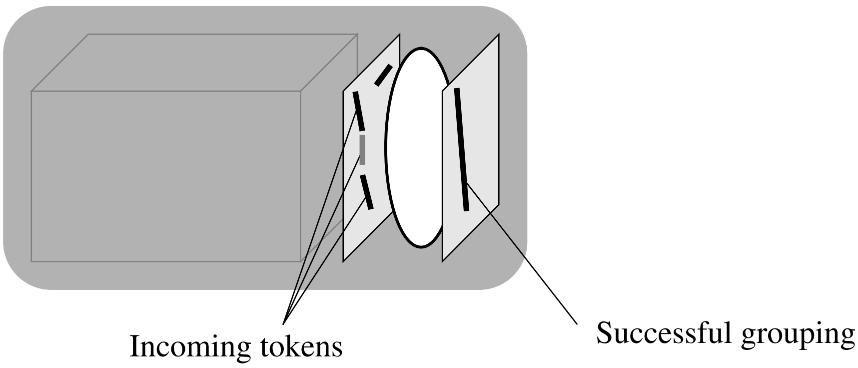


Fig. 3: The grouping processor transforms tokens from the $plane_{in}$, and stores outputs in the $plane_{out}$. Three line segments are grouped together to form a line, and the upper segment has not produced anything yet.

II.3.3 Channel

The channel vehicles tokens to the processor, and adds new functionalities to a classical architecture (see figure 4). Although a processor could work with all data simultaneously available, the channel *delays* tokens so that they arrive *asynchronously.* When a token is available at the input of the channel, it is picked and assigned an initial speed. The speeds are so allocated that the most useful tokens for the module have greater velocities and reach the exit before the others. The channel serves the processor, so that two channels can have different properties even though they are connected to the same source, as shown in figure 7.

The channel also conveys feedbacks data, which are tokens interpreted as requests or constraints. Feedbacks generated by the processor travel backwards (see figure 4, the white bullets), causing a reverse flow of tokens in the channel. Due to this particularity, the term *reversedfeed* seems more appropriate than feedback. The main reason to have reversedfeed is that tokens the reversedfeed is looking for may already be in the channel. So they can not be missed by the reversedfeed token while going backwards (f.ex. the bottom white bullet of figure 4), which would not be the case if the feedback was sent to the channel input.We will see in the specification that the continous aspect of the channel must be discretized into a sequence of *slices*, each slices modelling a region of the interaction part of the channel.

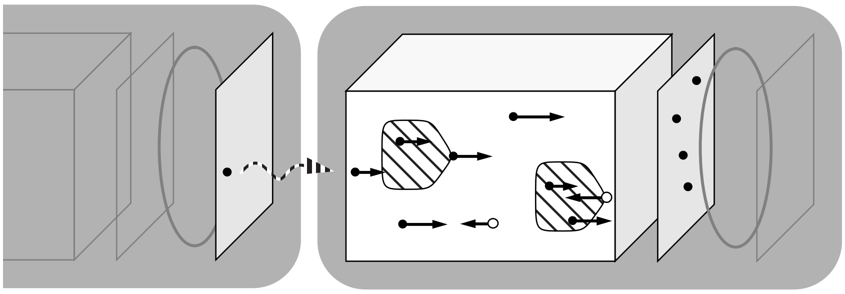


Fig. 4: The channel conveys tokens from their source (left plane) to their destination (right plane). Feedforward tokens (black bullets) and reversedfeed tokens (white bullets) have different directions of motion, and both can influence (hatched areas) other tokens in a spatio-temporal neighborhood.

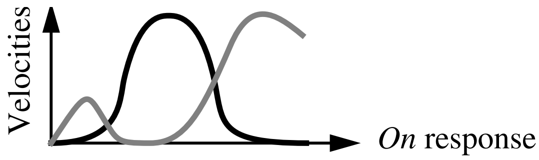


Fig. 5: Examples of possible velocities assigned to the *On* module outputs, for the *Curves* module (in grey) and the *Segments* module.

So far the channel can be thought of as a buffer delaying tokens. In reality it is more than this, as tokens can be modified while they stay in the channel by other tokens traveling in a neighborhood. Implicitly, reversedfeed tokens are sent to modify feedforward tokens, but also feedforward tokens with each others. As the most relevant tokens are travelling ahead of the others, feedforward tokens modify following ones, for example by adjusting their speed or even their values. In all cases, interactions are local in space and time, located around the current token position and in the wake of a faster token (see the hatched areas of figure 4).

III The specification language: CO-OPN

As the goal of this paper is not to introduce a specification language, the description of the specification language we will use, CO-OPN, will be brief and presented in an intuitive way. Interested readers can find complete and detailed description of this language in [9] and [11].

The project CO-OPN has been developed in order to propose a formal specification language for the design of large parallel systems. This aim is reached by using a modular specification language combined with Petri Nets. More precisely, the CO-OPN objects are interconnected modular entities described by algebraic nets. Tools have been built for supporting this formalism such that: a graphical editor, a compiler and a simulator for prototyping specification. This environment, called SANDS: Structured Algebraic Net Development System, is described in [10].

In fact, CO-OPN is based on two simple and elegant concepts, *algebraic data types* [14]and *Petri nets*, gathered into a more general model, *algebraic net*s [16]. Although algebraic nets are an important improvement of Petri nets, they are quite useless when facing to large problems. Therefore, some facilities has been included in the CO-OPN model, including:

- standard constructors on data types,
- Iterators on algebraically specified arrays,
- structuring capabilities.

Data types and operations

Both abstract data type and the theory of algebraic specification find their origin in the impossibility to manage and to develop large software system without methods and tools. Before going deeper in details we need to introduce some terminology.

- An *abstract data type* is the representation of a data structure coupled with a set of operations that manipulate this data structure independently (data type) of any particular implementation (abstract).
- The *algebraic* approach to specification considers a system in terms of abstract data types represented, in the mathematical discourse, by many-sorted algebra together with axioms or equations. An algebra is simply a domain provided with a set of operations. An algebraic specification consists of Spec = <S, F, X, AX> where S is the set of sort names, F the set of operation names, X the variables and AX the conditional axioms. In this kind of approach, a data type is called «*sort*», and a function is called an *operation*.

The axiomatic theory [14] is composed of the logical axioms, the equality axioms and the non- logical axioms including the user defined axioms. The set of deductive theorems DTh(Spec) consists of the set of equality deduced by the 'cut' and 'substitution' rules from the axiomatic theory, it is thesyntactic semantics of algebraic specifications.

Petri nets and algebraic nets

Petri nets are used to modelize and analyse discrete systems in which notions such as interacting components, concurrency and synchronization play a key role. Petri net modelization embeds both the structural aspect of the system and its evolution. Systems are modelled through the use of places and transitions.Their behavior is described using rules including pre and post conditions on the states. We use the following textual convention: `transition: condition => pre -> post`.

The use of tokens with an internal structure greatly increase the modelling power of Petri nets. Structured tokens give an insight into the way data structures evolve during the system running and many aspects of the modelization are described in a more accurate and concise manner. The evolution of the structured tokens are specified by rules of the underlying formalism used to represent these tokens. By its abstractness and soundness, algebraic specifications turn out to be well-suited for this purpose. The high-level nets obtained by describing structured tokens in the algebraic framework are called *algebraic nets* ([16]).

The CO-OPN formalism

A CO-OPN specification is composed of objects described by algebraic Petri nets, in which transitions are classified into two classes: the methods representing the external accesses and the internal transitions representing the reactions of the object, and a synchronization mechanism linking these nets together. These synchronisation mechanism are defined in order to keep object state encapsulation, allowing us to specify independently each object. The language CO-OPN has a textual and a graphical equivalent notation, both notations are used in the rest of the paper. Some examples for the use of CO-OPN can be found in [11], [12] and [13].

Properties of CO-OPN formalism.

Some properties are formalized within the CO-OPN framework [9].

- The equivalence for an algebra A by the notion of concurrent bisimulation.
- One object can be replaced in an object system by an other one which is equivalent, producing a new object system bisimilar to the initial one.
- If two different algebras observationnaly equivalent are used in the same object system then the two different object systems are equivalent by bisimulation.

These properties allow us to refine an abstract CO-OPN specification by progressively replacing abstract object and specification by more concrete ones.

IV The abstract specification of the modular vision system

This section shortly describes the main components of the vision system using CO-OPN objects and algebraic specifications. The figure 7 shows the generic objects used in the vision system and their relation to the instance object such as the `Timer`. One concrete specification, for example in figure 7, is composed with instances derived from these generic objects. Moreover, algebraic specification describing for instance the tokens of the vision system are based on primitive types: `Bool`, `Nat` not described here.

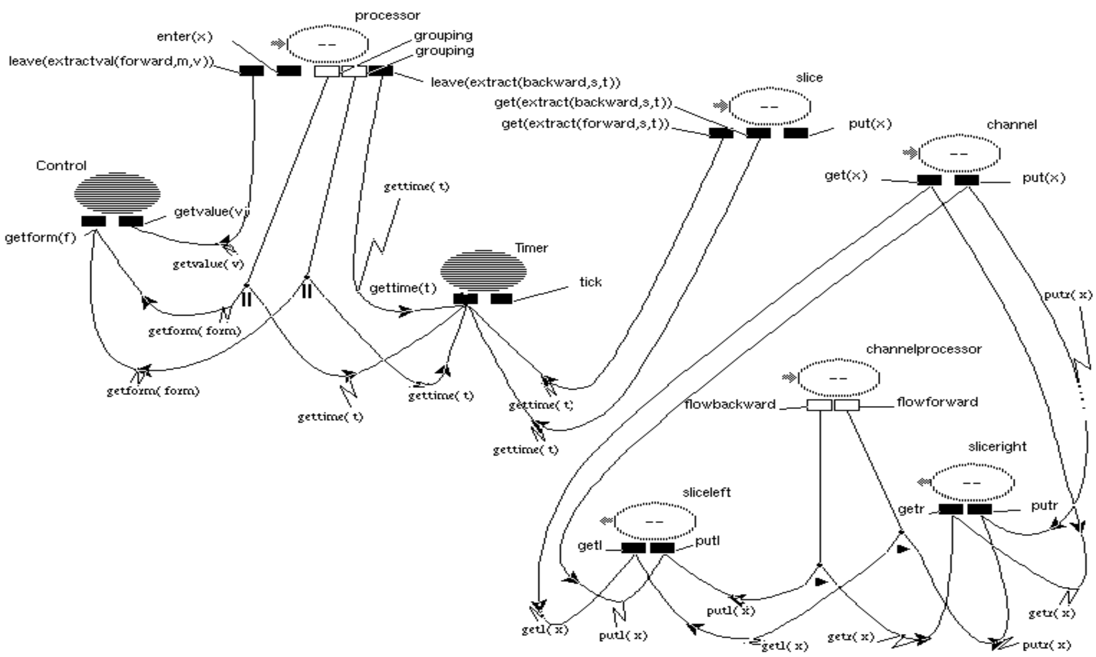


Fig. 6: The interconnection of the generic objects (white ovals) `slice`, `processor` and `channel` to the instance object (grey ovals) `Timer` and the parameter objects `sliceleft` or `sliceright`. The arcs between methods (black rectangles) represent the synchronization expressions (the kind of synchronization is annotated).

IV.1 The time description

The evolution of the system is based on a time model, each token evolves in the system according to the time evolutions. The absolute time in the system is modelled by values isomorphic to the natural numbers and operations such that the increment (`succ`) and the comparisons (`equal, le, lt, ge, gt`). Specification of the sort `time` in the module `Time`:

```
spec Time as Nat ;
renaming nat into time;
end
```

The reference of time is given by the object `Timer` (figure 7). This object provides two services, the time is incremented after activating the method `tick` and the current time value is given when activating the method `gettime`. Concurrent access to `gettime` and `tick` is allowed although the method `gettime` is not auto-concurrent and then sequentialize the access to the reference of time.

```
object Timer;
use Time,Bool;
places tickcount:[bool];
 currenttime:[time];
methods tick;
 gettime:time;
internal incr;
axioms
axtick: tick :: => -> tickcount([b]);
axgettime: gettime(t)::
                  => currenttime([t]) -> currenttime([t]);
axincr: incr::    => tickcount([b]) , currenttime([t])
                  -> currenttime([succ(t)]);
marking currenttime([zero]);
where b:bool; t:time;
end
```

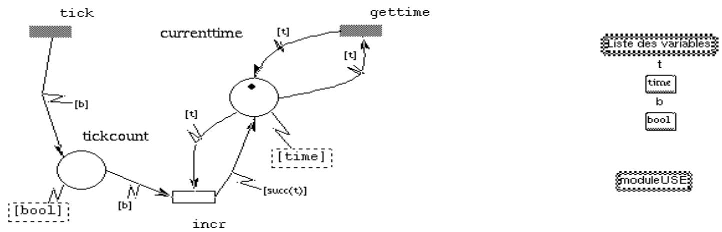


Fig. 7: The internal behavior of the Timer object graphically described. It corresponds to the above textual definition.

IV.2 The description of the tokens

IV.2.1 Informations contained in the token

The tokens flowing into the system must be identified by their speed, direction and contents. In our abstract modeling the content is not yet defined, we model it with a natural number defining how many interactions have been performed on it. The signature of `Tokens` is the following:

```
spec        Tokens;
use         Bool, Nat, Time;
sorts       direction, content, tokens;
```

```
generated by
```

The tokens are built with the following constructor:

```
desc:time, direction, content -> tokens;
```

The directions are of two types:

```
forward:    -> direction;
backward:   -> direction;
operations
```

The content of a token, sort '`content`', is in our simplified case a natural number:

```
ap: nat -> content;
```

The access to these data can be obtained through the use of the following functions:

```
time:       tokens-> time;
whichdirection:tokens-> direction;
```

The time modification on tokens (application dependant):

```
f: tokens -> tokens;
```

with the axioms

```
axioms
axtime: => time(desc(t,d,c)) = t;
axwhichdirection: => whichdirection(desc(t,d,c)) = d;
where
e:nat; t:time; d:direction; c:content;
end;
```

IV.2.2 Speed modeling

If we suppose that the channel in which the tokens are moving from its beginning to its end during an interval of time T has a physical length 'L', the speed of these tokens is 'S = L/T'. We use, to simplify the treatment of the speed of the tokens, time intervals instead of speeds. The speed of a token depends on its importance. The function '`f`' is activated when entering into each *slice* composing a *channel*, this function calculates how long time the token stay into the *slice*.

This operation satisfy the following lemma:

```
time(tokens) + Δt = time(f(tokens)) with Δt>0
```

IV.2.3 Token interaction

The tokens are interacting when they are in the same time slice, these interactions are modelled by the function `int`.

```
int:        tokens, tokens-> tokens;
```

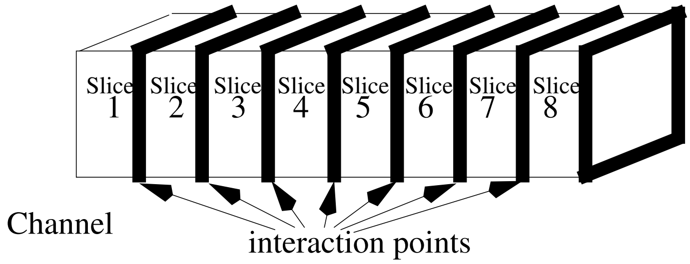
The interaction of one token with an another produces an other modified token, in the abstract modelling only the content of the token is changed.

The direction of the tokens can be modified:

```
reverse:    tokens-> tokens;
axreverse0:=> reverse(desc(t,forward,c)) = desc(t,backward,c);
axreverse1:=> reverse(desc(t,backward,c)) = desc(t,forward,c);
```

IV.3 The channel description

A channel looks like a communication buffer into a producer consumer system. In fact its internal protocol is much more complex. In order to be able to model a channel with a sequence of interactions between the tokens, the continous interactions are modelled by discrete ones. A channel is decomposed into slices, each slice representing one point of interaction. The interaction, in a slice, is made between the outgoing forward token and the other tokens staying into the slice. The result of an interaction is a token staying into the slice.



Moreover, the channels are bidirectionnal, the tokens can move in two directions. The interaction of the backward token is made when the token enter into the slice, this means that the point of interaction is the right side of each slice.

IV.3.1 General view of a channel

The view of a channel composed of three slices connected via two channel processors is given in figure 8. We use a naming convention such that : the slices are identified by the alphabetic channel identifier and the slice number, the channel processor are identified by their left and right slices.

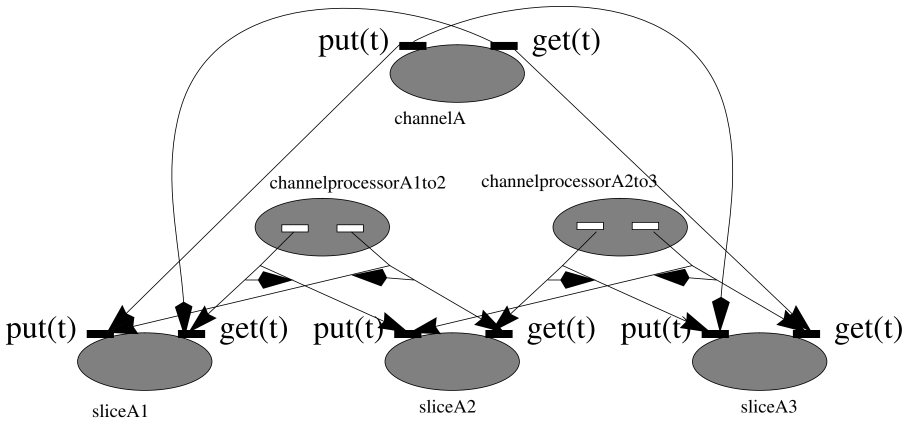


Fig. 8: The interconnection of slicesA1 to sliceA3 through channelprocessorA1to2 and A2to3 composing the channel.

The channel is defined by instantiating the generic objects channel, slice and channelprocessor. The renaming mechanism allows to eliminate the collision of identifier.

```
OBJECT S1channelA AS channel(S1sliceA1,S1sliceA3);
RENAMING
putl INTO putA1 ; getl INTO getA1 ;
putr INTO putA3 ; getr INTO getA3 ;
put INTO putA ; get INTO getA ;
USE S1sliceA1, S1sliceA3;
END
```

```
OBJECT S1channelprocessorA1to2 AS
                         channelprocessor(S1sliceA1,S1sliceA2);
RENAMING
putl INTO putA1 ; getl INTO getA1 ;
putr INTO putA2 ; getr INTO getA2 ;
USE S1sliceA1, S1sliceA2;
END

OBJECT S1channelprocessorA2to3 AS
...

OBJECT S1sliceA1 AS slice;
RENAMING put INTO putA1 ; get INTO getA1 ;
END

OBJECT S1sliceA2 AS slice;
...
OBJECT S1sliceA3 AS slice;
...
```

IV.3.2 The transmitted values

A sort describing the messages transmitted is given which is just an encapsulation of the `tokens` sort.

```
spec         Messages;
use          Tokens;
sorts        message;
generated by
             Datas: tokens -> message;
```

IV.3.3 The description of the channel

A channel is build over a sequence of slices connected by the channel processors. A channel must only know the first and the last slice of this sequence which are given by the parameter object `sliceleft` and `sliceright`.

```
generic object channel(sliceleft, sliceright);
use Messages, sliceleft, sliceright;
methods put:message; get:message;
axioms
ax1put: put(x) with putl(x)::whichdirection(x) = forward => -> ;
ax2put: put(x) with putr(x)::whichdirection(x) = backward =>->;

ax1get: get(x) with getr(x)::whichdirection(x) = forward => -> ;
ax2get: get(x) with getl(x)::whichdirection(x) = backward =>->;
where x:message;
end
```

IV.3.4 The abstract slice description

The slices are abstractly described as an entity with two point of access the `put` and the `get` methods, used to transmit and extract the tokens to the slices. The parameter objects are used to model the possible slices used when instantiating the system of `channel`, `slices` and `channel processors` objects composing a `channel`.

```
parameter object sliceleft;
use Messages;
methods putl:message; getl:message;
end
```

```
parameter object sliceright;
use Messages;
methods putr:message; getr:message;
end
```

IV.3.5 The channel processor

The channel processors are simple objects used to convey tokens from the left to the right or from the right to the left according to the token directions. To do this, the internal transition `flow-forward`, `flowbackward` are defined. These actions are immediate when the state of the left or right slices allow to fire the `getl` or `getr` methods, the methods `putl` and `putr` being always firable.

```
generic object channelprocessor(sliceleft, sliceright);
use Messages, sliceleft, sliceright;
internal flowforward, flowbackward;
axioms
ax1forw: flowforward with getl(x) .. (putr(x)) ::
 whichdirection(x) = forward => -> ;
ax1backw: flowbackward with getr(x) .. (putl(x)) ::
 whichdirection(x) = backward => -> ;
where x:message;
end
```

IV.3.6 The set of tokens

The sort '`setoftoken`' is used to model the content of the slices, which are multi-set of tokens. The values are generated by the constructors `emptyset` and `u`.

```
spec Setoftoken ;
use Time, Tokens, Nat,Bool, Controldata;
sorts setoftoken;
generated by
 emptyset: -> setoftoken;
 u: setoftoken, tokens -> setoftoken;
operations
```

Various operations must be performed on the tokens, the 'selection' operations and the 'modification' functions. The selection of one tokens from a set of tokens is modelled with three functions, one indicating if the extraction is possible, one giving the value extracted and one suppressing the extracted value of the set of token. Two kind of selections are available:

- selection by time and direction:
```
extractable:direction, setoftoken, time -> bool;
extract:direction, setoftoken, time-> tokens;
reduce:direction, setoftoken, time -> setoftoken;
```

- selection by direction and value:
```
available: direction, setoftoken, tokenvalues -> bool;
extractval: direction, setoftoken, tokenvalues -> tokens;
reduceval: direction, setoftoken, tokenvalues -> setoftoken;
```

The modification function make an interaction between a token and a set of token or group a token with a set of token. The result is a modified set of token.

```
interact:setoftoken, tokens -> setoftoken;
group: setoftoken, tokens -> setoftoken;
```

The axioms of these functions are ommited they can be found in the technical report [2].

IV.3.7 The slice description

We explain here the detail of the slice object, which constitute the main component of a channel.

The signature of the slices corresponds to the signature of the parameter slice already described. The state of the slice is modeled by a place '`cell`' containing in a unique multiset structure all the tokens flowing through the slice. This choice allows to access to all tokens with algebraic functions and then simplify the operation of interactions. It would be difficult to perform interactions if the place '`cell`' was modeled using multiple Petri net tokens. Although we impose to access sequentially to the '`cell`' content, it seems that this choice is satisfying.

The temporal relations (speed modeling) are ensured by the access to the object `Timer`. The tokens can exit the slice when their own time is greater than the current time. When exiting is performed the interaction between the tokens are computed. It must be noted that the actual model do not differentiate the tokens with the expired time of exiting from the others when computing interactions.

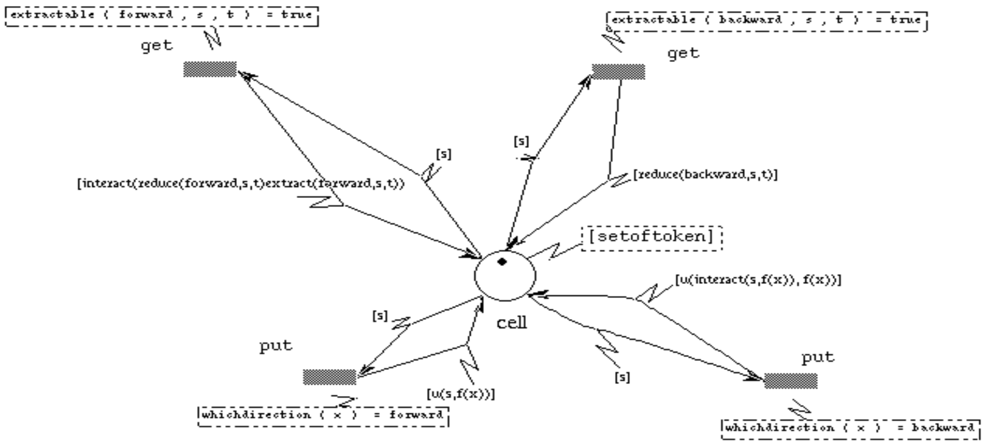


Fig. 9: Internal behavior of the object slice.

```
generic object slice ;
use Setoftoken, Tokens,Messages, Timer,Time,Bool;
places cell:[setoftoken];
methods put:tokens; get:tokens;
axioms
ax0put: put(Datas(x)) ::
 whichdirection(x) = backward =>
 cell([s]) -> cell([u(interact(s,f(x)),f(x))]) ;
ax1put: put(Datas(x)) ::
 whichdirection(x) = forward => cell([s]) -> cell([u(s,f(x))]) ;
ax0get: get(Datas(extract(backward,s,t))) with gettime(t) ::
 extractable(backward,s,t) = true =>
 cell([s]) -> cell([reduce(backward,s,t)]) ;
ax1get: get(Datas(extract(forward,s,t))) with gettime(t) ::
 extractable(forward,s,t) = true =>
 cell([s]) ->
cell([interact(reduce(forward,s,t),extract(forward,s,t))]);
marking cell([emptyset]);
where s:setoftoken; x:tokens; t:time;
```

```
end
```

IV.4 Simulation on the SANDS environment

The development of such a large application needs the assistance of a tool including simulation, analysis and debugging facilicilities. The CO-OPN language is supported by the SANDS environement allowing to perform such kind of activities. Furthermore, due to the modular nature of the CO-OPN formalism it is possible to partially explore and analyze, module by module, the behavior of an application. In the simulation performed in SANDS (figure 10) by interpreting the generated Prolog code, the Nat specification is implemented using the Prolog integers.

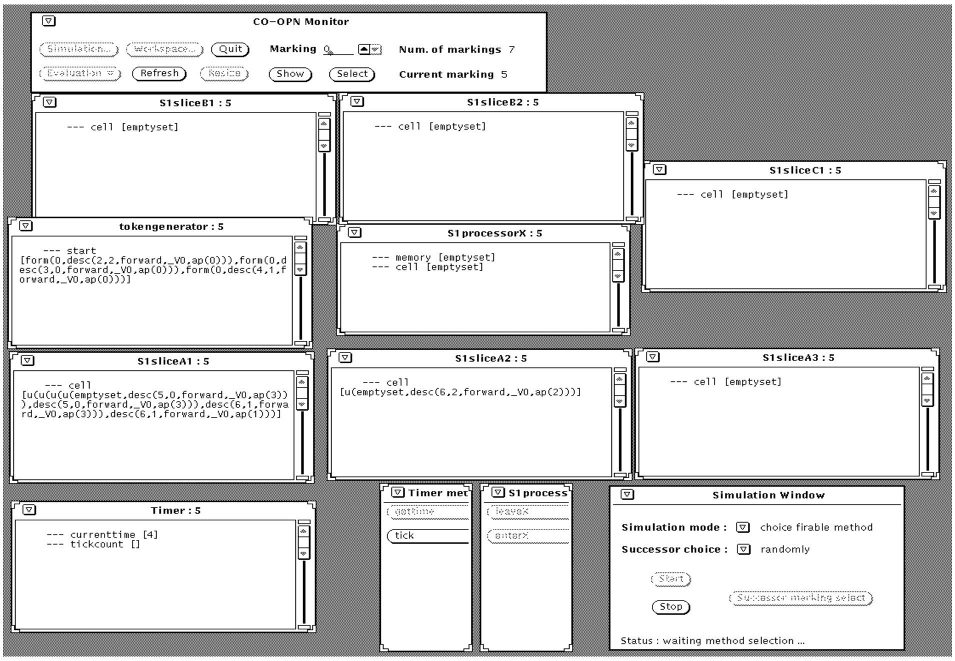


Fig. 10: Simulator windows showing the state of the object's application (S1sliceXX windows), the firable events (Timer and S1processor windows) and the possible control of the simulator tool (CO-OPN monitor and simulation Window).

V The distributed algorithm

V.1 Construction of the distributed coherent time algorithm

We are interrested to elaborate a stable global property, the causal time consistency, of a set of distributed processes through the definition of a distributed algorithm. This property must be maintained during the whole life of the application which end when no communication can occur.

V.2 Hypothesis on the application

The application is made of n processes that can be either grouping processors or slices. The slices collect tokens and form a chain. The grouping processors can have several slices as left or right neighbours, they includes also tokens. The processes communicate through message passing along communication channels. We suppose that the communication mecanism is without duplication, loss and corruption of messages. The channels are bi-directionnal and they can have a fifo behavior.

Any process of the system identified by the number i knows its left and right neighbors. We consider, in the following, that the left neighbors will have the numbers lower than i and the right neighbors the numbers greater than i. The slice case is a simpler case in which its left and rightneighbor are unique. It must be noted that the distributed algorithm must be general enough in order to take into account any kind of topology.

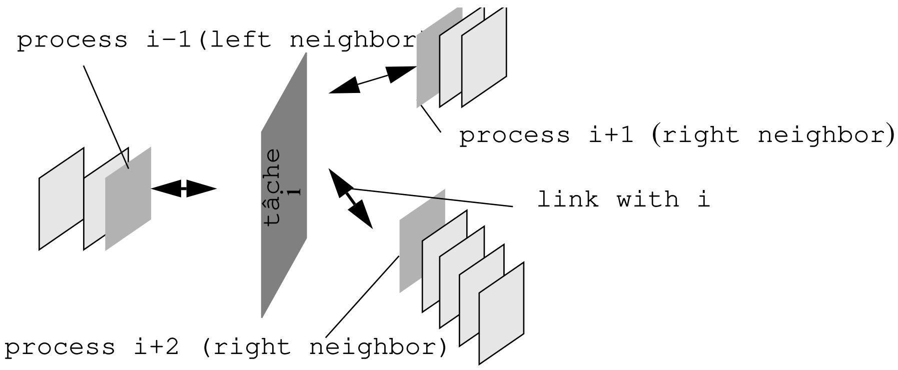


Fig. 11: A grouping processor with one left neighbor slice and two right neighbor slices.

V.3 Structure of the solution

We use the wave principle [17] to find a unique global time to the nodes of the system, it is the minimum of next moving time of the tokens of the slices and the grouping processors. Two waves are necessary to compute this minimum. The first one, the analysis wave, collecting the information of the processes from the left to the right and the second one, the distributing wave, giving the result to all processes in the opposite direction (figure 14).

Starting with the unstructured topology of the vision system we build a system with two new processes: the `Feeder` on the left and the `Consumer` on the right of the oriented topology (figure 12). All processes without left neighbor are connected to the Feeder and all processes without right neighbors are connected to the Consumer.

The analysis wave is initiated by the Feeder and arrives to the Consumer after visiting all the nodes. The message transmitted to the right neighbor contains the minimum token time collected during all previous visits. The distributing wave is initiated by the Consumer after receiving all the message comming from the analysis wave. It gives to all processes the minimum time collected by the analysis wave.

.

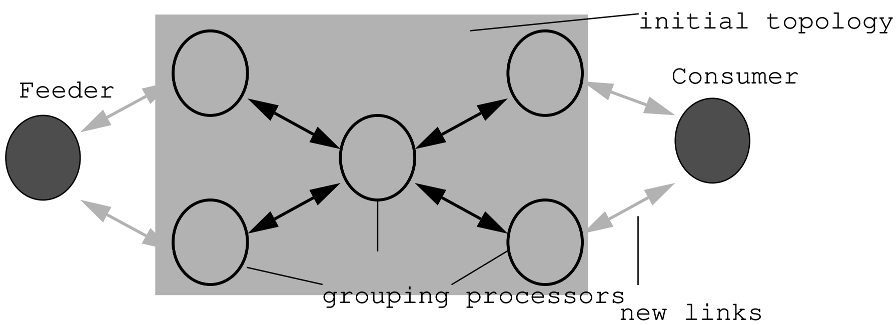


Fig. 12.: Vision system topology and the added processes: the Feeder and the Consumer

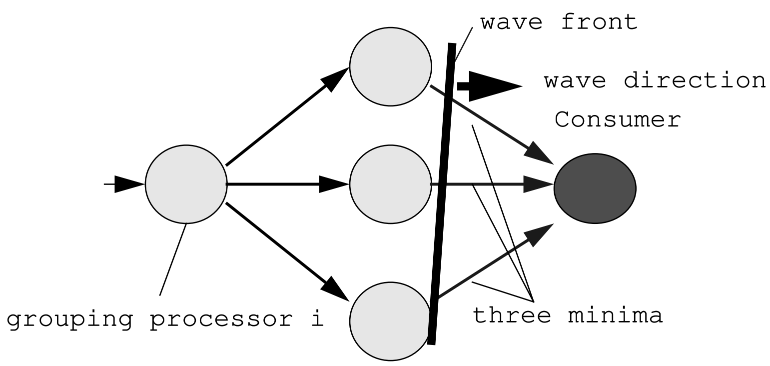


Fig. 13: The consumer receives the minima comming from its left neighbors.

It must be noted that launching the analysis and distributing waves is a repetitive process which ends when the minimum does not increase.

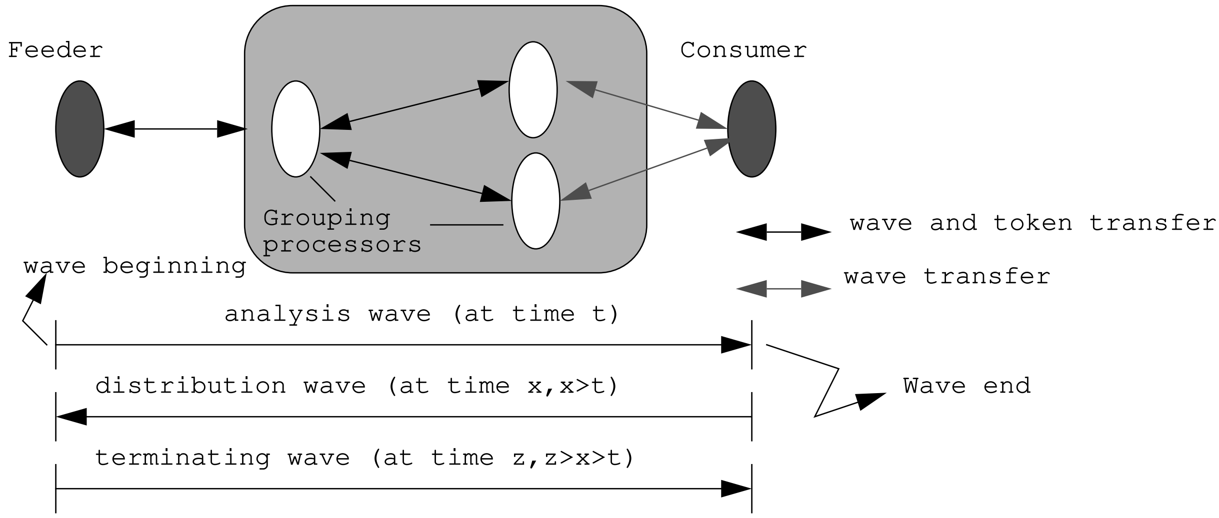


Fig. 14: The three kind of waves and their directions.

V.4 Conditions on the token transmissions

In the wave algorithm, the position into the wave determines different states of the processes composing the vision system. Some rules can be defined on the process states, denoted with colors, during the movement of the front wave, in order to make possible the token transmissions.

A variable $\mathtt{color}_i$ is defined for each process i, it takes one of the following colors:

- `.Red` the process is on the left of the analysis or distribution wave front and cannot transmit any tokens.
- `.Green` the process is on the right of the analysis or distribution wave front and has tokens to transmit.
- `.White` the process is on the right of the analysis or distribution wave front and has no

tokens to transmit.

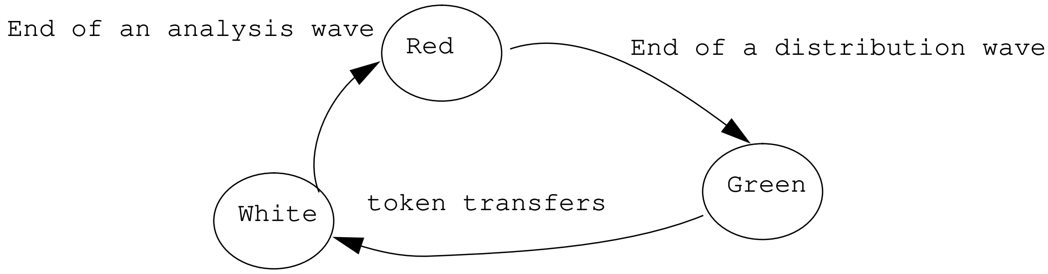


Fig. 15: The automaton of the process states during the vision algorithm.

Each process must manage its color in relation to the position of the wave front (figure 15) and its ability to transfer tokens to its direct neighbors. If its color is `Green,` its next possible color is `White` when its tokens are transmitted, and so on.

The fact that these conditions are imposed guarantee that no wrong minimum can be found by the `Consumer` if no tokens are sent when the analysis wave front is on the right of the process. Indeed, when the `grouping processor`$_i$ send a token, with time equal to the global minimum, to the `grouping processor`$_{i-1}$, the minimum find by the `Consumer` is the old time of the token sent by the `grouping processor`$_i$. This is possible if the analysis wave front goes through the `grouping processor`$_i$.

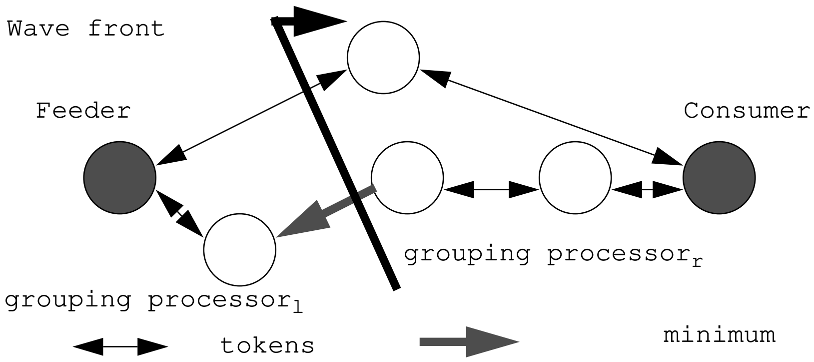


Fig. 16: Detection of a wrong minimum because the token is sent to its neighbor when the wave move through the process.

In addition to its own color, a process must know its neighbors colors. The analysis wave front must not move forward if the processes to be visited are `White`. The validity of this constraints could be easily proved by the following counterexample: suppose that the global minimum is in the `grouping processor`$_r$ and that it is sent to the `grouping processor`$_l$, at the same moment this last process send the wave to the `grouping processor`$_r$. This situation corresponds to the previously described case of the figure 16.

The behavior of the `slices` is a restriction of that of the `grouping processors`. The `slices` have only one left or right neighbor. This is the reason why slices are not mentionned into the algorithms.

The correction of the algorithms described in this section are explained in the technical report [2].

VI Specification of the Distributed Algorithm

We will show, in the following, how to translate the original abstract specification into a concrete specification. The concrete specification will include the aspects needed by an implementation on parallel distributed machines, that means, no assumption on a global reference of time and a mapping between the objects and execution processors. The abstract algorithm was explained before in an informal way, we will now explain how to introduce this algorithm into the initial formal specification. The global structure of the specification and the signatures of all components will not be changed, nevertheless the internal behavior of the slices and processors will include the new algorithms. In order to deal with synchronization different kinds of modifications will be made:

- the inclusion of local behavior modelling the phases of the algorithm (with state coloration of the nodes).
- an encapsulation of messages into new sorts modelling the two waves (analysis and distribution wave).
- the inclusion of the analysis wave initiating algorithm into the 'Feeder' object.
- the inclusion of the analysis wave terminating algorithm into the 'Consumer' object.
- the inclusion of the distribution wave initiating algorithm into the 'Consumer' object.
- the inclusion of the distribution wave terminating algorithm into the 'Feeder' object.

For instance the sort message is extended with the new kind of messages managing the distributed algorithm.

```
WaveAnalysis: time -> message;
WaveDistribution: time -> message;
WaveEnd: time -> message;
NeighborColor: time -> message;
```

These generators have corresponding operations. Due to the lack of space it is not possible to explain more on the transformation techniques, in [2] the treatment of the vision problem is fully described while in [12] CO-OPN transformation language and tools are also explain.

VII Implementation on Parallel Machine

This part was a direct translation of the concrete specification which has a model of synchronous communicating processes, into the parrallel C language (a C compiler with some communication primitives) availlable on our network of transputer (VOLVOX machine). The placement of the processes on the processors can be easilly changed in our implementation through the modification of the placement file. Nevertheless, more information must be added concerning the problem that is solved, this is a vision problem and consequently not presented here.

We realize a concrete example, solving the problem of line extraction built with five modules and nine slices, and find that the most efficient placement was to exploit the pipeline principles. Another concrete example showing possible parallelism between modules is described in [2].

VIII Conclusion

This paper has presented the use of formal methods to specify a concurrent problem which is general enough to cover a large class of vision application. Both abstract specification and design phase have been presented. In the design phase an appropriate algorithm has been developped to solve the problem of time reference. We have suggested that it is possible to embed the distributed algorithm into the abstract in a systematic way by transforming the abstract specification. Moreover, the use of our SANDS tool was a great benefit in the process of specification and design.

On the vision side, we have presented the concept of a prototypical module that can be used at any level of abstraction so as to compose a uniform vision architecture. The module is made of a processor and a transmission channel which connects it to other modules. The processor receives tokens and by a grouping mechanism produces new ones. The channel asynchronously feeds earlier in time the processor with relevant data. The channel also modifies travelling tokens according to its current content of feedforward and reversedfeed tokens. A module can be used at any level of processing, from pixels to more abstract representations. We hope these developments will confirm the asynchronous principle developed in the modular approach presented here.

We plan to continue this work by exploring the performance aspects of the realizations on various vision applications and to systematize the transformation principles [15], as it is already made in the case of parallel algorithm design [12], [13].

Acknowledgments

We are especially grateful to Jean-Marc Bost, Mathieu Buffo, Ruggero Milanese and Pascal Racloz for their fruitful discussions and advice and to the numerous satellites revolving around this work.

Bibliography

[1] D. Marr, *Vision*, W.H. Freeman and Co., 1982.
[2] D. Monteiro, *Etude, spécification et implantation sur machine répartie d'un système modulaire de vision*, Diplôme en Informatique, Université de Genève, Novembre 1993.
[3] E.N. Willmer, Anatomy of the visual system, in *The senses*, pp. 35-45, Cambridge University Press, 1982.
[4] D. Mumford, On the computational architecture of the neocortex, Biological cybernetics, vol. 65, pp. 135-145, 1991.
[5] M. Livingstone, Segregation of form, color, movement, and depth processing in the visual system: anatomy, physiology, art, and illusion, in *Vision and the brain: the organization of the central visual system*, vol. 67, pp. 119-138, Raven Press, 1990.
[6] D. Brunet and T. Pun, *A modular architecture for asynchronous visual processing*, Swiss Vision, Sept. 1993.
[7] J.-M. Bost, Discriminating agents for visual indexing, Proc. 3rd SGAICO Meeting (Swiss Group for Artificial Intelligence and Cognitive Science), Biel-Bienne, Oct. 10-11, 1991.
[8] J.-M. Bost, R. Milanese and T. Pun, Transforming an image into a data-flow for visual indexing, Proc. Int. Conf. on Image Analysis and Processing, Bari, Italy, Sept. 20-22, 1993.
[9] D. Buchs, N. Guelfi: CO-OPN: A Concurrent Object Oriented Petri Net Approach. *12th International Conference on theory and application of Petri Nets*, Aahrus, pp. 432-454, 1991.
[10] D. Buchs, J. Flumet, P. Racloz: *SANDS/CO-OPN Structured Algebraic Net Development System*. Petri Nets Tool Presentation 1993 Chicago.
[11] Buchs D. and Guelfi N., *Distributed System Specification using CO-OPN*, 1992, IEEE 3rd Workshop on Future Trends of Distributed Computing Systems, Taipei.
[12] Buffo M., *Transformations : théorie et outils*, 1993, Cahier du CUI No 78, Université de Genève.
[13] Buffo M., *Collection de Transformations*, 1993, Cahier du CUI N^{o} 79, Université de Genève.
[14] Ehrig h. and Mahr B., *Fundamentals of algebraic specification 1: equations and initial semantics*, 1985, EATCS Monographs, Springler Verlag.
[15] Krieg-Brückner, *PROgram development by SPECification and TRAnsformation*, article in TSI vol 9 n 2.
[16] Reisig W., *Petri nets and algebraic specifications*, 1991, in Theoretical computer science 80.
[17] Raynal M., Networks and distributed computation: concepts tools and algorithms, London, North Oxford academic, 1987.