Developing parallel programs based on rewriting techniques[1]

Mathieu BUFFO (*), Didier BUCHS (+), José ROLIM (*)

(*) : ***Université de Genève***
24, rue du général Dufour
1211 Genève 4, SUISSE
email: {buffo,rolim}@cui.unige.ch

(+) : ***Swiss Federal Institute of Technology***
1015 Lausanne
SWITZERLAND
email: buchs@di.epfl.ch

Abstract

Starting with a formal problem specification, we show how to progressively achieve a parallel distributed solution, using convenient transformations of specifications. In order to achieve a solution, we use rewriting techniques applied in the context of transformations. This paper explains the methodology, points out its advantages and its requests. We first, introduce the specification language CO-OPN used to express both abstract specification and implementation. CO-OPN is a structured version of algebraic specification and Petri nets in which adequate notions of equivalence can be formalized to handle the concept of correct transformations. Then we introduce the general technique of applying transformations to get specifications progressively closer to the implementation level. These transformations are classified into two levels, the high level transformation performing abstract modification of algorithm and the low level transformations which are applicable on the specification closer to the implementation. Finally, a concrete example will show how the simple «permutation sort» can be transformed quite automatically in a distributed implementation of a «quick sort» algorithm.

1. Introduction

This paper describes results obtained in a project called TSPP (for «Transformations de Spécifications orientées vers le Parallélisme et le Probabilisme»). The objective of this project is to give a framework for defining abstractly a problem, finding a first parallel solution which may be is not the most efficient one, and then refining this solution by exploiting parallelism and distributive features.

To describe formally a parallel algorithm, we shall use modelling techniques which in our case demand the following features [1]:

- The necessity of defining formal languages in order to capture the meaning of a problem, these languages must have a precise definition and must be able to express a problem in a very abstract way, saying what the problem should do but not how it could do it.

1. This research is supported by the «Fonds National Suisse de la Recherche Scientifique», program SPP IF 5003-034295

- The necessity of defining a formal framework of program derivation from the abstract specification to the concrete algorithm.

Our answers to these two requirements are the choice of a suitable specification language CO-OPN («Concurrent Object-Oriented Petri Nets») as specification language, and the use of transformation techniques to refine specifications. Transformation is a term regrouping a set of rewriting techniques, allowing the conversion between a specification and another one. In this context, the main idea is to use transformations as simple as possible, in order to easily prove the equivalence between the two specifications. We use a succession of transformations to progressively achieve an implementation [4], which is guaranteed to be equivalent to the initial specification. In brief, we want to achieve a parallel distributed solution from a specification written in CO-OPN, using suitable transformations.

Ambitious projects on specification transformation has been carried out, such as the CIP project of Munich [8][9] or the PROSPECTRA project [12]. Even if these projects had produced many interesting results, it can be concluded that it is not possible to achieve an implementation only using specification transformations, because of the too large complexity of the transformation process. There are mainly two important differences between our project and the others who deal with transformations, notwithstanding the fact that we focuses especially on parallelism. The first one is that our aim is not necessarily to reach a final specification with all details of implementation. We concentrate our effort on finding first parallel solutions, which can be a good approximation to solve the original problem. Furthermore during the whole transformation phase, we stay in the context of a specification language, simpler to manage. This allow us to avoid the problems other projects have had in terms of manageable complexity of the transformation system.

We will first introduce the basic concepts needed to understand the aim of this paper. This includes an introduction to specifications formalism, a brief discussion about algebraic data types and Petri nets and about CO-OPN, and an explanation of what is a transformation.

Then, as an illustration, we will see how, starting with a description of «permutation sort», we can get a description of a parallel distributed «QuickSort» by applying -among others- the «divide to conquer» principle in the context of the transformations we need in this project.

2. The specification language: CO-OPN

We will now describe the formal context in which we will develop parallel distributed applications. As mentioned above, starting with a formal description of a problem, our goal is to achieve an implementation of a solution, in a secure, progressive and automatic way. In addition, we would like to introduce parallelism in this implementation. So we will first describe the chosen specification language CO-OPN, and then we will show how the implementation can be reached, using the transformation of specification concept.

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 [2] and [3].

In fact, CO-OPN is based on two simple and elegant concepts, *algebraic data types* and *Petri nets*, gathered into a more general model, *algebraic net*s. 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 and structuring capabilities.

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 ([11]) 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. The syntactic satisfaction relation $\vdash$ is defined as: Spec $\vdash$ t=t' $\Leftrightarrow$ t=t' $\in$ DTh(Spec). As an example of algebraic specification, this is a module generating natural numbers, and defining addition on these numbers:

```
spec Nat;
sorts nat;
generated by
     zero : -> nat;
     succ : nat -> nat;
operations
     add : nat,nat->nat;
axioms
          add(zero,x)=x;
          add(succ(x),y)=succ(add(x,y));
where x,y:nat;
end
```

Figure 1 : The specification Nat

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 modeling embeds both the structural aspect of the system and its evolution.

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 modeling 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* ([13]).

A CO-OPN specification is composed of objects described by algebraic Petri nets 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. Some examples for the use of CO-OPN can be found in [3], [4] and [5].

Some properties are formalized within the CO-OPN framework.

- 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.

3. Transformations

Starting from the first abstract specification, the standard way to achieve an implementation is to get trough a programming phase, in which software developers often introduce many errors (with respect to specification). As an alternative way to this classical development method, we can progressively transform the specification, step by step, in a way that it tends to an implementation. In such a context, it is easy to insure the correctness of the solution with respect to the specification: each step has to be correct, has to preserve the semantics of the specification. Such a single step is called «transformation». This method has three obvious advantages:

- It is easier to prove correctness (with respect to specification) of such a single step than to prove correctness of the whole result of the implementation.
- It is possible to do fine backtracking during the transformation phase. This leads to a convenient development method, using prototyping and reorientation, and to an easy possibility of readaptation (of the implementation) to a new execution platform, using new transformations (making the specification tending to the new platform).
- It keeps the same language during the whole phase, avoiding traduction errors.

In the figure below we propose to realize such a context, with all transformation rules stored in a so-called transformations library, and the development phase is controlled by an operator, which choose a new transformation T in the library or choose to terminate this process:

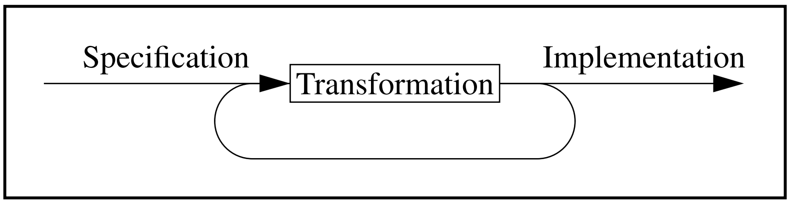

Figure 2 : Transformation system scheme

Transformations are described in a general and convenient way, through *transformation rules*. A transformation rule must describe how to pass from a specification to another one, in a general way. Therefore, in their operational description and semantics, transformation rules are mainly based on rewrite systems techniques, like pattern matching and variable substitution.

A transformation rule description consists of:

- a set of pattern that a specification must match to be handled by the transformation,
- a set of preconditions,
- a set of patterns to be added to the specification,
- a set of variables.

Given a specification, the (informal) operational semantics of such a transformation is:

- If the specification is not encompassed [10] into the matching-pattern, STOP.
- If not all (variable substituted) conditions are true, STOP.
- The new specification is the old one, where the matched pattern have been removed, and with the addition of (an evaluation of) the «adding» patterns.

Moreover, to expand (in a simple way) the power of our transformation descriptions, so called meta-operations [9][4]are provided in the method.

One of the main problems in software development is the difficulty to obtain correct implementations with respect to the specifications; either the whole implementations (programs) must be proved, with complex and painful techniques, or long and terrible debugging phases must be carried out.

In our context however transformations must preserve the semantics of the specification, to insure the adequacy of the final specification with respect to the initial one. In other word, transformations must preserve the defined equivalence of specifications. And remembering the fact that transformations are only a small step of the whole implementation process, it is in general easier to prove the result with this technique than with the classical one.

We said that a transformation is *correct* if the transformed specification is observationnaly equivalent to the original one. Likewise, we said that a transformation rule is correct if all generated transformations are correct. The use of correct transformation rules is useful; it avoids to prove at each step the equivalence with respect to the initial specification, providing that the rule was once proved to be correct, typically by a computer scientist.

In brief, our method leads to a «without proving, without debugging, but guaranteed correct» software development.

Transformation rules can be regrouped into subsets, according to their use [5]. Indeed, it is often judicious to respect an order in the application of transformations. For example, it is absurd to optimize the internal behavior of an object of a specification, and then replace this object by another one which is completely different! Moreover, to find a first algorithmic parallel solution is not the same kind of problem than distribute an implementation on exactly four processors. This leads us to categorize transformation rules into two main classes:

- The *high level* transformation rules are useful to obtain new specifications describing new algorithms, new resolution methods. For example, from a specification describing a sequential processing of a set of elements, an high level transformation rule leads to a set of independent, possibly parallel processings.
- The *low level* transformation rules are involved in simple, useful, repetitive tasks, allowing optimization or adaptation to an execution platform. These low level rules are themselves splited into two sub-classes: the «modifications of objects structures», which permit to control the number of objects described in the specification and permit also to adapt a specification to another specific transformation, and the «refinements» which permit to perform tree expansion, to derive a computing object from a specification or to increase the parallelism degree of specifications. For example, there is a so called «pipe-lining» rule which expand repetitive function computations into sequential sub-computations, creating thus a pipe-

lined processing.

In brief, high level transformation rules prepare a specification such that it can be concretized in a parallel way (if possible), while low level transformation rules adapt these specifications to an executable model, realizing (implementing) the distribution on several computing units.

4. Illustration

We will first choose a «good» specification, with well known parallel equivalents. Then, we will introduce a set of suitable transformation rules, followed by the description of the transformation on the chosen specification. At last, we will discuss about the use we can make of the resulting specification, in and out of this transformation context.

4.1. A paradigmatic problem: the sorting problem

As a paradigmatic example to illustrate this paper, we will take the well known sorting problem. This problem is suitable, because it is simple to describe and there is already parallelizable solutions, like QuickSort or MergeSort.

In the description of specifications below, we use some standard basic modules, describing data types such as booleans or integers. Consequently, we assume that we dispose of the following correct modules (only a part of their header is shown here):

```
spec      Bool;
sort      bool;
generated by
     true : -> bool;     % true value %
     false : -> bool;    % false value %
operations
     or :          bool -> bool;      % boolean disjunction %
     and :         bool -> bool;      % boolean conjunction %
end
```

Module 1 : Bool specification

```
spec      Nat;
use       Bool;
sort      nat;
generated by
     zero : nat -> nat;           % represents zero %
     succ : nat -> nat;           % increment by one %
operations
     less :         nat -> bool;  % «less» test %
     lessorequal : nat -> bool;   % «less or equal» test %
end
```

Module 2 : Nat specification

```
spec      ListOfNat;
use       Bool, Nat;
sort      listofnat;
operations
     concat :  listofnat,listofnat -> listofnat;    % concatenation %
     length :  listofnat           -> nat;          % length of a list %
     min :     listofnat           -> listofnat;    % smallest element %
     allwithoutmin :listofnat      -> listofnat;    % complement of min %
end
```

Module 3 : ListOfNat specification

Specification

```
spec      Sort;
use       Bool, Nat, ListOfNat;
operation
     sort : listofnat -> listofnat;
axioms
     lessorequal(length(x),1)=true =>
          sort(x) = x;

     lessorequal(length(x),1)=false =>
          sort(x) = concat(min(x), sort(allwithoutmin(x)));
where
     x : listofnat;
end
```

Module 4 : Sort starting specification

This CO-OPN module describes an operation, «sort», which sorts a list of integers. This operation is described in a simple, obvious way; the sorting of a list with zero or one element is the identity, and the sorting of more than one element corresponds to get the smallest element, sort the rest of the list, and then concatenate both results. In fact, this specification describes a well known sort, «permutation sort».

Now, we will develop a general method of problem splitting: the «divide to conquer» principle.

4.2. An high level transformation rule: «divide to conquer»

As an example of a high-level transformation rule, we will examine «divide to conquer», a transformation rule that realize the principle with the same name.

Definitions

Given «Spec» a specification including:

- the «bool» and «nat» algebraic abstract data types, as described above,
- X a sort,
- ϕ a constant of sort X,
- x,y,z three variables of sort X,
- F, H, T, Len, L, R six operations.

An operation F: $X,X \rightarrow X$ has the property of *fusion* iff

Spec $\vdash$(F(x,y),z) = F(x,F(y,z))	(F is associative)
Spec $\vdash$ F(x,ϕ) = x	(F has ϕ as right neutral element)

An operation H: $X \rightarrow X$ has the property of *head* iff

Spec $\vdash$ H(ϕ) = ϕ

Spec $\vdash$ H(H(x)) = H(x)

An operation T: $X \rightarrow X$ has the property of *tail* with respect to an operation H which has the property of head iff:

Spec $\vdash$ T(ϕ) = ϕ

Spec $\vdash$ T(H(x)) = ϕ

An operation Len: $X \rightarrow$ nat has the property to be a *length* on X with respect to an operation T which has the property to be a tail iff:

Spec $\vdash$ x=ϕ $\Rightarrow$ Len(x) = zero

Spec $\vdash$ x$\neq\phi$ $\Rightarrow$ Len(x) = succ(Len(T(x)))

An operation L: $X \rightarrow X$ has the property of *left part* for a predicate C with respect to operations F, H, T, Len which have the property of fusion, head, tail and length on X iff:

Spec $\vdash$ C(x)=true $\Rightarrow$ L(x) = ϕ

Spec $\vdash$ C(x)=false $\Rightarrow$ L(x) = F(H(x),L(T(x)))

Spec $\vdash$ C(x)=true $\Rightarrow$ Len(x) = zero

Spec $\vdash$ C(x)=false $\Rightarrow$ Len(x) = succ(Len(L(T(x))))

Spec $\vdash$ lessorequal(Len(x),1)=false $\Rightarrow$ less(Len(L(x)),Len(x))=true

An operation R: X $\rightarrow$ X has the property of *right part* with respect to an operation L which has the property of left iff:

Spec $\vdash$ C(x)=true $\Rightarrow$ R(x) = x

Spec $\vdash$ C(x)=false $\Rightarrow$ R(x) = R(T(x))

Spec $\vdash$ less(Len(x),1)=false $\Rightarrow$ less(Len(R(x)),Len(x))=true

We can now introduce the transformation rule itself:

Given «Spec» a specification including:

the «bool» and «nat» algebraic abstract data types, as described above,

X,Y two sorts, Op, p two operations X $\rightarrow$ Y,

F an operation with the property of fusion on Y,

F' an operation with the property of fusion on X,

H an operation with the property of head on X,

T an operation with the property of tail on X with respect to H,

Len an operation with the property to be a length on X with respect to T,

L an operation with the property of left part on X with respect to F', T and Len,

R an operation with the property of right part on X with respect to L,

an axiom A_0: lessorequal(Len(x),1)=true $\Rightarrow$ Op(x)=p(x)

an axiom A_1: lessorequal(Len(x),1)=false $\Rightarrow$ Op(x)=F(p(H(x)),Op(T(x))).

Let «Spec'» be a specification such that Spec' = Spec \ $\{A_1\} \cup \{A_2\}$, with

A_2 an axiom: lessorequal(Len(x),1)=false $\Rightarrow$ Op(x)=F(Op(L(x)),Op(R(x))).

It can be proved that DTh(Spec) = DTh(Spec'). ([6])

This formula is useful to establish new axioms of the operation Op, which can be used to derive a proper algorithm.

4.3. Application: QuickSort

Assuming the basic modules are well defined, it can be shown that in our basic specification:

«concat» has the property of fusion on the list of integers,

«length» has the property to be a length (in the sense described former),

«min» has the property of head (in the sense described former),

«allwithoutmin» has the property of tail (in the sense described former).

We construct two new operations, «left» and «right», which have the property of left part and right part respectively:

«left» returns a sublist containing the smallest half-part of the argument,

«right» returns a sublist containing the largest half-part of the argument.

```
spec      LeftRight;
use       Bool, Nat, ListOfNat;
operations
     left, right : listofnat -> listofnat;
     cptleft, cptright : listofnat, nat -> listofnat;
end
```

Module 5 : LeftRight header definitions

We can now apply the «divide to conquer» rule, and receive as result:

```
spec      Sort;
use       Bool, Nat, ListOfNat, LeftRight;
operation
     sort : listofnat -> listofnat;
axioms
     lessorequal(length(x),1)=true =>
          sort(x) = x;

     lessorequal(length(x),1)=false =>
          sort(x) = concat(sort(left((x)), sort(right(x)));
where
     x : listofnat;
end
```

Module 6 : QuickSort specification

This corresponds to a QuickSort specification. At this point, the specification is now ready to be handled by low level transformations, as a tree expansion.

4.4. A low level transformation rule: «tree expansion»

The «tree expansion» rule expand an (adequate) operation into two independent partial operations, by unfolding the previous axioms. This leads to a so-called «parallel specification» of the operation to compute. The transformation rule is quite simple:

The rule

Given «Spec» a specification including:

- X,Y two sorts,
- Op an operation $X \rightarrow Y$
- g,h two operations $X \rightarrow X$,
- f an operation $Y,Y \rightarrow Y$,
- an axiom A_0: Op(x) = f(Op(g(x)), Op(h(x)))

Let «$Spec_1$» be a specification such that $Spec_1 = Spec|_{Op \twoheadrightarrow Op1}$ ($Spec_1$ is equal to Spec where Op is renamed as Op1).

Let «$Spec_2$» be a specification such that $Spec_2 = Spec|_{Op \twoheadrightarrow Op2}$ ($Spec_2$ is equal to Spec where Op is renamed as Op2).

Let «Spec'» be a specification such that Spec' = Spec \ $\{A_0\} \cup \{A_1\} \cup Spec_1 \cup Spec_2$,with

A_1 an axiom: Op(x) = f(Op1(g(x)), Op2(h(x)))

Then we have DTh(Spec) = DTh(Spec') (restricted to the signature of «Spec»)

This formula let us rewrite Op with Op1 and Op2, performing a «tree expansion».

4.5. Application: parallel distributed QuickSort

The transformation of our example with a «tree expansion» rule leads to the following specification, what corresponds to a «parallel distributed specification» of a QuickSort, in the sense of a possible parallel and asynchronous evaluation of «sort1» and «sort2». The repetition of this transformation rule on the partial operations leads to an increase of the depth of the computation tree. The next figure shows the resulting specification:

```
spec      Sort;
use       Bool, Nat, ListOfNat, LeftRight;
operations
     sort : listofnat -> listofnat;
     sort1, sort2 : listofnat -> listofnat;
axioms
     lessorequal(length(x),1)=true =>
          sort(x) = x;

     lessorequal(length(x),1)=false =>
          sort(x) = concat(sort1(left((x)), sort2(right(x)));

     lessorequal(length(x),1)=true =>
          sort1(x) = x;
     lessorequal(length(x),1)=false =>
          sort1(x) = concat(sort1(left((x)), sort1(right(x)));
     lessorequal(length(x),1)=true =>
          sort2(x) = x;
     lessorequal(length(x),1)=false =>
          sort2(x) = concat(sort2(left((x)), sort2(right(x)));
where
     x : listofnat;
end
```

Module 7 : «parallel distributed QuickSort» specification

5. Conclusions

In this paper, we have introduced our concept of transformation of specifications, as a way to achieve parallel distributed systems, namely:

- Starting from an abstract specification expressed in the framework of algebraic abstract data types, we derived using high level transformations a concrete specification, describing a concrete algorithm.
- From this concrete specification, it is possible to apply more elementary transformations, achieving parallelism, optimizing the result and adapting it to an execution platform.

On the contrary of other similar techniques, as those proposed by the CIP ([8][9])or PROSPEC-TRA ([12]) projects, we focus on parallel aspects instead of the implementation into programming languages. Therefore, we pay attention to other projects such as BACS ([7]), which could handle our concrete parallel distributed specification, in order to obtain an executable optimized code. However, the lack of the concrete code generation leads to a simpler context of transformation, avoiding the well-known problems of the two above mentioned projects.

In this paper, we illustrate our concepts by showing how the «divide to conquer» principle can be formalized and applied to the sorting problem. This framework is now more deeply described into several internal technical reports ([4][5]).

The research direction on our project continues with the consideration of new transformation rules, covering a wide spectrum of algorithmic principles, such as dynamic programming or branch and bound techniques, in the perspective to achieve parallelism. Moreover, this project will be improved by introducing probabilistic aspects into the transformations, allowing the formal design of probabilistic algorithms, such as Monte-Carlo or Las Vegas parallel algorithms. At last, the actual transformation tool, which is a prototype, will be converted into a user friendly and complete software.

6. Bibliography

[1] Bidoit M., Kreowski H.-J., Lescanne P., Orejas F. and Sanella D., «Algebraic System Specification and Development», 1991, LNCS, Springler-Verlag.

[2] Buchs D. and Guelfi N., «A concurrent object oriented Petri net approach for system specification», 1991, article in «12th international conference on application and theory of Petri nets, Aahrus (Denmark)», M. Silva editor.

[3] Buchs D. and Guelfi N., «Distributed System Specification using CO-OPN», 1992, IEEE 3rd Workshop on Future Trends of Distributed Computing Systems, Taipei.

[4] Buffo M., «Transformations : théorie et outils», 1993, Cahier du CUI No 78, Université de Genève.

[5] Buffo M., «Collection de Transformations», 1993, Cahier du CUI N^{o} 79, Université de Genève.

[6] Buffo M., «Diviser pour régner», 1993, Internal technical report, CUI, Université de Genève.

[7] Burkhart H., Korn C. F., Gutzwiller S., Ohnacker P, and Waser S., «BACS: Basel Algorithm Classification Scheme», 1993, Technical report (93-3), RZR+IFI, Universität Basel.

[8] CIP language group, «The Munich Project CIP, volume 1: The wide Spectrum Language CIP-L», 1985, LNCS 183, Springler-Verlag.

[9] CIP language group, «The Munich Project CIP, volume 2: The program Transformation System CIP-S», 1987, LNCS 292, Springler-Verlag.

[10] Derhowitz N. and Jouannaud J-P., «Rewrite systems», 1990, article in «Handbook of theoretical computer science, volume B: formal models and semantics», editions Jan van Leeuwen.

[11] Ehrig h. and Mahr B., «Fundamentals of algebraic specification 1: equations and initial semantics», 1985, EATCS Monographs, Springler Verlag.

[12] Krieg-Brückner, «PROgram development by SPECification and TRAnsformation», article in TSI vol 9 n 2.

[13] Reisig W., «Petri nets and algebraic specifications», 1991, in «Theoretical computer science 80».