Modeling and Testing of PILOT Plans Interpretation Algorithms

Laurent NANA, Jérôme LEGRAND, Frank SINGHOFF and Lionel MARCE
University of Brest
Computer Science Department – LIMI, EA2215
20 Av. Victor Le Gorgeu, BP 832
29285 Brest Cedex FRANCE
Phone: 33-298 016 487 Fax: 33-298 016 252, E-mail: {nana,jlegrand,singhoff,marce}@univ-brest.fr

Abstract - **PILOT (Programming and Interpreted Language Of actions for Telerobotics) is a graphical and interpreted language dedicated to the remote control of systems. It is provided with a control system whose various modules have been modeled with the help of finite state automata. The work described in this paper follows a preceding work which consisted in applying static and dynamic testing techniques to PILOT plans interpreter programs. The goal is to adopt a more "formal" approach aiming at modeling the interpretation algorithms, at verifying their conformance to the operational semantics of the language, at correcting the possible malfunctions, then to regenerate the interpreter code from the validated model. We use colored Petri nets and Design/CPN for the modeling. Design/CPN has been developed in the USA and is currently maintained by the University of Aarrhus in Denmark (see http://www.daimi.aau.dk/DesignCPN for more information).**

Keywords : Modeling, testing, simulation, Petri nets, languages, telerobotics.

I. INTRODUCTION

THE language PILOT [LER, 96][LER, 97] is provided with a control system whose various modules have been modeled with the help of finite state automata [FLE, 98], but whose code has neither been generated automatically from specifications nor checked with the help of formal verification techniques. The aim of the work described in this paper is to adopt a more rigorous approach aiming at modeling the interpretation algorithms, at verifying their conformance to the operational semantics of the language, at correcting the possible malfunctions, then to regenerate the interpreter code from the validated model. It is situated in the prolongation of the works described in [NAN, 01] and is a continuation of a preceding work which consisted in applying static and dynamic testing techniques to PILOT plans interpreter programs [NAN, 02]. Actually, although those techniques proved themselves to be very useful for error detection and correction within the interpretation programs, their use doesn't make it possible to guarantee the conformance to the operational semantics of the language PILOT. We use colored Petri nets (CPN) for the modeling. The software support is Design/CPN. This tool makes it possible not only to simulate and to verify properties of our models, but also to have a modular representation and a simple "expression" of some concepts needed such as non-destructive reading of information. Design/CPN has been developed in the USA and is currently maintained by the University of Aarrhus in Denmark.

This paper starts by a brief presentation of the language PILOT and of its control system, with a particular care on the working of the interpreter. An overview of the modeling of discrete event systems is then given in the next section and the choice of Petri nets for the modeling of the interpretation algorithms is argued. This second section also situates the current work with respect to other related works. The third section discusses the modeling of the plan as well as the modeling of the interpreter algorithms and the simulation of their execution. The paper ends by the testing of the models and the interpretation of the simulation results.

II. PILOT : A LANGUAGE AND A CONTROL SYSTEM

A. *The language PILOT*

The language PILOT is based on the notion of action. Two kinds of actions are defined (see figure1) : *elementary actions* which have their own end and which generally end when their predefined goal is reached, and *continuous actions* whose end is triggered by another primitive of the language.

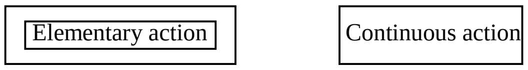


Fig. 1 Elementary and continuous actions

The language PILOT provides different control structures for plans building :

- Sequentiality : it is made of a succession, possibly empty, of actions and/or control structures. Figure 2 shows an example of sequence made of two elementary actions (Action1 and Action2). The execution of Action2 starts after the end of Action1.

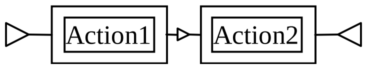


Fig. 2. Example of sequence

- Conditional : It is made of one or more alternatives ordered from top to bottom. Each alternative comprises a condition followed by a sequence. The only sequence executed is the first one whose condition is true.
- Iteration : it comprises a continuation criterion followed by a sequence. The criterion is either a number of loops (fix iteration) or a boolean expression (conditional iteration).

- Parallelism : it is made of sequences which execute in parallel. Its execution ends when all the sequences reach their end.
- Preemption : as parallelism, it is made of sequences which execute in parallel, but unlike the latter, when one of the sequences ends, it triggers the end of the other sequences and the end of the preemption.

B. The control system of PILOT

The control system is the interface between the user and the tele-operated machine. It comprises six concurrent modules : a *man-machine interface (MMI)*, an *interpreter*, a *communication server*, a *rules generator*, an *evaluator* and an *execution module* or *driver*. These modules communicate through sockets and shared memory.

The *interpreter* reads the plan stored into a shared memory space by the *MMI* and sends orders to the other modules in order to achieve the execution of the plan. Its behavior can be summarized as follows :

```
Begin Interpreter
    Create structures for plan's execution
    Initialize the communication medium
    Repeat
        MessageHandling
    End Repeat
End Interpreter
```

MessageHandling is a procedure which awaits the messages sent to the interpreter and calls the proper functions for their processing. The interpreter is therefore a *dynamic discrete event system* (DDES) which evolves at the rate of the messages it receives : launching order of plan execution, notification of the end of an action, etc. This characteristic (DDES) also applies to the interpretation of plans. For example, during the interpretation of the sequence of figure 2, it is the reception of the event "end of the first action" which triggers the execution of the next action.

After this overview of the language PILOT and of its control system, we briefly present, in the next section, the main modeling formalisms of Discrete Event Systems (DES) and we argue the choice of Petri Nets (PN) for the modeling of the interpretation algorithms of PILOT plans.

III. PETRI NETS AS MODELING FORMALISM

There are three basic types of **DES models** [ZHO, 99] : *queuing* models, *state-transition* models whose representatives are state machines and PN, and *object oriented* models. Queuing models offer mathematically concise models allowing to develop analytic solutions for first-cut quick decision making under certain restrictions. Nevertheless, it can be difficult to map some systems in terms of only queues, servers and customers as required in such approach. Another drawback of the queuing model is that the development of hierarchical models with different levels of detail is not straightforward. State-transition models are easier to relate to systems in general, allowing facile validation of a simulation model. They can be used for hierarchical modeling to facilitate system understanding and perform efficient simulation. Within state-transition models, state machines tend to be too complex for realistic industrial systems. They also have limited modeling capability in representing explicitly concurrent operations. PN offer a more powerful tool to handle discrete event dynamics [ALLA, 85] and are more compact in general. CPN offer a more compact model than PN. Object-oriented models arose from the application of object-oriented technology to modeling and simulation of DES. Their basic elements are objects whose models, as well as their interactions, can be the ones described above.

Among the approaches above, we have chosen PN and particularly CPN for several reasons :

- Their graphical nature offers the user-friendliness desired in order to use the model later as the communication medium among the various people involved into the development of the control system software, such as to avoid errors due to information distortion as mentioned in [NAN, 02].
- They make it possible to represent, with a relative simplicity, the various concepts of algorithmic and programming (sequence, iteration, conditional, parallelism, synchronization mechanisms, etc.). Unlike ordinary PN, CPN make it possible to model general assignment, structured data or boolean condition that can appear in synchronization.
- They make it possible to build compact models. CPN offer a more compact model than ordinary PN. This characteristic is very useful for a concise description of the complex structures and mechanisms of programming. With the other approaches, the model quickly becomes very complex and difficult to use.
- The availability of tools for simulation and verification.
- Their potential for mathematical analysis which make it possible to verify some properties of the system (reachability of a state, deadlock, etc.).
- The PN model is multipurpose. It may be used for the evaluation of behavioral properties, for systematic code generation or performance evaluation. This characteristic is useful in a modeling formalism [FER, 89].

Various works related to the modeling of software and programs with the help of PN have already been done : automatic generation and verification of control programs [KRO, 88], design of control software for robotics [CAL, 98], analysis of Ada programs by translation into PN [MAN, 85],[HEL, 85], [SHA, 89], [MUR, 89], [TU, 90], [KAI, 97], [BRU, 99]. The works related to the analysis of Ada programs are closer to ours. Nevertheless, they only consider parts of the language related to concurrent programming such as tasks, protected objects, entry calls, etc. In our approach, no restriction is a priori done on the set of programming structures and mechanisms to model.

After presenting the language PILOT, illustrating the working of the interpreter and arguing the choice of CPN for the modeling, we present in the next section, the modeling and testing of the interpretation algorithms of PILOT plans with the help of CPN and the Design/CPN tool.

IV. MODELING AND TESTING OF THE INTERPRETATION ALGORITHMS

Before presenting the modeling and testing of the interpretation algorithms, it is important to look into the modeling of the plans which will be used for the simulation and testing of those algorithms.

A. Plan modeling

A plan is made of a list of nodes. A node represents an action (elementary or continuous), a beginning of sequence, an end of sequence, a parallel structure, a preemption, a conditional, an iteration or a structure which is internal to one of the previous ones (condition expression of an iteration or of a conditional, number of loops, ...).

The following elements are useful for the characterization of a node of the plan : an identifier, a type, the identifier of the node encapsulating the current node, the identifier of the node following the current node, and a pointer to the internal structure. The pointer to the internal structure is only meaningful for nodes representing encapsulating primitives such as parallelism, preemption, conditional and iteration. It is also called the argument of the node.

As a consequence, each node of the plan is represented by a quintuplet (i_n,t,a,i_e,i_f) where i_n is its identifier, t its type, a its argument, i_e the identifier of the node encapsulating it, and i_f the identifier of the following node (the identifier –1 is used when there is no following node). Unary minus on numbers is represented under Design/CPN with the tilde symbol (~).

In the following subsections, we describe the fields of a node, starting from the field *type*.

1. The type of a node

The types of nodes used in plan modeling are the following : **Se** (beginning of sequence), **Sf** (end of sequence), **Ac** (continuous action), **Ae** (elementary action), **Pa** (parallelism), **Pe** (preemption), **Cd** (conditional), **It** (iteration), **Lb** (list of branches of a conditional), **Bc** (branch of a conditional; such a branch is made of a condition followed by a sequence).

2. Node identifiers

They make it possible to identify nodes uniquely within the plan. The identifier of an existing node is a natural value. The following rules apply to encapsulating node identifiers :

- The node representing the beginning of the main sequence (the one which starts the plan) has no encapsulating node. Such a node (i.e. a node which doesn't have an encapsulating node), has its field i_e set to –1.
- For a sequence SE appearing at the first encapsulation level of a preemption PR (respectively a parallelism PA, an iteration IT, a conditional CO), the node representing the beginning of SE has its field i_e set to the identifier of PR (respectively PA, IT, CO).
- A node representing a list of branches or a branch of a conditional has its field i_e set to the identifier of the node representing that conditional.
- For the other nodes (i.e. nodes of type Sf, Ac, Ae, Pa, Pe, It or Cd) , the field i_e is set to the identifier of the node representing the beginning of the sequence which contains them at the first encapsulation level.

3. The argument of a node

The argument is an integer value whose meaning depends on the type of the node. For a node of type :

- **Pa** or **Pe** : The argument is the identifier of the node representing the beginning of the first internal sequence in the textual order.
- **It** : The argument represents the number of loops or the iteration condition (positive or null in the case of a fix iteration, -1 for a conditional iteration).
- **Se** : If the node represents the beginning of sequence of a conditional branch CB, the argument is the identifier of the node representing CB. If the node represents a sequence of a parallelism PA (respectively a preemption PR), the argument is the identifier of the node representing the beginning of the next sequence in PA (respectively PR) in the textual order.
- **Cd** : The argument is the identifier of the node representing the head of the list of branches of the conditional (this head of list points to the first conditional branch of the conditional).
- **Lb** : The argument is the identifier of the node which represents the conditional branch pointed at by the list of branches.
- **Bc** : The argument represents the expression of the condition of the conditional branch. Its value is 0 when the expression is "false", 1 when it is "true", and 2 otherwise.
- other : The argument is meaningless and has a dummy value.

4. Examples of plan models

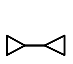


Fig. 3.

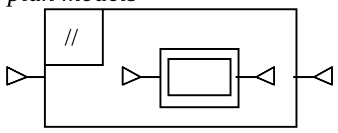


Fig. 4.

The plans of figures 3 and 4 can be modeled respectively by the following lists of nodes : [(0,Se,?,-1,1), (1,Sf,?,0,-1)] and [(0,Se,?,-1,1), (1,Pa,2,0,5), (2,Se,-1,1,3), (3,Ae,?,2,4), (4,Sf,?,2,-1), (5,Sf,?,0,-1)].
The symbol " ?" means "dummy". In practice, we use an integer value (0 in general) as the dummy argument value. The use of an integer value is necessary in order to match the argument type. By convention, the identifier of the node representing the beginning of the main sequence is 0.

B. *Modeling of interpretation algorithms*

After the presentation of the plan modeling, this section deals with the modeling of the interpretation algorithms. The general modeling approach is first presented, then the aspects of the modeling which are common to the different algorithms are described : modeling of variables, modeling of the execution states of a node, set of colors and of variables used, fusion places.

1. *General modeling approach*

We have chosen a modular approach in order to have a good readability and to facilitate the use of the model. The Petri net model comprises a page of declarations which contains the definition of the colors and variables used throughout the modeling, a page containing the initialization Petri sub-net and a page for each interpretation algorithm Petri sub-net (sequence, action, iteration, conditional, parallelism and preemption). The Petri sub-nets communicate through common places called **fusion places**. A fusion place is a place which has several instances within the same page or within different pages. The initial marking and the color set must be the same for all the instances of a fusion place. It is not necessary to have the same name for all the instances of a fusion place. Five fusion places are used in the model of the interpretation algorithms.

2. *Modeling the variables*

The approach consists in creating a colored token for each variable instance. The life cycle and the scope of the token (creation, reading, writing/modification, destruction) reflect those of the corresponding variable. For example, a token “ pre” created during the simulation run of a preemption interpretation disappears at the end of the execution of that model.

In the modeling of the algorithms, input variables (nodes of the plan, …) are accessed by bi-directional arcs (non destructive read) contrarily to input/output variables and to local variables for which two distinct arcs are used, one outgoing from the place which contains them (read operation) and one incoming to the same place (write operation).

3. *The execution states of a node*

An execution node is created at the beginning of the simulation of the execution of each node of the plan. This **execution node** is represented by a token which comprises the node itself, the state of execution of the node, the execution mode (only in the case of a node representing a beginning of sequence), the identifier of the inner most encapsulating preemption node and a pointer to the list of continuous actions attached to the sequence which contains the node. The possible execution states are the following :

- READY (REA): the node is ready for execution. When a node becomes ready, it is put into the place *Processing* (except the nodes representing end of sequences which need no processing, and the nodes of type “beginning of sequence” which are put into the place *Sequence* because of their particular processing due to the execution mode) in order to be processed by the Petri net corresponding to its type (for example, for a node of type **Pe**, the processing is done by the PN modeling the preemption interpretation algorithm).
- EXECUTING (EXE): the node is being processed.
- ENDED (END): the processing of the node is ended.

4. *The set of colors and the variables used*

The colors are defined into a version of the language ML which includes some adjustments specific to Design/CPN. The set of colors which are used are the following :

Color I = int ; (* I redefines the set of integers *)
Color B = bool **with** (false, true) ; (* B redefines the boolean type bool *)
Color T = **with** Se | Sf | Ac | Ae | Pa | Pe | Ma | Cd | It | Lb | Bc ; (* enumerated type *)
Color M = **with** KILL | NULL ;
Color E = **with** REA | EXE | END ;
Color N = **product** I*T*I*I*I ; (* set product *)
Color L = **list** N ; (* list of elements of type N *)
Color IB = **product** I*B ;
Color IL = **product** I*L ;
Color NEMPC = **product** N*E*M*I*I ;
Color NEPC = **product** N*E*I*I ;

The variable declarations are the following :
Var n1, n2 **:** N ; **Var** i1, a1, a2, e1, e2, s1, s2, pre, lc **:** I ; **Var** t1, t2 **:** T ; **Var** l **:** L ;

5. *The fusion places*

- <u>The fusion place *Plan* :</u> This place is only useful for storing the nodes of the plan to interpret. Its marking is constant. Actually, The interpretation doesn’t modify the plan.
- <u>The fusion place *Pre* :</u> This place stores all the tokens modeling the preemption variables which are generated during the interpretation simulation of the plan. Each preemption variable *pre* is modeled by a couple (*i, b*) where *i* is the identifier of the node whose interpretation led to the creation of the preemption variable, and *b* a boolean variable indicating if the preemption condition is true or not (*b* is initialized to false at the creation of the variable *pre*). A variable *pre* is created at the beginning of the interpretation of a preemption node. A particular variable *pre* is also created when starting the interpretation of the plan. The identifier of the latter is that of the main sequence, i. e. 0.
- <u>The fusion place *ActionsC* :</u> In this place, one finds the variables *list_cont* which make it possible to access the continuous actions which are being processed within a parallelism, a preemption or a sequence executed in mode KILL. A variable *list_cont* is represented by a couple (*i, l*) where *i* is an identifier and *l* a list of continuous actions. The variable *list_cont* associated with a sequence executed in mode KILL (respectively a parallelism, a preemption) has the same identifier as the sequence (respectively the parallelism, the preemption). The list associated with it is made of the continuous actions being executed within the sequence (respectively the parallelism, the preemption) at the first encapsulation level.

- The fusion place *Sequence* : In this place, one finds only the execution nodes corresponding to the beginning of sequences. What makes the difference between these execution nodes and the others is the presence of the component "execution mode" which makes it possible to distinguish sequences executed in mode KILL from those executed in mode NULL. Unlike sequences executed in mode NULL, a sequence executed in mode KILL stops the execution of its continuous actions when it reaches its end.
- The fusion place *Processing* : The place *Processing* models the execution of the function "processing" appearing into the algorithms. In that place, one processes the nodes of the sequences being processed. When the execution of a node starts, an execution node is created for it in the place *Processing*. The execution nodes of this place have almost the same components as those of the place *Sequence*. The only difference is that they do not have the "execution mode" component.

After this presentation of the elements which are common to the modeling of the different algorithms, we present below the modeling of the interpretation algorithm of the sequence. For concision reasons, the PN of the other interpretation algorithms aren't presented (elementary and continuous actions, conditional, iteration, parallelism and preemption).

C. Modeling of the sequence interpretation algorithm

1. The interpretation algorithm of the KILL (respectively NULL) sequence

```
Local Parameters (respectively Input / Output)
    list_cont : list of continuous actions
Input
    pre : boolean
Local Parameters
    current_prim : pointer to the current primitive
Begin
    While current_prim ≠ null and not (pre)
        If current_prim→type = 4        /* continuous action */
            Then add(current_prim, list_cont)
        End_If
        processing(current_prim, pre)
        current_prim ← current_prim→next
    End_While
    stop(list_cont) /* stopping continuous actions */ (respectively nothing)
End
```

Figure 5 shows the PN modeling the sequence interpretation algorithm. It is described below.

2. Description of the PN modeling the sequence interpretation algorithm

The places

As one can note, the model proposed comprises, in addition to the 5 fusion places presented above, 2 specific places *current_prim* and *stop_seq*. The place *stop_seq* is used for storing the identifiers of the beginning of the sequences to stop. The place *current_prim* models as for it the variable *current_prim*. Its color is the same as that of the place *processing*.

The variables

Each variable "pre" (respectively "list_cont", "current_prim") is represented by a token stored into the place *Pre* (respectively *ActionsC*, *current_prim*).

Initializations

A sequence is ready for execution when it has an associated execution node into place *sequence* with a state field set to REA. Before the actual launching of the algorithm simulation, the variables are initialized according to the execution mode of the sequence :

- mode NULL (transition InitNull): the next node is put into the place *current_prim*.
- mode KILL (transition InitKill): the variable "list_cont" is created and the next node is put into the place *current_prim*.

In both cases, the execution state of the sequence changes from REA to EXE (executing).

Handling of the condition of the loop "While"

The handling of the loop **While** starts as soon as the current primitive is ready for execution in the place *current_prim*. At this point, there are 3 possibilities :

1. The sequence is preempted i.e. "pre" has the value true (transition PreemptiveStop): the identifier of the sequence is put into the place *stop_seq*.
2. The current primitive is an "end of sequence" (transition EndSeq) : the identifier of the sequence is put into the place *stop_seq*. This case models the test of the condition *current_prim ≠ nul*. The transition EndSeq is fired when that condition is false.
3. The sequence is not preempted and the current primitive is not an "end of sequence" (transition CondProcessingOk) : the node is put into the place *processing*. One can note the transmission of the parameter "pre"; that of "lc" is necessary, although the latter is not present in the call "*processing (current_prim, pre)*". In fact, since the place *processing* is a fusion place used for different calls to the function *Processing* among which some require the parameter list_cont, it is convenient to have the same interface, in order to satisfy the type constraint on the color type associated with the different instances of a fusion place (the color type must be the same).

Handling of the Processing

The handling of the **If** statement appearing in the sequence interpretation algorithm is done within the place *Processing*. Depending on the type of the node, the corresponding PN processes it. At the end of the processing of a node, the state of the corresponding execution node passes from EXE to END into the place *processing*.

Passing to the next primitive

The change from the current primitive to the next one happens when the execution node of the current primitive is in state END within the place *current_prim* (transition "Next") : an execution node is created for the next primitive and is put into the place *current_prim* with the state ready (REA).

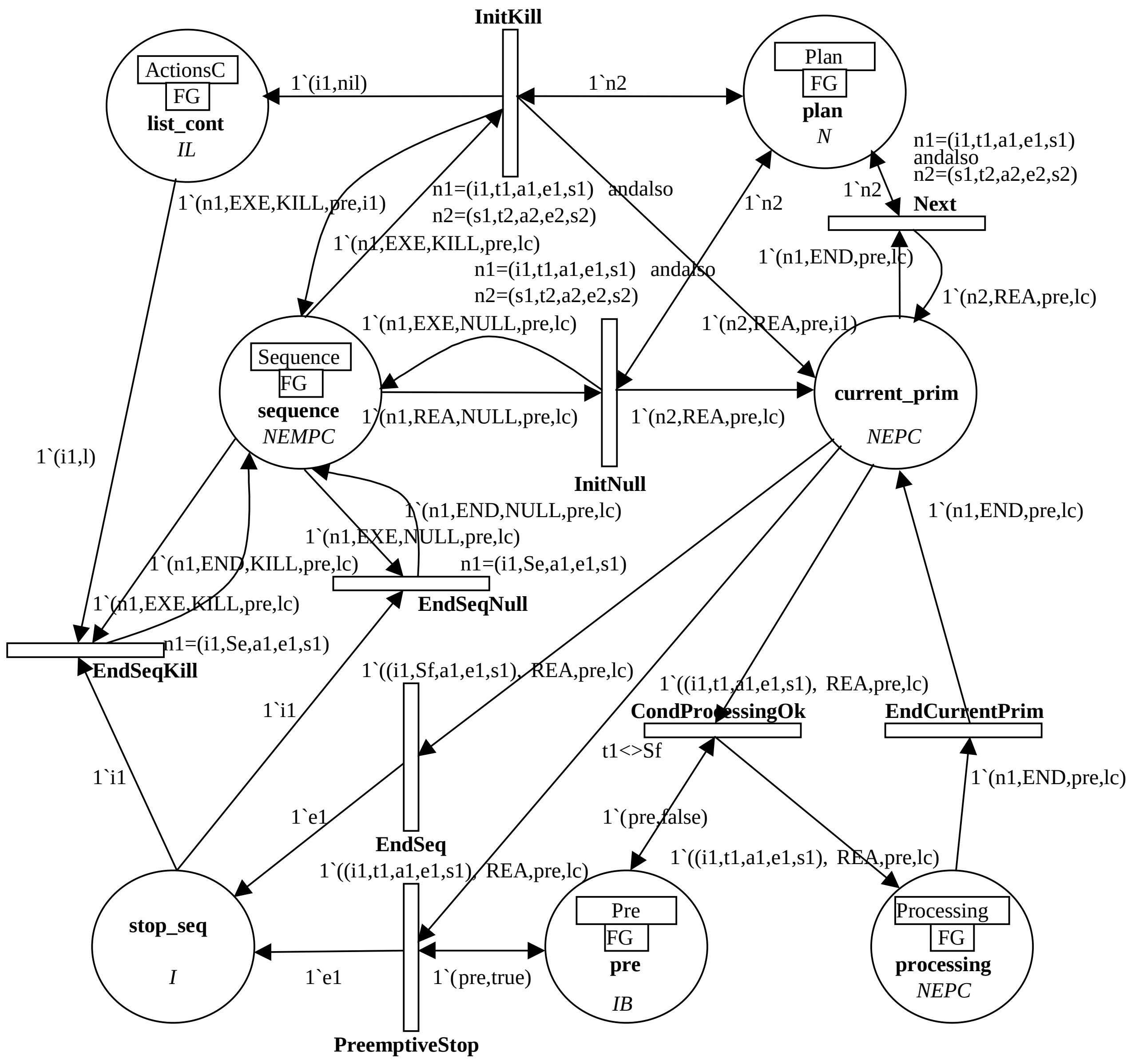


Fig. 5 Petri net modeling the sequence interpretation algorithm

Handling of the termination of a sequence

When an identifier is within the place stop_seq, the corresponding execution node into the place “sequence” passes to the state END (transitions EndSeqNull and EndSeqKill). In other respects, in the case of a sequence executed in mode KILL (transition EndSeqKill), the corresponding continuous actions are removed from the place ActionsC (*list_cont*) to simulate their end.

D. The tests

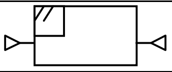
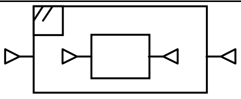
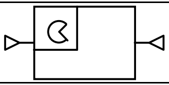
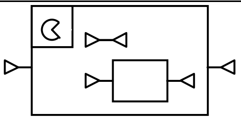
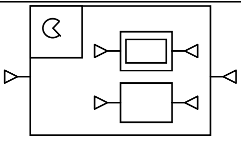
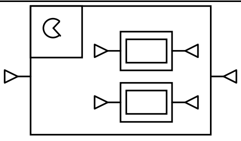
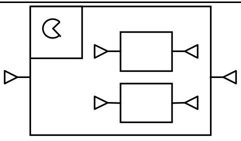
In order to test the interpretation algorithms, testing data have been generated on the basis of the approach described in [NAN, 02]. The following initializations are done before the simulation :

- The place *Plan* is initialized with a plan model (testing data).
- The place *Pre* is initialized with a preemption token 1`(0,false). This token is used as input parameter for the call to the interpretation algorithm on the main sequence.
- The place *Sequence* is initialized with a token representing the execution node associated with the main sequence. The execution mode and the state are respectively set to KILL and REA (ready).

A sample of test data whose execution has been simulated with the help of the interpretation algorithms model is shown in table 1. Only plans which are useful for a good understanding of the analysis of simulation results appear in the table.

The third column of the table shows the initial marking of the fusion place *Plan* for each testing plan. The matter is the representation, in the shape of multi-set (shape of markings under Design/CPN), of the plan model built according to the approach described in section 2.

Table 1. Sample of testing data

No	Plan	Design/CPN model
8		1`(0,Se,0,~1,1)+1`(1,Pa, ~1,0,2)+1`(2,Sf,0,0,~1)
11		1`(0,Se,0,~1,1)+1`(1,Pa, 2,0,5)+1`(2,Se,~1,1,3)+ 1`(3,Ac,0,2,4)+1`(4,Sf, 0,2,~1)+1`(5,Sf,0,0,~1)
20		1`(0,Se,0,~1,1)+1`(1,Pe, ~1,0,2)+ 1`(2,Sf,0,0,~1)
28		1`(0,Se,0,~1,1)+1`(1,Pe,2, 0,7)+1`(2,Se,4,1,3)+1`(3, Sf,0,2,~1)+1`(4,Se,~1,1,5) +1`(5,Ac,0,4,6)+1`(6,Sf,0, 4,~1) + 1`(7,Sf,0,0,~1)
29		1`(0,Se,0,~1,1)+1`(1,Pe,2, 0,8)+1`(2,Se,5,1,3)+1`(3, Ae,0,2,4)+1`(4,Sf,0,2,~1)+ 1`(5,Se,~1,1,6)+1`(6,Ac,0, 5,7)+1`(7,Sf,0,5,~1)+1`(8, Sf,0,0,~1)
30		1`(0,Se,0,~1,1)+1`(1,Pe,2, 0,8)+1`(2,Se,5,1,3)+1`(3, Ae,0,2,4)+1`(4,Sf,0,2,~1)+ 1`(5,Se,~1,1,6)+1`(6,Ae,0, 5,7)+1`(7,Sf,0,5,~1)+1`(8, Sf,0,0,~1)
31		1`(0,Se,0,~1,1)+1`(1,Pe,2, 0,8)+1`(2,Se,5,1,3)+1`(3, Ac,0,2,4)+1`(4,Sf,0,2,~1)+ 1`(5,Se,~1,1,6)+1`(6,Ac,0, 5,7)+1`(7,Sf,0,5,~1)+1`(8, Sf,0,0,~1)

V. RESULTS OF SIMULATION AND ANALYSIS

The simulation of the execution of the testing plan number 11 raises the question of meaning of such a plan. In fact, although its execution ends, the continuous action is stopped by the end of the sequence which contains the continuous action, i.e. just after the launching of the execution of that action. The simulation of the execution of the testing plan number 20 leads as for it to a deadlock into the waiting of the end of a first sequence which doesn't exist; one would have however expected an immediate end as in the case of a parallelism without internal sequence (case of testing number 8). The testing number 28 also revealed a problem in the preemption algorithm : the sequence containing the continuous action can trigger the stopping of the other sequence of the preemption. The same problem remains in the other tests related to the preemption (test 29, 30 et 31). The other tests (those which don't appear in table 1) have revealed no malfunction during simulation.

VI. CONCLUSIONS AND PERSPECTIVES

The modeling has made it possible to realize that some simplifications are possible on the internal representation of the plan and on the interpretation algorithms. As far as the internal structure of the plan is concerned, one of the possible simplifications is the suppression of the node "liste_seq". As far as the interpretation algorithms are concerned, the suppression of the sequence execution modes is possible and makes it possible to merge the KILL and NULL sequence interpretation algorithms. This simplification has been done in the current version of the interpreter. Plans such as the number 11 of table 1 are no longer accepted by the syntax of the language. Similarly, preemption contains, as every parallelism, at least one normal sequence (sequence without continuous action at the first encapsulation level), and this steps aside the problem observed in the execution simulation of test 20. In [NAN, 02], we have mentioned that some software errors are inherent in information distortion or loss throughout the development process and that errors may also be due to an imperfect knowledge of the programming language. The interpretation algorithm model proposed in this work may be used, after validation, as support for the learning of the semantics of the language and of the working of the interpreter, for the people involved into the development and the maintenance of the source code of the interpreter. We are currently investigating the conformance of the interpretation algorithms to the operational semantics of the language, the next step being the code regeneration from the validated interpretation models.

REFERENCES

[ALL, 85] H. Alla, P. Ladet, J. Martinez, and M. Silva, 1985. *Modeling and validation of complex systems by colored Petri nets : application to a flexible manufacturing system,* Advances in Petri Nets 1984, G. Rozenberg, H. Genrich, and G. Roucairol (Eds.), Springer-Verlag, pp. 15-31.

[BRU, 99] E. Bruneton, and J.-F. Pradat-Peyre, 1999. *Automatic Verification of Concurrent Ada Programs,* Ada-Europe'99 International Conference on Reliable Software Technologies, Santander, Spain, pp. 146-157.

[CAL, 98] A. Caloini, G. Magnani and M. Pezze, 1998. *A technique for designing robotic control systems based on Petri nets*, IEEE Trans. on Control Systems Technology, 6(1):72-87

[FER, 89] S. Ferray-Beaumont and S. Gentil, 1989. *Declarative Modelling for process supervision,* Revue d'intelligence artificielle, vol 3, no 4.

[FLE, 98] J.L., 1998. *Vers une méthodologie d'un système de programmation de télérobotique: comparaison des approches PILOT et GRAFCET,* PhD thesis, Université de Rennes 1, France.

[HEL, 85] D. Helmbold and D. Luckham, 1985. *Debugging Ada-tasking programs,* IEEE Transactions on Software Engineering, 2 (2):45-57.

[KAI, 97] C. Kaiser and J.F. Pradat-Peyre, 1997. *Comparing the reliablity provided by tasks or protected objects for implementing a resource allocation service : a case study.* In Tri'Ada, St Louis, Missouri. ACM SIGAda.

[KRO, 88] B. H. Krogh, R. Willson and D. Pathak, 1988. *Automated generation and evaluation of control programs for discrete manufacturing processes.* Proc. of 1988 Int. Conf. On Computer Integrated Manufacturing, Troy, NY, pp. 92-99.

[LER, 96] E. Le Rest, 1996. *PILOT: un langage pour la télérobotique,* PhD thesis, Université de Rennes 1, France.

[LER, 97] Le Rest, E., J.L. Fleureau and L. Marcé, 1997. *PILOT: semantics and implementation of a language for telerobotics*, In IROS'97, IEEE, Grenoble, France.

[MAN, 85] D. Mandrioli, R. Zicari, C. Ghezzi, and F. Tisato, 1985. *Modeling the Ada task system by Petri nets*, Computer Languages, Vol. 10 (No. 1) : 43-61.

[MUR, 89] T. Murata, B. Shenker, and S. M. Shatz, 1989. *Detection of Ada static deadlocks using Petri nets invariants*, IEEE Transactions on Software Engineering, 3 (15):314-326.

[NAN, 01] L. Nana Tchamnda and L. Marcé, 2001. *Vers une programmation sûre avec PILOT*, MSR'2001, Colloque Francophone sur la Modélisation des Systèmes Réactifs, Toulouse, France.

[NAN, 02] L. Nana Tchamnda, V-A. Nicolas and L. Marcé, 2002. *Towards a formal approach for the regeneration of PILOT control system*, 6th World Multiconference on Systemics, Cybernetics and Informatics, SCI'2002, IEEE Venezuela (Technical Co-Sponsor), Orlando, Floride.

[SHA, 89] S. M. Shatz, K. Mai, D. Moorthi, and J. Woodward, 1989. *A toolkit for automated support of Ada-tasking analysis*, In Proceedings of the 9th Int. Conf. On Distributed Computing Systems, pp. 595-602.

[TU, 90] S. Tu, S. M. Shatz, and T. Murata, 1990. *Applying Petri nets reduction to support Ada-Tasking deadlock detection*, In Proceedings of the 10th IEEE Int. Conf. On Distributed Computing Systems, pp. 96-102, Paris, France.

[ZHO, 99] M. C. Zhou and K. Venkatesh, 1999. *Modeling, simulation and control of flexible manufacturing systems, A Petri Net Approach*. Series in intelligent control and intelligent automation, Vol. 6., World Scientific Publishing.