Implementation of a Toolbox for Colored Petri Nets in G2

Sarraut, Patrick

CODEN: LUTFD2/(TFRT-7480)/1–66/(1991)

Implementation of a Toolbox for Colored Petri Nets in G2

Patrick Sarraut

Department of Automatic Control
Lund Institute of Technology
September 1991

Department of Automatic Control **Lund Institute of Technology** P.O. Box 118 S-221 00 Lund Sweden	*Document name* Report *Date of issue* September 1991 *Document Number* CODEN: LUTFD2/(TFRT-7480)/1-66/(1991)
Author(s) Patrick Sarraut	*Supervisor* *Sponsoring organisation*

Title and subtitle
Implementation of a Toolbox for Colored Petri Nets in G2.

Abstract

This report describes the implementation of a toolbox for colored Petri nets in G2. Special attention is paid to the evolution procedures the discolored version of the colored Petri nets that is built automatically and on which the evolution is based. Some modeling examples are shown. Petri nets and G2 are briefly presented.

Key words

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title	*ISBN*

Language English	*Number of pages* 66	*Recipient's notes*
Security classification		

The report may be ordered from the Department of Automatic Control or borrowed through the University Library 2, Box 1010, S-221 03 Lund, Sweden, Telex: 33248 lubbis lund.

ABSTRACT .

This report describes the implementation of a toolbox for colored Petri nets in G2. Special attention is paid to the evolution procedures the discolored version of the colored Petri nets that is built automatically and on which the evolution is based. Some modeling examples are shown. Petri nets and G2 are briefly presented.

RESUME .

Ce rapport présente la boîte à outils pour réseaux de Petri colorés qui a été implementée dans G2. La construction du réseau se fait à l'aide des objets implementés et de leurs attributs. L' évolution du réseaux est basées sur une version décolorée du réseau qui est construite automatiquement. Les procédures et les règles permettant l'évolution sont décrites dans ce rapport. Finalement des examples d'applications sont présentés.

Table of contents.

1. Introduction.

Knowledge based systems for monitoring and control of complex industrial processes are becoming increasingly common. The systems are built upon heuristics and/or models. The models are usually of a more qualitative nature than what is usual in systems based on conventional techniques. Most models, e.g., signed digraph model, constraint equation models, and qualitative physics models, only capture the continuous part of the process. However, processes also always contains discrete, sequential parts, e.g., start-up sequences, batch operation, etc.

Petri nets is a powerful modeling technique for representing sequential, discrete processes. Several different classes of Petri net exists. One of them is the colored Petri nets. Petri nets are, however, normally not associated with knowledge based systems.

The aim of this thesis is to implement a toolbox for colored Petri nets in the knowledge based environment G2 from Gensym Corp (Moore et al ,1990). G2 is one of the most commonly used systems for the industrial real time application of knowledge based systems. G2 is quite unique in that it allows knowledge based programming techniques to be combine with traditional procedural techniques in a uniform object oriented environment. The toolbox is based upon a G2 toolbox for Grafcet, called Grafchart (Årzén ,1991), that previously has been implemented at the Lund Institute of Technology. The colored Petri net toolbox maintains the intersting properties of Grafcet: transitions can be conditioned, temporal expression can be used, and places can be associated with actions. The work has been done as a master thesis project in a collaboration between Lund and Grenoble. The work has been carried out at the Department of Automatic Control in Lund.

The report is divided in two parts. In the first chapter, the implementation of colored Petri nets in G2 is explained. The objects and their attributes are described, and rules and procedures used for the evolution of colored Petri nets are presented. In the last chapter, examples are presented.

In the appendix A, a brief overview of Petri nets is persented with special emphasis on colored Petri nets. A general description of G2 is given in appendix B. Finally in appendix C a listing of the G2-procedure used in the toolbox is given.

2. The G2 implementation.

In this chapter, the aim of the toolbox will be expressed, the way to implement colored Petri nets (CPN) will be discussed, the objects of the tool box will be described, and the evolution of CPN is presented.

2.1 The aim of the toolbox.

The toolbox has been designed with two aims in mind:

- Firstly, building a colored Petri net must be easy. In the tool box, the user must find all the facilities that he has when he builds a colored Petri net on paper. He must find all the predefined functions that are usually used in CPNs. The way to enter data must be very easy. The data of a CPN consists of functions, firing colors, initial marking, events or conditions linked to transitions, and delays or actions linked to places.
- Secondly, the evolution of a CPN must respect the real time requirements and must respect the rules of PN evolution, specially the way to fire transitions on the occurrence of an event.

2.2 Overview.

2.2.1 To build a CPN in G2.

First, the user creates an object called **petri-net**. Then he buildss the net on the subworkspace of this object. The user creates places and transitions and links places and transitions with arrows. After the drawing, the user has to enter data:

- When the user clicks on the table of a place, he finds the attributes to enter the delay of the place and the initial marking of the place. There is a predefined syntax to enter the initial marking.
- When the user clicks on the table of a transition, he finds an attribute called *color-set-def* where he must enter the set of firing colors of the transition. The user must respect a predefined syntax.
- When the user clicks on the table of an arrow, he finds an attribute called *input-function-def* or *output-function-def* . The name depends on if the arrow is at the input or at the output of a transition. A syntax has been defined for defining new functions or for using predefined functions.
- If an action is linked to the presence of some marks in a place, then the user has to define an action-procedure and create a rule in the subworkspace of the place that starts the action-procedure. The form of the rule indicates if the action is a pulse action or a continuous action. The action can depend on the color of the mark.

- If conditions or events condition the firing of a transition with a firing color, then a predefined rule must be put in the subworkspace of the transition.

When all these things have been created, the Petri net is ready to be 'activated'. When the user wants to 'start' a petri-net he clicks on the Petri-net object and clicks on **initialization**. When the Petri-net object becomes green, he clicks on this object and selects **run**. Then the evolution starts.

2.2.2 Initialization and evolution.

Initialization.

The initialization consists of creating the discolored version of the Colored Petri Net . This version is obtained automatically from the elements of the CPN. Each place of the CPN generates subplaces (one per mark color that this place can contain). Each transition generates a subtransitions (one per firing color of the transition). Finally, each arrow of the CPN generates sublinks that link subplaces and subtransitions. The creation of sublinks depends on the function of the arrow of the CPN.

After obtaining the discolored version, the initial marking of the CPN is set. Then the evolution can start.

Evolution.

The evolution is based on the discolored version of the CPN. G2 calculates the evolution in the discolored version of the CPN and the CPN shows the evolution of its discolored version:

- The firing of a subtransition in the discolored PN leads to the firing of a transition of the CPN with a firing color.
- Creating or deleting a mark in a subplace leads to the creation or the deletion of a colored mark in a place of the CPN.

The evolution of the PN are carried out by G2 procedures that perform the different tasks of the evolution. The procedures are started from rules or from other procedures. The procedures respect the rules of evolution of a non-autonomous PN.

2.3 Definition of classes of objects.

In this section the different objects will be described.

2.3.1 A petri-net.

An application can contain more than one CPN. For each CPN, the user must create a petri-net object. The CPN is built on the subworkspace of the petri-net object. The CPN is initialized or stopped through the appropriate menu choices of the petri-net object. A petri-net is shown in Fig 2.1.

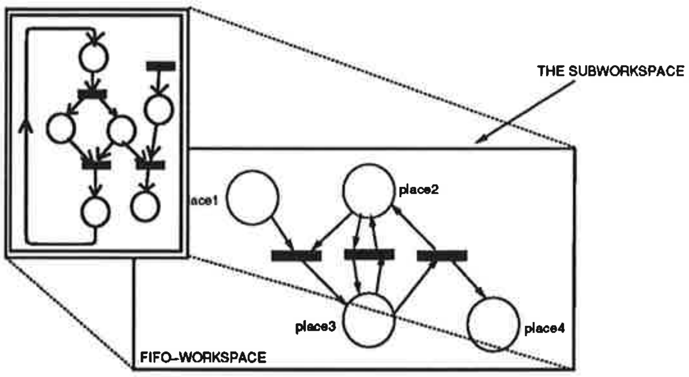


Figure 2.1 A petri-net object with is subworkspace.

2.3.2 A place.

A place is represented by a place object shown in Fig 2.2. A place has two attributes: *delay* and *initial-marking*. *Delay* is a quantity with initial value is 0 that gives the delay of the temporized place. *Initial-marking* is a text. In this text, the user can enter the initial marking using a predefined syntax.

The syntax of initial marking.

- If the initial marking of a place is 1 mark with the color *blue* and 3 marks with the color *red* then the user has to enter: "3<red>1<blue>".
- If the initial marking of a place is : $\sum_{i=1}^{n} convoyer_i$ then the user has to enter: "SUM(*convoyer*,n)". In that case a succession of *convoyer.i* is created. It is the shortest way to enter: "1<convoyer.1>1<convoyer.2> ...1<convoyer.n>" □

Place actions.

If an action is linked to the presence of a mark in an place, then the action is expressed as G2 items on the subworkspace of the place. The most common way is to represent actions by a G2 rule that is combined with a G2 procedure as shown in Fig 2.2.

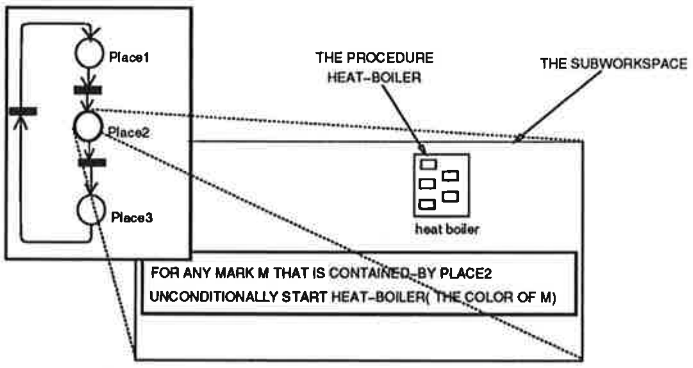

Place1
Place2
Place3
THE PROCEDURE
HEAT-BOILER
THE SUBWORKSPACE
heat boiler
FOR ANY MARK M THAT IS CONTAINED-BY PLACE2
UNCONDITIONALLY START HEAT-BOILER(THE COLOR OF M)

Figure 2.2 A place with its subworkspace.

Continuous actions can be expressed by scanned rules and pulse actions by whenever rules. The condition part of the rule always begins with the same phrase. Some examples are given below:

○ ***continuous actions***

for any mark *M* that is contained-by *Place*1 unconditionally

```
        start left-motion-car(the color of $M$)
(scanned rule)


    for any mark $M$ that is contained-by $Place2$
        if level-of-tank(the color of $M$) $\geq 10$
        then start tank-full-procedure(the color of $m$)
(scanned rule)
```

o ***pulse actions***

```
    whenever a mark $M$ becomes contained-by $place3$ then
        start open-valve-tank(the color of $M$)


    whenever a mark $M$ becomes contained-by $place4$ and
        when the status of the process is OK then
        start start-of-production-of-piece(the color of $M$)
```

When a mark is created in a place, then the mark is related to the place with the relation *contained-by.*

Every action can depend on the mark color. For instance, in the action **open-valve-tank(the color of mark)**, the action to performed is '*open the valve of a tank*' and *the color of mark* indicates which tank to open.

2.3.3 A transition.

A transition object and its connections is shown in Fig 2.3.

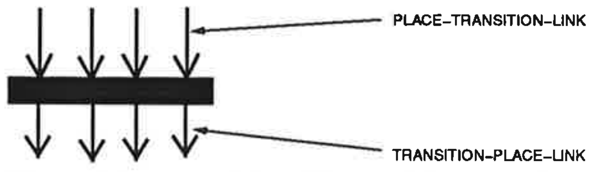


Figure 2.3 A transition object with its connections.

Transition objects have two attributes: *color-set* and *color-set-def* . *Color-set-def* is a text where the user has to enter the set of firing colors using a predefined syntax. The example below explains the syntax. *Color-set* is given by a text-list. After the initialization of the petri net this list contains all the firing colors of the transition. It is obtained by translating the text of *color-set-def* . It is necessary to define *color-set-def* for each transition.

The syntax of *color-set-def* .

- If the set of firing colors is {blue,red,black,green,yellow}, then the user has to enter: "<blue><red><black><green><yellow>".
- If the set of firing colors is : $\sum_{i=1}^{n} boiler_i$, then the user has to enter : "SUM(*boiler*,*n*)". In this case a succession of *n boiler.i* is created. It is the shortest way to enter: "<boiler.1><boiler.2> ... <boiler.n>". □

Transition condition rules.

If a transition is linked to a condition or an event, then the user expresses the condition or event with a rule that he puts in the subworkspace of the transition. The action part of the rule is always: ***start fire-event(transition, firing color)*** or ***start fire-event1(subtransition)***. The action of these two procedures are identical. Which to use depends on the argument that is used in the condition part of the rule as shown in the following example.

Whenever-rules are used for representing the occurrence of an event (an event is considered as a pulse condition). For the other conditions when-rules are used.

EXAMPLE 2.1—The syntax of transition condition rules.

For any subtransition ***st*** that is waiting-an-event to $T1$

When sensor-temperature-boiler(the color of ***st***) ≥ 100 then

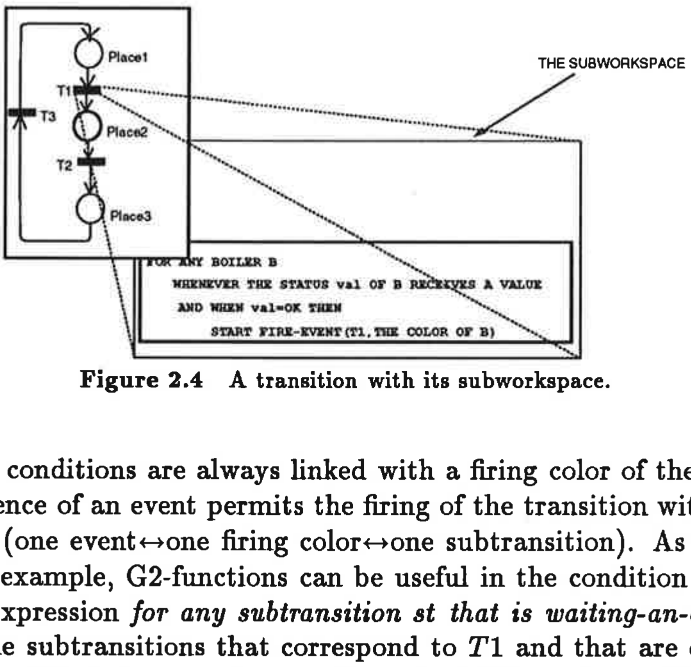
Start fire-event1(***st***).

(scanned rule)

For any boiler ***boiler***

whenever the status ***val*** of ***boiler*** receives a value and when ***val***=OK then

Start fire-event($T2$,the color of ***boiler***). □



```
FOR ANY BOILER B
   WHENEVER THE STATUS val OF B RECEIVES A VALUE
    AND WHEN val=OK THEN
        START FIRE-EVENT(T1,THE COLOR OF B)
```

Figure 2.4 A transition with its subworkspace.

Events and conditions are always linked with a firing color of the transition. The occurrence of an event permits the firing of the transition with a defined firing color (one event↔one firing color↔one subtransition). As it is shown in the first example, G2-functions can be useful in the condition part of the rule. The expression *for any subtransition st that is waiting-an-event to* $T1$ indicates the subtransitions that correspond to $T1$ and that are enabled. In other words, this is the way to have the set of firing colors with which the transition is enabled.

In the second example, there is a rule that translates the occurrence of an event. The enableness of the transition is not a part of the rule. Each time the event occurs the procedure Fire-event is started; but in this procedure, the enableness of the transition is calculated and if the transition is not enabled, then nothing happens.

2.3.4 A mark.

In colored Petri nets, the state is given by the marking of places with colored marks. A mark object is a small object that is placed on places. This object has two attributes: *color* and *time-of-creation* . *Time-of-creation* is given by a quantitative-parameter. Its value is the time instant when the mark was created. *Color* is given by a text-parameter. The color of a mark can be complex or with an index. The predefined syntax of these two kinds of marks is shown in the following example. It is necessary to respect this syntax in order to use the predefined functions.

EXAMPLE 2.2—The predefined syntax for mark color.

Index	: a mark 'car_i' is transformed to 'car.1'.
Complex	: a mark '$red/robot_2/work_4$' becomes 'red-robot.2-work.4'.

□

Mark relations.

Contained-by is a relation that links mark objects with places. A mark is *contained-by* a place as soon as it is created in the place.

Available-in is a relation that links mark objects with places. A mark is *available-in* a place after having waited during the delay of the place.

2.3.5 Functions.

The functions on the arrows of a CPN are implemented in G2 using attributes of the connections objects representing the arrows which link place objects to transition objects. These connections are instances of the place-transition-link or the transition-place-link objects according to the direction of the arrows. The two links have one attribute : input-function-text (output-function-text) that is a text. In this text, the user enters the function.

A lot of predefined functions have been defined. In the case of predefined functions, the user has to type the name of the function with, possibly, some arguments. The function is applied to each firing color of the transition that is connected to the arrow. The syntax is important but simple to understand and very close to the habitual syntax. The user has to learn the syntax of these predefined functions and the syntax for building his own functions.

Predefined functions

Name	Definition	Note
ID	ID(c)=c	Identity
DEC	DEC(c)=dec	Discoloration(dec represents a mark without color)
ADD(C2)	ADD(C2)(c)=c-C2	Addition(ADD adds the color C2 at the end of the color c)
INV(C2)	INV(C2)(c)=C2	Invariant function (for all firing color the result is C2)
SUCCi	SUCCi(c1-...-ci.j-...-cn)=ci.j+1	Successor of(gives the successor of the i^{th} color of a complex color)
PRECi	PRECi(c1-...-ci.j-...-cn)=ci.j-1	Precessor of (gives the precessor of the i^{th} color of a complex color)
COLOi	COLOi(c1-c2-...-ci-...-cn)=ci	Color (gives the i^{th} color of a complex color)

SUCC, PREC and COLO can be combined to give very complex functions. In this case the user use the function COMB. Comb stand for combine The syntax is:

COMBi(n1,n2,...,ni) where :

i is the number of arguments
ni is the name of the i^{th} function to combine.

Each function is applied to the firing color and all the results are combined to give a complex color.

EXAMPLE 2.3—Some results of predefined functions.

ID(red-car)=red-car
DEC(red-car)=dec
ADD(blue)(red-car)=red-car-blue
INV(blue)(red-car)=blue
SUCC1(car.1-pro.3)=car.2 ; SUCC2(car.1-pro.3)=pro.4
PREC1(car.2-pro.3)=car.1 ; PREC2(car.2-pro.3)=pro.2
COLO1(car.1-blue-volvo-clean)=car.1
COLO2(car.1-blue-volvo-clean)=blue
COLO3(car.1-blue-volvo-clean)=volvo
COMB3(COLO3,SUCC2,PREC1)(car.2-fiat.3-red)=red-fiat.4-car.1
the last results is the combination of:

COLO3(car.2-fiat.3-red)=red
SUCC2(car.2-fiat.3-red)=fiat.4
PREC1(car.2-fiat.3-red)=car.1

□

Other functions.

If the function is not a predefined function, the user must define the function with the syntax :
"$fc_1 : n_1^1 < c_1^1 > n_1^2 < c_1^2 > ...n_1^k < c_1^k >; fc_2 : n_2^1 < c_2^1 > ...; fc_n : ...;$"
where fc_i are firing colors of transition
c_j^i are mark colors and n_j^i the number of marks.

EXAMPLE 2.4—A function.

The user has to enter the following text to define the function **f** that is shown in Fig 2.5: "$m1 : 1 < m1 > 2 < m2 >; m2 : 3 < m1 >;$" .

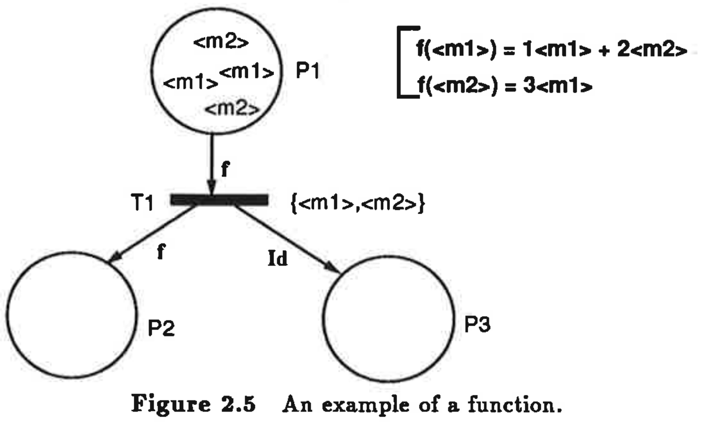


Figure 2.5 An example of a function.

Functions at the input of a transition indicate the number and the color of marks that imply the enableness of the transition for each firing colors. These marks will be removed during the firing of the transition with this firing color.

For each firing color, functions at the output indicate the number and the color of marks that will be created during the firing of a transition with the firing color.

2.3.6 Subobjects.

The subobject are placed on a special workspace named **discolored-version.** This workspace is not intended to been seen by the user. The user must not delete this workspace.

Subplace.

A subplace is created for each type of mark that a place can contain. Every subplace is linked with this place with the relation: **in-relation-with** with the inverse relation **related-with.** A subplace has two attributes: *color* and *nb-element-available. Color* is given by a text-parameter; *nb-element-available* is given by a quantitative-parameter. *Color* indicates the color of the type of mark; *nb-element-available* is the number of those marks that are available in the place.

Subtransition.

A subtransition is created for each firing color linked to a transition. Every subtransition is linked with a transition with the relation: **in-relation-with** with the inverse relation **related-with.** A subtransition has two attributes: *color* and *enable. Color* is given by a text-parameter that indicates the firing color. *Enable* is a logical-parameter that indicates if the transition is enabled with the firing color or not.

Sublink.

There are two sorts of sublinks: **output-sublink** and **input-sublink.** An output-sublink (input-sublink) connects a subplace at an output (input) of a subtransition. Both have an attribute: *weight. Weight* is given by a quantitative-parameter. *Weight* gives the number of marks to remove or to create during the firing.

2.3.7 Other objects.

Color-petri objects.

A color-petri object is used for giving a special icon-color to a mark. The object has only one attribute: *color. Color* is a symbolic-parameter that contains the desired icon-color. The name of the color-petri object is the color of the mark that the user wants to animate.

Example : If the user wants the mark <red-car-s.1> to have the icon-color RED then he creates a color-petri object with the name **red-car-s.1** and sets the color attribute to RED.

Property objects.

Each property object represents a property of a petri object. A relation is created between the property object and the objects that have this property.

In the implementation four properties related with subtransitions have been defined: event, non-event, is-fireable and must-be-recalculated. Those properties are used in the evolution of the PN to gather subtransitions that can be fired or that may becomes enabled when a new mark has been created in a place at the input of the subtransition.

Place-zoom objects.

Place-zoom object is used when the user wants to zoom in a place. The relation **place-zoom-of** links a place-zoom object with the place.

2.4 Discoloration.

2.4.1 Why used a discolored version of the CPN.

CPNs are very good for describing very complex system with a small number of places and transitions. The graphical representation is more simple than the graphical representation of the same system with an ordinary PN. However, the calculations to determine if a transition is enabled are not very simple.

Each time that a mark is created in a place, the transitions that follow can become enabled. Hence a calculation of the enableness of the transition must be carried out for each firing color. This implies a lot of unnecessary calculations because there are only some firing colors that are sensible to the new mark. A way to link firing colors of transitions with color of marks that are in the input places of the transitions must be found. The solution adopted is to create a discolored version of the CPN where each place is decomposed into subplaces and each transition is decomposed into subtransitions.

In the first version of the tool box, this approach was not taken. The calculation of enableness took so long time that the real time requirements were not met and G2 was saturated. In the discolored version of the CPN, all these problems are resolved.

2.4.2 How the discolored version of the CPN is obtained.

Creation of subplaces.

It is easy to find the list of types of marks that a place can contain from the functions that are linked to the place. These functions give the mark to remove or the mark to create in the place. By analyzing the functions, G2 defines the set of colored marks that can be contained in the place. A subplace is created for each mark color. This subplace is related with the place and the color of the subplace is equal to the mark color.

Creation of subtransitions.

The creation of a subtransition is easier than the creation of a subplace. A set of firing colors is defined by the user for each transition. A subtransition is created for each firing color in this set. The subtransition is in relation with the transition and the color of the subtransition is equal to the firing color.

Creation of sublinks.

If an arrow exists between a place P and a transition T in the CPN, several sublinks are created between the subplaces related to P and the subtransitions related to T. The best way to explain how sublinks are created is to follow the explanation of the discoloration of the CPN of Fig 2.6.

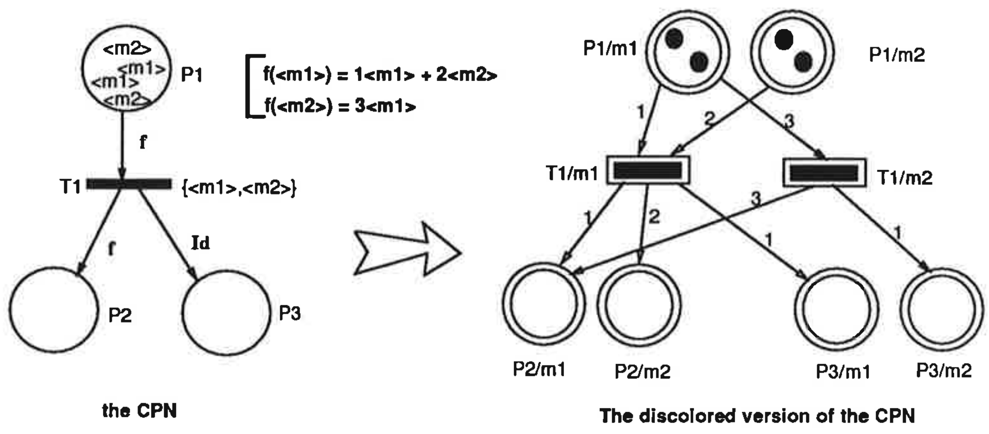
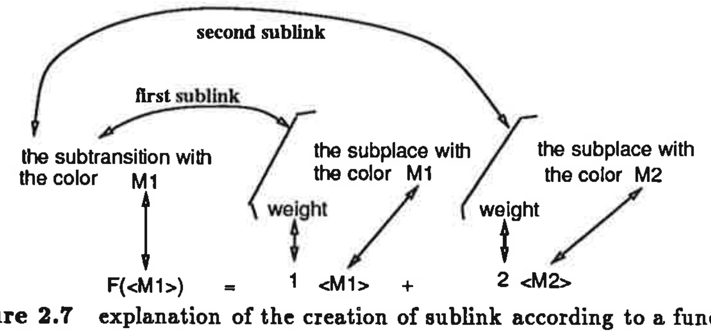


Figure 2.6 An example of discoloration.

1. Creation of subplaces: In the function **f**, there is 1<ml>+2<m2> and 3<ml>. So the P1 and P2 can only contain 2 types of marks:<ml> and <m2>. With the function **Id**, that is on the arrow between T1 and P3, G2 can deduct that P3 can only contain 2 types of marks:<ml> and <m2>. For this 2 types subplaces are created: Pi/mi is the subplace related with Pi and which has the color mi.

2. Creation of subtransition : T1 has two firing colors <ml> and <m2>. Hence two subtransitions are created: Ti/mi is the subtransition related with Ti with the color mi.

3. Creation of the sublinks : On the arrow between P1 and T1, there is the function **f**. f(<ml>)=1<ml>+2<m2> implies the creation of two sublinks: the first one links T1/ml with P1/ml with a weight 1; the second links T1/ml with P1/m2 with the weight 2. f(<m2>)=3<ml> implies the creation of a sublink that links T1/m2 with P1/ml with the weight 3. The same things are performed for each arrow.



second sublink
first sublink
the subtransition with the color M1
the subplace with the color M1
the subplace with the color M2
weight
weight
F(<M1>) = 1 <M1> + 2 <M2>

Figure 2.7 explanation of the creation of sublink according to a function.

2.4.3 Procedures used to discolor the CPN.

The discoloration is made automatically in G2 by the procedure **initialisation**. This procedure calls numerous other procedures. All these procedures are used as a formula analyzer. They transform the text of all the data into more convenient objects or translate the text into the action that it represents.

Main procedures.

- **trans-tra**(T:class transition) : This procedure works with the *color-set-def* of the transition T. It extracts each firing color from the text *set-color-def* of T and puts these firing colors in the list *color-set.*
- **create-subtransition**(T:class transition,C:text): This procedure creates the subtransition that is related with the transition T. The *color* of the subtransition is C. If T has a rule on its subworkspace then the subtransition is related with the property **event** else with the property **non-event** .
- **trans-out**(T-P-L : class transition-place-link) : Trans-out analyses the *output-function-text* of T-P-L. The link T-P-L connects a transition T to a place P. This procedure creates output-sublinks between the subtransitions related to T and the subplaces related to P based on the function. If the *output-function-text* is not a predefined function then the text is analyzed as in Fig 2.7. If the *output-function-text* corresponds to a predefined function then the function is applied to each firing color (fc) of T in order to have the mark color(mc). An output-sublink is created between the subtransition related with T and that has the *color* fc and the subplace related with P and that has the *color* mc. If this subplace does not exists then it is created. Hence, this procedure creates subplaces.
- **trans-inp**(P-T-L:class place-transition-link): This procedure analyses the *input-function-text* of P-T-L. The actions of this procedure are similar to the actions of **trans-out** but instead it creates input-sublinks. This procedure also creates subplaces.
- **trans-pla**(P:class place): This procedure analyses the text *initial-marking* of P in order to create the initial marking of the place and initialize the *nb-element-available* of the subplaces related with P. (This procedure calls the procedure **create-a-mark**)

Other procedures.

The main procedures call other procedures.

- **find-subplace** (P: class place, C: text) = class subplace : Find-subplace searches for the subplace that is related with P and that has the *color* C. If this subplace does not exists it is created. The subplace is returned by the procedure.
- **create-outlink**(ST:class subtransition,SP:class subplace,W:number): This procedure creates a output-sublink between SP and ST . The *weight* of this sublink is equal to W.
- **create-inputlink**(ST:class subtransition, SP:class subplace, W:number): This procedure creates a input-sublink between SP and ST . The *weight* of this sublink is equal to W.
- **value-function**(FU:text,COL:text)=text: This procedure gives the value of the function that is in the text FU for the color COL.
  (e.g.:**Value-function**("succl","convoyer.2") returns "convoyer.2")
- **trans-sum**(T:class transition,TEX:text): This procedure gives the list of firing colors *color-set* in the case that the *color-set-def* of T contains $\sum_{i=1}^{n}$.
- **trans-sum2**(P:class place,TEX:text): This procedure creates marks in the case that the *initial-marking* of P contains $\sum_{i=1}^{n}$.

The listing of all these procedures can be found in Appendix C at the end of the report.

2.5 Evolution.

In this section the G2-procedures that are used in the evolution of the Petri net are presented and the way to link these procedures together is discussed.

2.5.1 Evolution procedures.

State evaluation procedures.

- **calcul-subtransi**(ST:class subtransition): This procedure calculates the enableness of ST. The enableness is calculated by comparing the nb-element-available in the input subplaces of ST and the weight of the sublinks that connect those subplaces to ST. If for each input subplace, the first number is superior to the second one then the enable of ST= true else it is false.
- **change-implied**(ST:class subtransition): This procedure 'lists' the subtransitions that follows the output subplaces of ST. These subtransitions are related with the property **must-be-calculated**.
- **new-list-subtransi**(ST:class subtransition): This procedure helps to calculate the properties of ST. At first, it calculates the enableness of ST. If ST is enabled and is not linked to an event or a condition, then ST is related with the property **fireable**. If ST is enabled and is linked to an event or a condition then ST is related with the property **wait-event**.
- **enable-mark**(M:class mark): This procedure is used to carry out all the tasks when the mark M becomes available in a place. The *nb-element-available* of the subplace that is related with the place and has the color of the mark, is incremented. The procedure **new-list-subtransi** is called for each subtransition that follows this subplace. After this, a stabilization sequence is carried out.

Firing procedures.

- **create-a-mark**(P:class place,C:color): This procedure creates a mark. The *color* of the mark is equal to C. If C represents the name of a color-petri object then the mark is colored else it stays black. This mark is placed on P. If the *delay* of P=0 then the *nb-element-available* of the subplace corresponding to P and C is incremented. Otherwise the procedure **enable-mark** it started after the *delay* of the place.
- **delete-a-mark**(P:class place,C:text): This procedure deletes a mark that is available in P and that has the color C.
- **fire-subtransition**(ST:class subtransition): This procedure is used to fire the subtransition ST and the transition that is related to ST with the right firing color. At first, the enableness of the subtransition is calculated. If the subtransition is still enabled, it is fired. The *nb-element-available* in the input subplaces are decreased according to the weight of the input-sublinks. Some marks corresponding to the subplaces are deleted. Some marks are created in the places related to the output subplaces. The number of marks

to create or to delete is given by the weight of sublinks. The color of the marks is given by the color of the subplaces.
At the end, the procedure **change-implied** is called.

- **sequence-of-stabilization()**: This procedure is used to create a series of firing sequences consisting of all the subtransitions without any events or conditions. Hence, a firing sequence is made with all those transitions that can be fired. Then, the enableness of all the subtransitions that are sensitive to the new marking are calculated. Finally if a subtransition without event or condition is enabled, the procedure is recursively restarted.
- **fire-event**(ST:class subtransition): This procedure is used to fire ST. After this, a stabilization sequence is carried out.
- **fire-event1**(T:class transition,C:text): This procedure fires the subtransition that is related to T and that has the color C. After this, a stabilization sequence is carried out.

The listing of all these procedures can be found in Appendix C at the end of the report.

2.5.2 Some comments about evolution procedures.

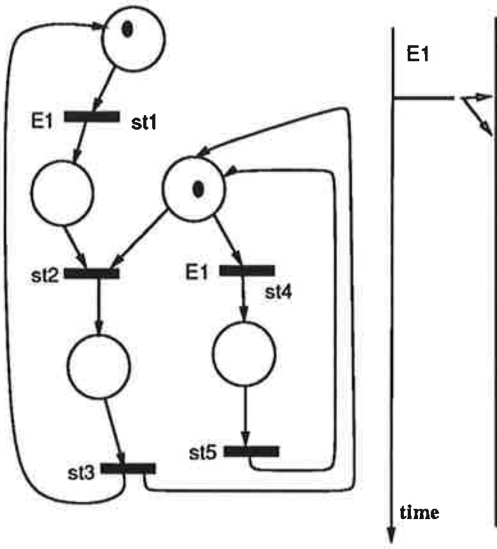
The important procedures are **sequence-of-stabilization**, **fire-events**, and **enable-mark**. These three procedures take care of the three means to start the evolution of the petri net:

- When a event occurs the procedure **fire-event** is started from a rule. This leads to the firing of the subtransition that is linked with the event. If several subtransitions are linked to this event, they are fired one by one. The subtransitions without event are only fired afterwards. The firing sequence on the occurrence of the event is created automatically by G2. The firing order of the subtransitions is deterministic but unknown to the user. It depends on the implementation of rules and of the order with which G2 checks rules. After the firing sequence on the occurrence of the event, G2 takes care of the subtransitions without events in the procedure **sequence-of-stabilization** that is started in **fire-event**.
- The procedure **sequence-of-stabilization** deals with the firing of subtransitions without event. This procedure carries out a firing sequence with the subtransitions that are enabled and restarts if there is a subtransition without event that is enabled after this first firing sequence. The recursive calling is made with a START. So this procedure can be interrupted by other procedures. Those interruptions are necessary to start the action procedures that can be linked to the subplace.

  When there is no more subtransitions without event that can be fired, the PN has reached a stable state. If an event occurs before the PN has reached a stable state, the event is taken into account and a firing sequence on the occurrence of the events is carried out. In this case, the rules of evolution of the PN are not very well respected. But this case can happen only if the sequence of stabilization is very long and it has never occurred in practice.
- The procedure **enable-mark** is the procedure that makes a mark available in a subplace and calculates the enableness of all the subtransitions that follow this subplace. It is the third case which can lead to an evolution since the new available mark may involve the enableness of a subtransition.

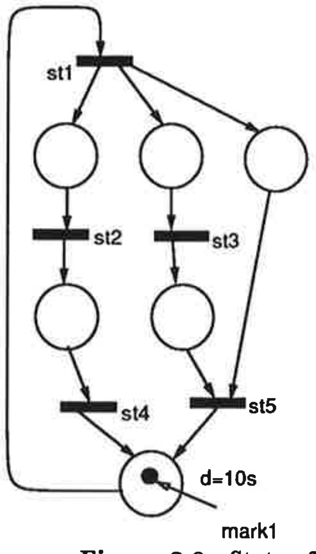
The enableness of a subtransition is only calculated if one of the subplaces at the input of this subtransition has received an available mark. This calculation takes place after the firing sequence. Hence the number of calculations is very low. The enableness is recalculated before the firing of the subtransition to be sure that the subtransition is still enabled. (In order to solve the problem of conflict)

The figures that follow show several cases of the state of the scheduler.



scheduler	result
fire-event(st1) *	st1 is fired
fire-event(st4) *	st4 is fired
sequence-of-stabilisation() *	st5 is fired
sequence-of-stabilisation() *	st2 is fired
sequence-of-stabilisation() *	st3 is fired
sequence-of-stabilisation()	nothing is done

* sequence-of stabilisation is started at the and of this procedure



scheduler	result
enable-mark(mark1) *	
sequence-of-stabilisation() *	st1 is fired
sequence-of-stabilisation() *	st2 and st3 are fired
sequence-of-stabilisation() *	st4 and st5 are fired
10 seconds	
enable-mark(mark1) *	
sequence-of-stabilisation() *	st1 is fired
sequence-of-stabilisation() *	st2 and st3 are fired
sequence-of-stabilisation() *	st4 and st5 are fired
⋮	

Figure 2.8 State of the scheduler during the evolution.

In fig 2.8, it is the discolored version of the PN that is presented. As seen, working in the discolored version is the same as working with a ordinary PN. The scheme represents the initial marking. Read the column result of the table to follow the evolution. The order of principal tasks are given in the column scheduler. This column is a summary of the scheduler of G2 with all the important tasks of the evolution.

2.6 Graphical interface.

It is possible to zoom a place. When the place is zoomed, a large place-zoom object is created. This place-zoom is the picture of the place but the marks are shown in their colors. It is a easy way to know what are the color of the marks in a place. The user has just to click on the place to zoom and select the menu choice ZOOM. A zoom is shown in fig 2.9.

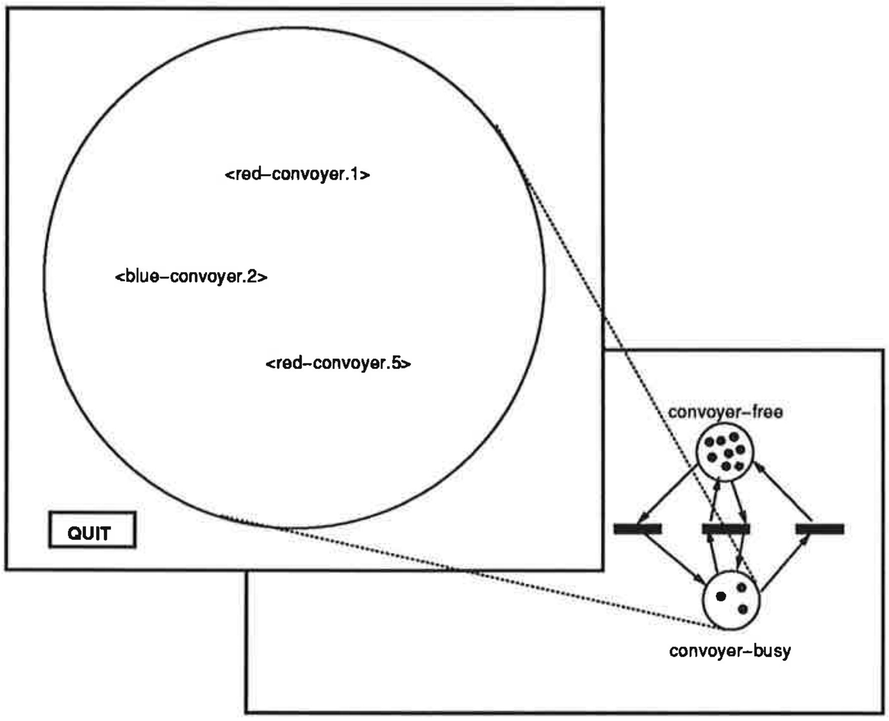


Figure 2.9 A zoom.

2.7 Size of the toolbox.

The colored Petri nets toolbox contains 15 object classes , 28 procedures, 10 rules, 10 relations, and 5 connection classes.

3. Examples

In this chapter three examples are presented: a model of a car race, a model of a queuing system, and a model of a manufacturing system. Only the first model is described. The other models are quite large and only the main parts are described in detail. In all the cases the evolution of the CPN is difficult to describe on paper. Hence, the animation of the CPNs is not presented.

3.1 Model of a car race.

3.1.1 Description.

The system is very simple. Three cars, one red, one blue, and one black, participate in a race. Each car starts the race when the button **start** is pushed. In the first part of the race, the cars drive to the right. After one kilometer they turn and go back. The race ends when the cars cross the start line again. The car speed can be changed during the race.

3.1.2 The model.

The system has been modeled with a colored Petri net in G2. The model is presented in Fig 3.1.

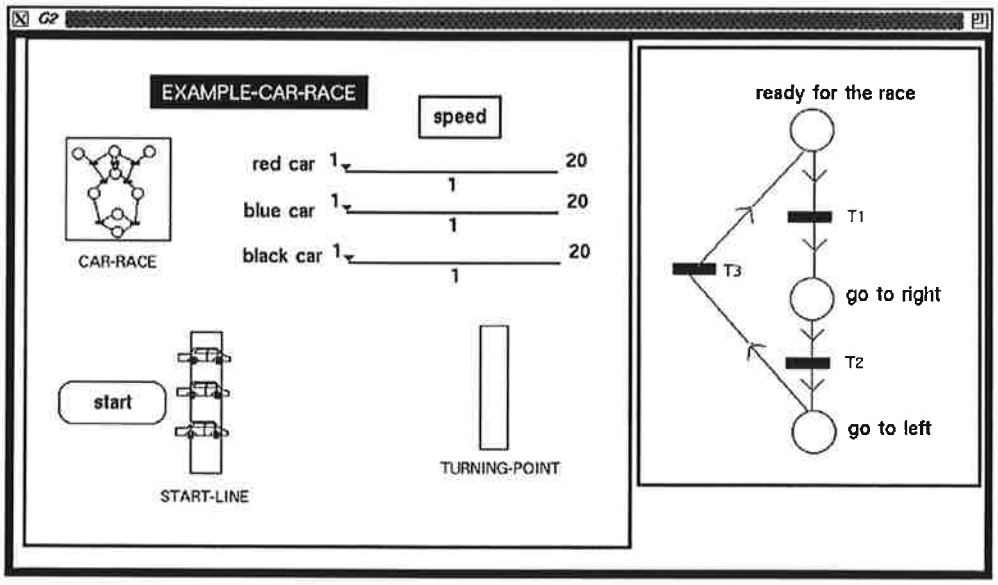
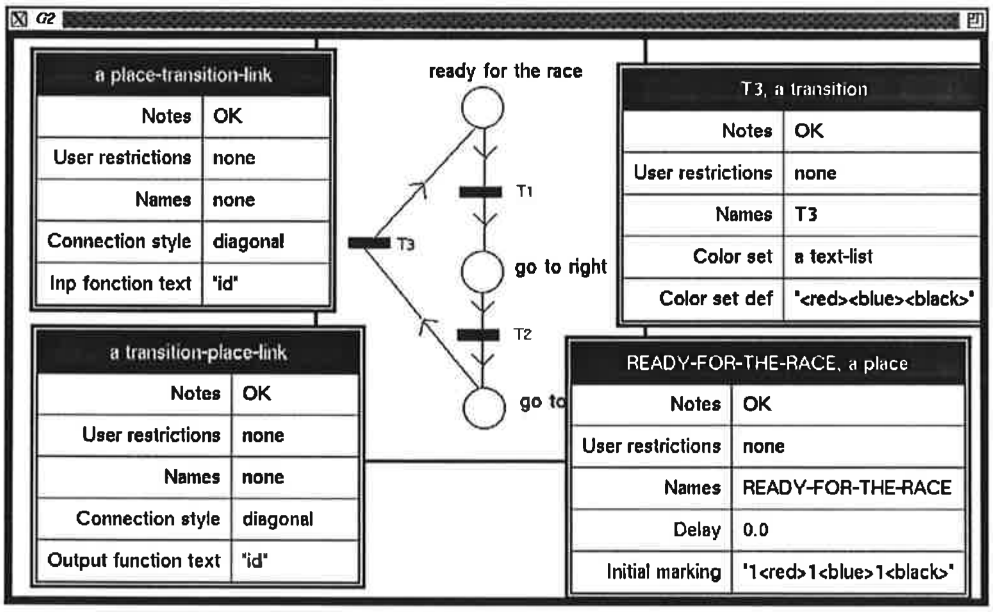


Figure 3.1 The system in G2 with its CPN.

Each car is represented by a colored mark. A mark with the color red represents the red car, etc.

Each transition has three ways to be fired (one way per car). Hence, three firing colors have been defined: red, blue, and black.



a place-transition-link

Notes	OK
User restrictions	none
Names	none
Connection style	diagonal
Inp fonction text	"id"

a transition-place-link

Notes	OK
User restrictions	none
Names	none
Connection style	diagonal
Output function text	"id"

T3, a transition

Notes	OK
User restrictions	none
Names	T3
Color set	a text-list
Color set def	"<red><blue><black>"

READY-FOR-THE-RACE, a place

Notes	OK
User restrictions	none
Names	READY-FOR-THE-RACE
Delay	0.0
Initial marking	"1<red>1<blue>1<black>"

Figure 3.2 Tables of petri objects.

In this simple system, all the functions on the arrows are the identity function. The tables of some of the Petri objects are shown in Fig 3.2.

The firing of transitions is linked with the occurrence of events. The event linked to the transition T1 is: the button START is pushed. The event is the same for all the firing colors of T1. When the user clicks on the button START, the parameter *ordre* gets the value true. Hence, the rule implemented in the subworkspace of T1 tests this parameter.

Each car has an attribute called *sensor*. When the car reaches the TURNING-POINT, the sensor of the car gets the value *turn*. When the car reaches START-LINE, the sensor of the car takes the value *end*. The attribute sensor is used in the rules implemented in the subworkspace of T2 and T3. These rules model the occurrence of the events required to fire T2 and T3 for different firing colors. The syntax of rules is:

```
T1:
     for any subtransition st that is waiting-event-to T1
              whenever ordre receives a value then
              start fire-event1(st)
T2:
     for any car c
              whenever the sensor val of c receives a value
              and
              when val="turn" then
              start fire-event(t2,"[the icon color of c]")
T3:
     for any car c
              whenever the sensor val of c receives a value
              and
              when val="end" then
              start fire-event(t3,"[the icon color of c]")
```

Actions are linked to the places called *go to left* and *go to right*. The actions move the car icons on the workspace. The rules that start those actions are:

```
GO-TO-LEFT:
        for any mark that is contained-by go-to-left
             unconditionally
             start move-car-left(the color of the mark)

GO-TO-RIGHT:
         for any mark that is contained-by go-to-right
             unconditionally
             start move-car-right(the color of the mark)
```

The listing of the procedure move-car-left is:

```
move-car-left(color:text)
    vehicule:class car;
 begin
   case (color) of
      "red"  : vehicule=car1;
      "blue" : vehicule=car2;
     otherwise:vehicule=car3;
   end;
   move vehicule by (- the speed of the vehicule,0);
   if the item-x-position of vehicule <= the item-x-position
     of start-line then
     begin
       conclude that the sensor of vehicule="fin";
       rotate vehicule by 180 degrees;
     end;
 end
```

3.2 Model of a queuing system.

3.2.1 Description of the system.

The queuing system models a conveyor with 10 compartments. The conveyor is linear and has a FIFO structure.(FIFO: First Input, First Output). The conveyor can carry two types of pieces: red ones and blue ones. The user controls the arrival of pieces in the conveyor with two buttons.

3.2.2 The model.

The conveyor has been modeled in G2 with the CPN that is shown in Fig 3.3.

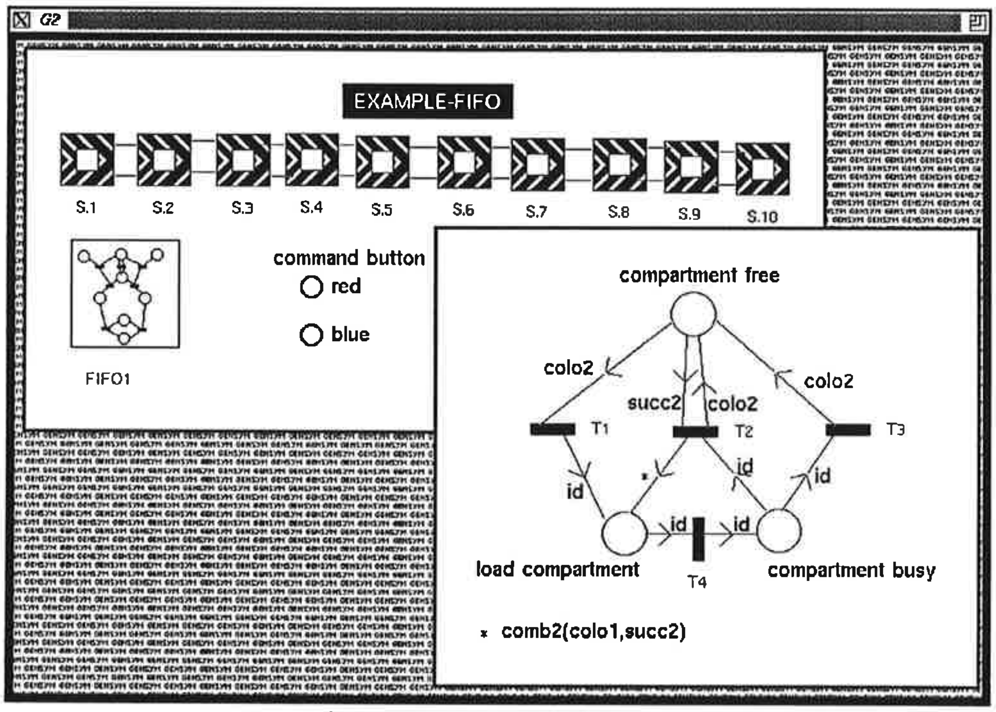


Figure 3.3 Model of a conveyor.

The CPN controls the different actions necessary for the motion of pieces on the conveyor. Hence, the CPN creates or deletes the appropriate colored piece in the appropriate compartment of the conveyor.

Places.

There is three places in the CPN.

- **Compartment free** represents the free compartments of the conveyor. The colored marks in this place indicate which compartments are free. The mark colors are indexed s_i (i=1 to 10). The place is linked to an action. When a mark appears in the place, i.e., the corresponding compartment becomes free, the action deletes the piece in the conveyor compartment given by the mark color.
- **Load compartment** represents the loading of a piece in a compartment. The mark colors in this place are complex: *red-s.i* and *blue-s.i* (i=1 to 10). The first part of the color indicates the piece color. The second part indicates the compartment. An action is linked with the place. When a mark appears

in this place, a piece must appear in the corresponding compartment. (The piece color and the compartment are given by the mark color). At the end of the action, the sensor of the compartment receives a value. This value is the mark color.

- **Compartment busy** represents the pieces which are on an compartment of the conveyor. The mark colors are identical to those in the place *load compartment.* The delay of the place models the speed of the conveyor. In our simulation this delay is equal to two seconds.

Transitions.

In the CPN, there are four transitions:

- **T1** models the arrival of one piece in the left-most compartment of the conveyor. T1 has two firing colors: *red-s.1* and *blue-s.1.* The firing of T1 is linked to an event for each firing color. The event occurs when the user pushes a command button.
- **T2** represents the motion of one piece from one compartment to the following one. The firing colors of this transition are complex: *red-s.i* and *blue-s.i* (i=1 to 9). The first part of the color indicates the piece color. The second part indicates the compartment from where the piece comes.
- **T3** models the output of the conveyor. T3 has two firing colors: *red-s.10* and *blue-s.10.*
- **T4** models the end of the loading of a piece. T4 is linked to an event that is the end of the loading of one piece. This event is represented by the fact that the sensor of an compartment receives a value. This value is the firing color with which the transition can be fired. The firing colors of T4 are complex: *red-s.i* and *blue-s.i* (i=1 to 10).

Functions.

In this model, a lot of predefined functions have been used. In Fig 3.3, the names of the different functions are shown together with the arrows.

3.3 A manufacturing system.

This system is the largest colored Petri net that has been implemented in G2. The size of the CPN is determined by the number of subtransitions and subplaces that there are in the discolored version of the CPN. In this case there are 242 subplaces and 182 subtransitions. The following example comes from the A.I.P.(Atelier Inter-universitaire de Productique) in Grenoble.

3.3.1 The description of the system.

The manufacturing system is shown in Fig 3.4.

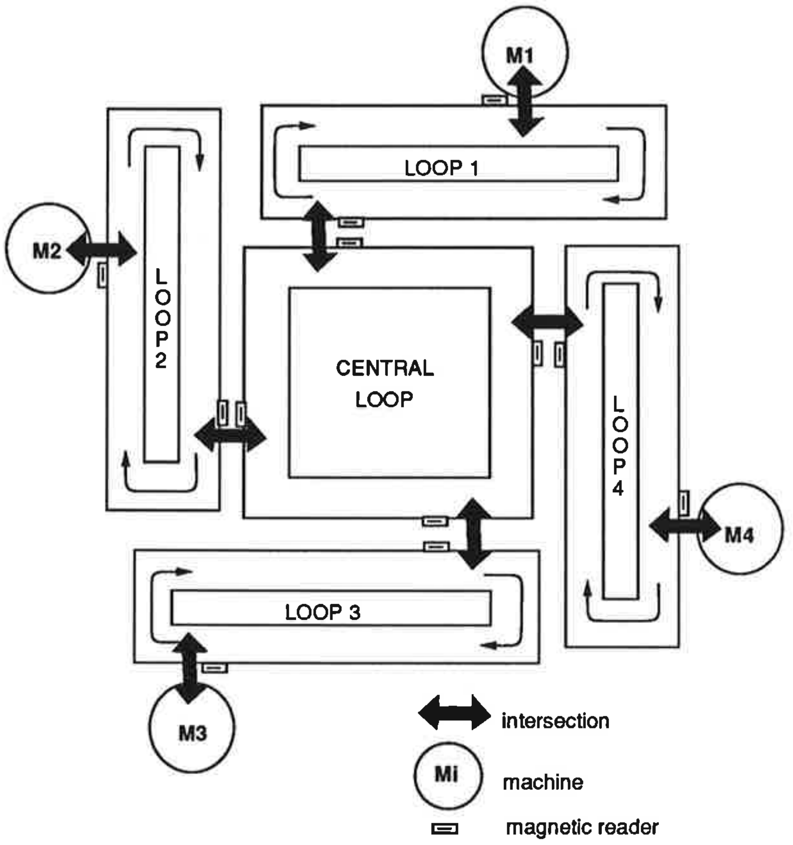


Figure 3.4 Scheme of the manufacturing system.

Loop 4 is the input-output loop of the system. Hence, pieces go in and go out through loop 4. Three types of pieces are produced in this manufacturing system:

. Red pieces first use machine 1, then machine 2 and after this they are ready and leave the system through Loop4.

. Blue pieces first use machine 2, then machine 3 and after this they are ready and leave the system through Loop4.

. Black pieces first use machine 1, then machine 3 and after this they are ready and leave the system through Loop4.

The arrival of pieces on the manufacturing system is controlled by the user through buttons. The conveyor capacity is limited to five pieces in each section. A section is limited by readers. In the external loops the section just ahead of

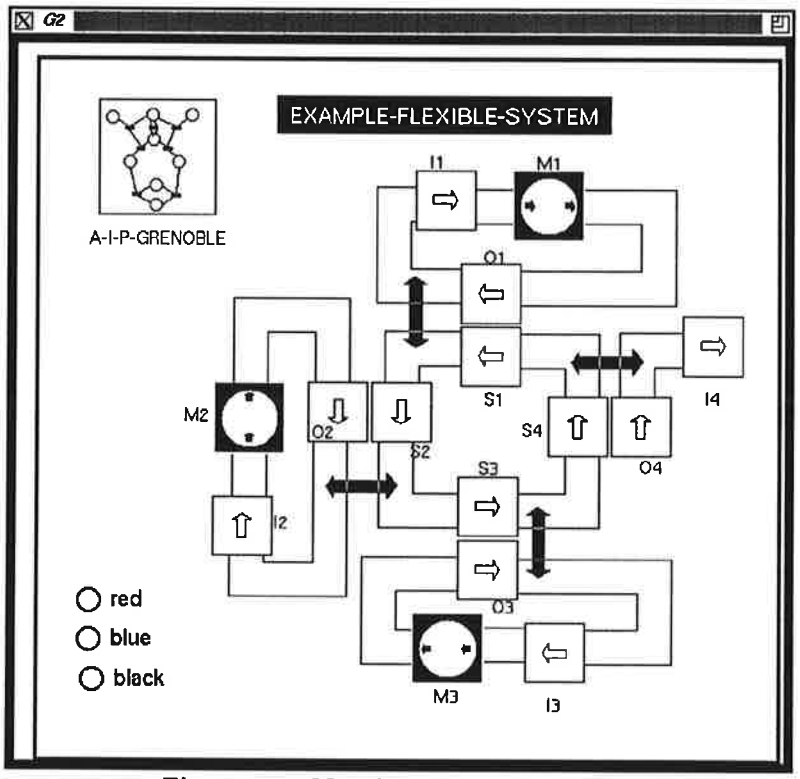


Figure 3.5 Manufacturing system model.

the machine is called input section; the section behind the machine is called output section. The time of work on a machine is ten seconds.

3.3.2 The model.

The model of this system is quite large and only the main part will be described. The G2 model of the manufacturing system and the corresponding CPN are shown in Fig 3.5and 3.6.

Conveyor sections.

As it is shown in Fig 3.5, the conveyor names depend on their positions in the system. **I** stands for input section, **O** for output section, and **S** for section of the central conveyor. The number that follows the name helps to differ the different sections in a same category.

Color marks.

- The machines are represented by three mark colors: m1, m2, and m3.
- The pieces are represented by three mark colors: red, blue, and black.
- The conveyor compartments are represented by index colors. E.g. , the conveyor compartment **I1** are represented by I1.i (i=1 to 5).

The fact that one colored piece is on a machine (or on an compartment) is modeled by a complex color that combines the mark color of the piece and the mark color of the machine (or the mark color of the compartment).

Places without actions.

The states of the machines are modeled with two places: *machine free* and *machine busy* . If the machine is free, a mark that represents this machine

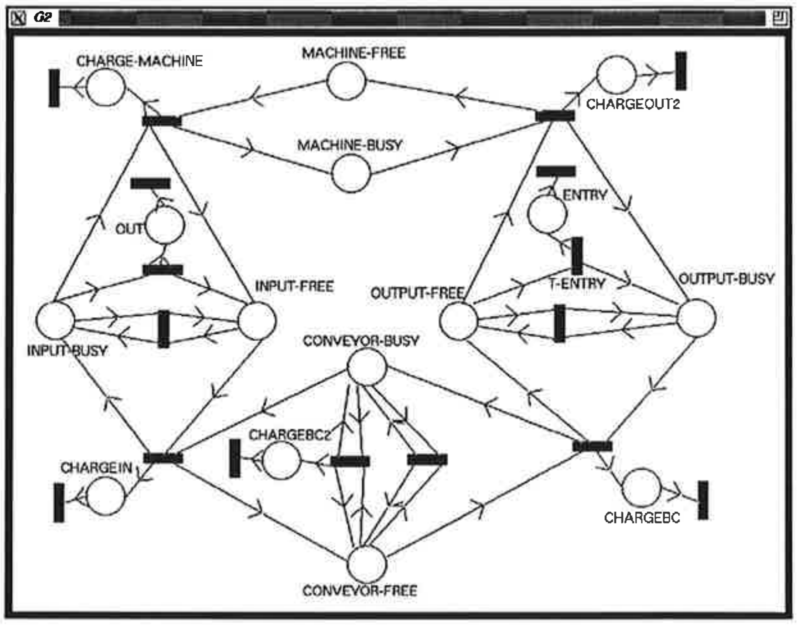


Figure 3.6 The colored Petri net.

is in the place *machine free.* Otherwise a mark with a complex color that represents the piece and the machine is in the place *machine busy.*

The state of the conveyors I1, I2, I3, I4 is given by the places *input busy* and *input free.* The state of the conveyors O1, O2, O3, O4 is given by the places *output busy* and *output free.* The state of the conveyors S1, S2, S3, S4 is given by the places *conveyor busy* and *conveyor free.*

Places with actions.

The actions in the simulation implement the motion of pieces in the manufacturing system model. The actions must be carried out when a piece moves to an other section of a conveyor, when it is loaded on a machine, when it enters the manufacturing system or when it exits the manufacturing system.

To model the actions, places with actions have been created. These places follow the transitions that model the different changes.

For example, in Fig 3.6, **Entry** is associated with an action that creates a piece and puts this pieces on the conveyor O4. The place **chargebc** is associated with an action that moves a piece from an output conveyor to a central conveyor, etc.

A rule and a procedure are implemented to carry out the actions. The actions depend on mark colors. The mark color indicates from where the piece comes, or what is the color of piece to created, etc.

Transitions.

T-entry is the only transition that is linked with an event. This event is linked to the command button used by the operator to order the production of one colored piece. Like the previous example, pushing a button makes the variable *ordre* have a value. The value of *ordre* gives the firing color with which the transition can be fired. This event is tested with a rule implemented in the subworkspace of the transition. This rule is :

```
whenever ordre receives a value then
      start fire event(t-entry,ordre)
```

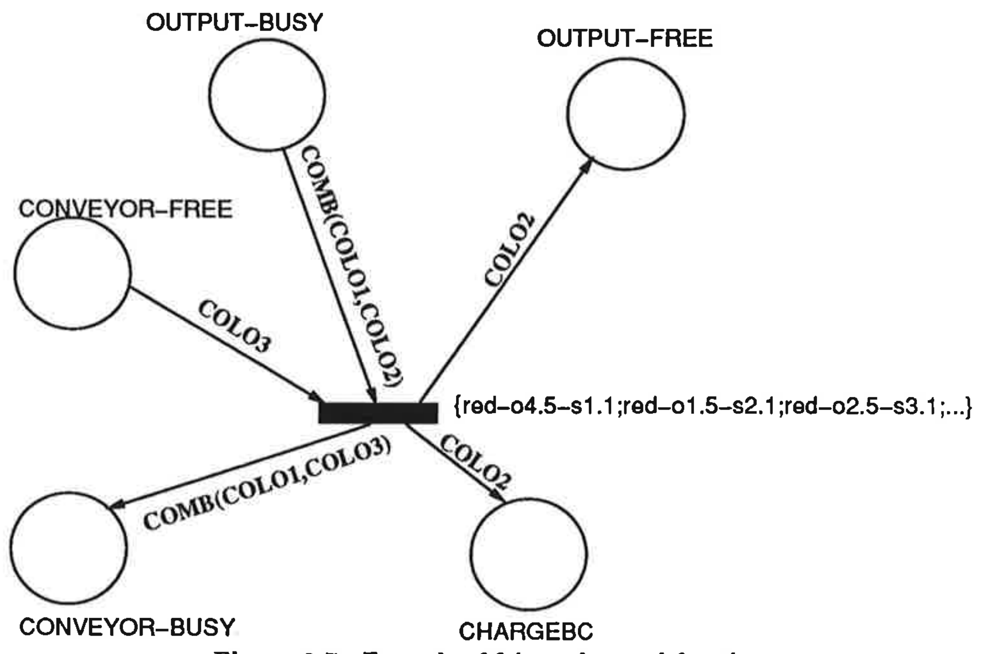


Figure 3.7 Example of firing colors and functions.

The firing colors of the transitions are sometime very complex. The use of these complex colors simplifies functions and permits the use of predefined functions. An example is given in Fig 3.7.

In Fig 3.7, the transition that models the move from an output convoyer to a central conveyor is shown. The firing colors of this transition are composed of three simple colors. The first one indicates the color of the piece, the second one indicates the compartment where the piece is, and the third color indicates the compartment where the piece will go after the firing. The functions associated with those firing colors are shown in Fig 3.7. Those functions are very simple and are rapidly implemented.

3.3.3 Evolution.

With this example, the toolbox has been tested with a large system. The aim of the tool box was reached. The evolution of the colored Petri net follows the real time requirements and the rules of evolution are respected.

4. Conclusions

A toolbox for colored Petri nets has been implemented in G2. The user can build Petri net quite easily. The way of entering the initial marking, firing colors, and functions is simple. The syntax used to enter all the data is close to the usual syntax used in the representation of colored Petri nets on paper. All the facilities for building a Petri nets are in the toolbox: several predefined functions are contained in the toolbox. A short way to enter a series of indexed colors is available.

The discolored version of colored Petri nets makes an evolution in real time possible. This evolution is implemented using procedures. The evolution of colored Petri nets has been tested through some applications. Those applications have permitted a validation of the toolbox.

Multiple extensions to this tool box can be added:

- Procedures can be built in order to detect the incorrectly designed of colored Petri nets.
- The discolored version of the colored Petri nets can be used to find the properties of colored Petri nets automatically.

This last extension makes it possible to verify and validate the model of a real system. Hence, research on properties of colored Petri nets can be useful for the designer.

G2 is really well adapt to this kind of toolbox. Firstly, its ways to treat simultaneous event is powerful. Secondly, G2 has a very nice graphical interface that is in the case of a tool like colored Petri nets. The drawback of G2 in this application is that the objects created dynamicly can not have name. So its quite difficult to refer to those objects and the speed of execution is shrink because of this drawback. The second drawback is that it is not easy to refer to an element of a list. Sometime the using of lists could facilitate the representation of data.

5. Résumé en francais.

Dans ce chapitre, les éléments essentiels de la boîte à outils pour réseaux de Petri colorés sont rappelés. Ainsi on trouve une brève description des objets permettant la construction d'un réseau de Petri coloré. La construction de la version dépliée du réseau coloré est brièvement évoquée. Enfin les procédures principales servant à l'évolution sont commentées.

5.1 Construction du réseau.

Chaque élément d'un réseau de Petri coloré a donné lieu à la création d'une classe d'objets.

On peut distinguer quatres objets principaux:

- la place,
- la transition,
- les connections *place-transition-link* et *transition-place-link*, et
- la marque.

Ces objets possèdent des attributs qui sont présentés dans le tableau de la figure Fig 5.1.

OBJET	ATTRIBUT	SIGNIFICATION
PLACE	DELAY	valeur de la temporisation
	INITIAL-MARKING	marquage initial de la place
TRANSITION	COLOR-SET-DEF	defintion de l'ensemble des couleurs de franchissement
	COLOR-SET	liste des couleurs de franchissement
MARK	COLOR	couleur de la marque
	TIME-OF-CREATION	instant de creation de la marque
TRANSITION-PLACE-LINK	OUTPUT-FUNCTION-DEF	definition de la fonction reliee a cet arc
PLACE-TRANSITION-LINK	INPUT-FUNCTION-DEF	definition de la fonction reliee a cet arc

Figure 5.1 Attributs des objets principaux.

Le marquage initial des places, l'ensemble des couleurs de franchissements des transitions, les fonctions reliées aux arcs sont donnés par un texte qui doit être entré dans les attributs correspondants. Ce texte doit être écrit avec une syntaxe prédéfinie qui est très proche de celle couramment utilisée par les utilisateurs de réseaux de Petri. Le texte est ensuite analysé pour créer d'autres objets plus commode à manipuler. Ce moyen d'entrer les données a de nombreux avantages. Il permet à l'utilisateur une grande liberté d'actions et rend très facile et très rapide l'entrée des données.

La boîte à outils étant surtout destinée aux controles de systèmes à événements discrets, il faut pouvoir relier les places à des actions et les transitions à des conditions ou des événements. Cela est fait grâce à l'implantation de règles dans les subwokspaces des transitions et des places.
Pour les règles liées aux transitions, la partie condition de la règle traduit les conditions ou l'occurence des événements liés à la transition et cela pour chaque couleur de franchissement. La partie action lance la procédure fire-event. Cette procédure aide à franchir la transition avec une certaine couleur de franchissement. Pour les règles liées aux places, la partie condition traduit le fait qu'une marque est contenue dans la place ou vient d' être créée dans cette place; La partie action lance la procédure qui modélise l'action liée à la place. Cette action peut évidement dépendre de la couleur de la marque.

Dans G2 chaque objet a une représentation graphique appelé icône. Ces icônes peuvent être reliées entre elles par des connections. Il est ensuite très facile de se référer à des objets géographiquement. Dans le cas de la boîte à outils pour réseaux de Petri colorés cela est particulièrement utile et permet de dessiner son réseau comme sur le papier. Chaque fois qu'un objet est créé, son icône apparait. Il suffit alors de la placer où l'on veut, de la relier avec d'autres icônes grâce aux connections disponibles. Ainsi, il est très aisé de décrire la géometrie du réseau puisqu'il suffit de le dessiner.

5.2 Construction de la version dépliée du réseau.

Le réseau de Petri coloré est un outil puissant pour modéliser des systèmes complexes avec peu de places et de transitions. La taille de ce réseau est réduite grâce à l'emploi de couleurs pour les marques et les transitions. Par la coloration, on enrichit donc les transitions et les places. Si cela réduit le nombre de places et le nombre de transitions, cela ne facilite pas le calcule de la validité des transitions. En fait à chaque fois qu'une marque apparait dans une place, elle peut permettre aux transitions qui la suivent d'être franchissables avec l'une de leurs couleurs de franchissement. Cela oblige à calculer la validité des transitions pour toutes les couleurs de franchissement et entraine beaucoup de calculs inutiles. En effet la couleur de la marque, en général, ne valide la transition que pour certaines couleurs de franchissement. Les calculs inutiles peuvent être évités en travaillant avec la version dépliée du réseau de Petri coloré. C'est le point de vue adopté dans ce projet.

Une version dépliée du RdPC est automatiquement construite grâce à des procédures. Elle est composée de trois types d'objets:

- les subplaces,
- les subtransitions, et
- les sublinks ***output-sublink*** et ***input-sublink***.

Ces objets possèdent des attributs dont le rôle est précisé dans le tableau de la Fig 5.2. La construction de ce réseau déplié se fait automatiquement en analysant les objets du RdPC:

- Chaque couleur de franchissement d'une transition entraine la création d'une subtransition qui est alors relié à la transition grâce à une relation et dont l'attribut *color* est égale à la couleur de franchissement.

OBJET	ATTRIBUT	SIGNIFICATION
SUBPLACE	COLOR	couleur de la marque
	NB-ELEMENT-AVAILABLE	nombre de marques disponible
SUBTRANSITION	COLOR	couleur de franchissement
	ENABLE	validite
OUTPUT-SUBLINK	WEIGHT	poids de l'arc
INPUT-SUBLINK	WEIGHT	poids de l'arc

Figure 5.2 Attributs des objets de la version dépliée.

- Chaque couleur de marques pouvant être contenues dans une place engendre une subplace qui est reliée à la place et dont l'attribut *color* est égale à la couleur de la marque.
- Chaque link entraine la création de plusieurs sublinks qui relient les subplaces et les subtransitions. La création de ses sublinks est guidée par la fonction contenu par le link. Cette fonction détermine aussi le poids du sublink.

Cela est illustré par la figure Fig 5.3.

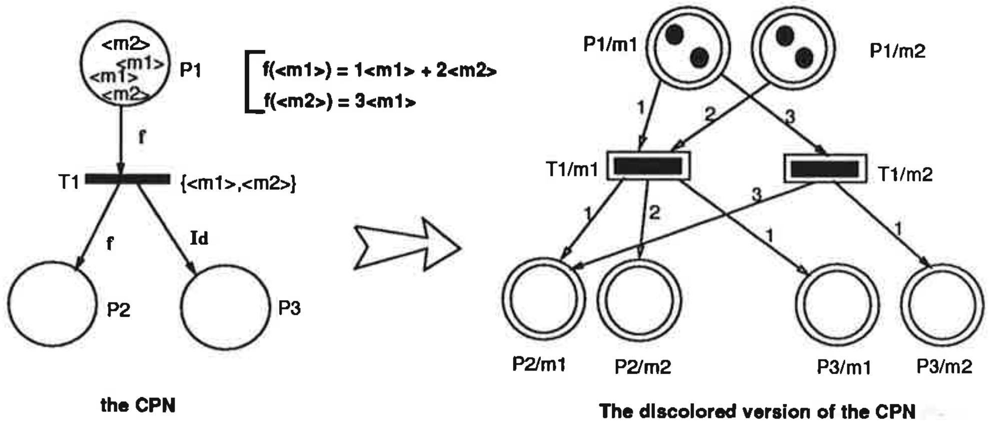


Figure 5.3 Example de dépliage.

5.3 Evolution.

L'évolution est calculée pour la version dépliée. On calcule la validité des subtransitions,... L'évolution du réseau coloré suit alors l'évolution de sa version dépliée. A chaque fois qu'une subtransition est franchie, la transition qui est en relation avec cette subtransition est franchie avec la couleur de franchissement correspondant à la subtransition. Ainsi des marques apparaissent dans les places en aval de la transition, et des marques sont détruites dans les places en amont de la transition. L'utilisateur voit donc une animation en temps réel de son réseau de Petri coloré.

L'évolution est effectuée grâce à des procédures. Ces procédures sont lancées grâce à des règles qui déclanchent le début de l'évolution, ou par d'autres procédures. Ces procédures servent à créer des marques, à détruire des marques, à lister les subtransitions susceptibles de devenir franchissables, à calculer la validité d'une subtransition, à franchir une subtransition, e.t.c. ...

Un algorithme très succinct des procédure principales est données dans la figure Fig 5.4.

5.3.1 Quelques précisions sur l'évolution.

L'évolution d'un réseau de Petri est dû aux franchissements des transitions. Dans le cas de réseaux de Petri interprétés, cet évolution a pour but de mener le réseau vers un état stable. A partir d' un état stable, le début de l'évolution correspond à l'un des trois cas suivants:

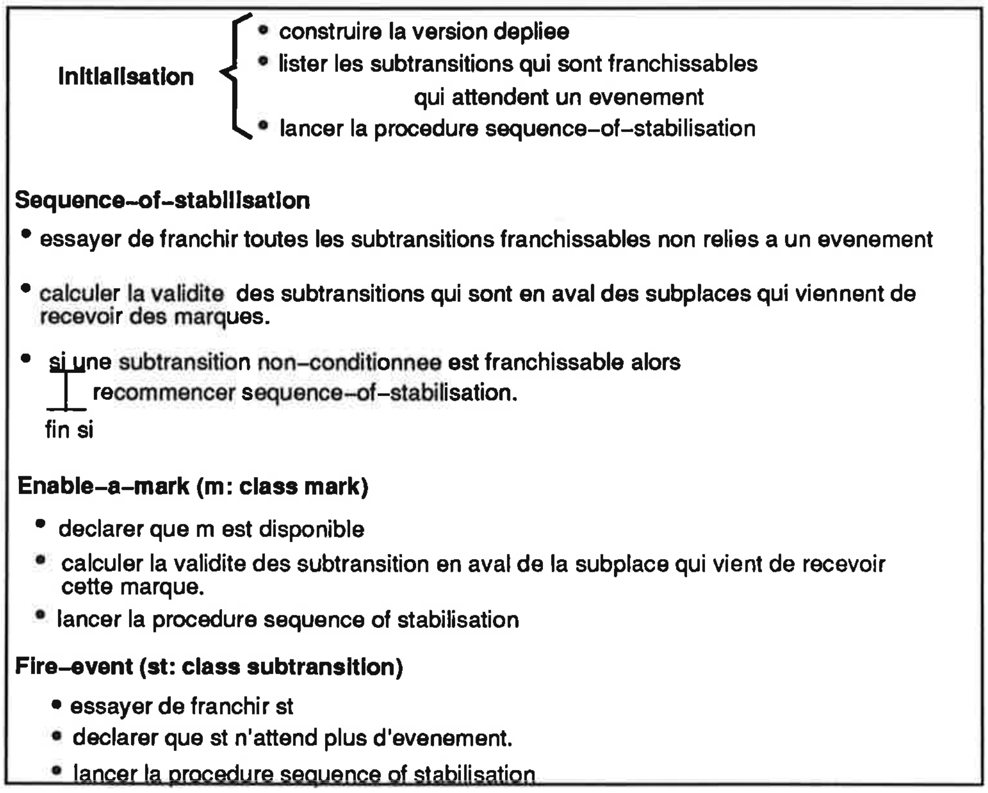
- La condition liée à une transition devient vrai.
- Un événement devient occurrent.
- Une marque devient disponible dans une place.

Dans les deux premiers cas, la règle traduisant l' événenement ou la condition lance la procédure *fire-event* . Cette procédure aide à franchir la subtransition associées à la condition, puis lance la procédure *sequence-of-stabilization* qui s'occupe du franchissements des subtransitions qui ne sont pas conditionnées. Cette procédure construit une séquence de franchissement avec toutes les subtransitions non-conditionnées. En fait, G2 essaie tour à tour de franchir les subtransitions qui été validées au début de l'appel de la procédure. La séquence se construit donc automatiquement en resolvant les conflits éventuels. Après cette première séquence, la validité des subtransitions qui suivent des subplaces ayant reçu des marques, est calculée. Puis si une subtransition non-conditionnée est franchisable alors on relance la procedure *sequence-of-stabilizatvion.*

Dans le troisième cas, une procédure *enable-mark* est lancée au moment où la marque devient disponible. Cette procédure est lancée à partir de la procédure *create-a-mark.* Quand une marque est créée dans une place temporisée, la procédure *enable-mark* est lancée après un intervalle de temps égal à la temporisation de la place. La procédure *enable-mark* calcule la validité des subtransitions qui suivent la subplace où la marque est devenue disponible et lance la procedure *sequence-of-stabilization.*

Si un même événement est relié à plusieurs subtransitions, la séquence de franchissement de ces subtransitions sur occurrence de l'événement s'effectue automatiquement par G2. En effet, G2 test l'occurrence de l'événement dans la partie condition de la règle, puis inscrit dans sous séquenceur l'action à effectuer. Dans notre cas, il inscrira: fire-event(st1),fire-event(st2),... où st1, st2,... sont les subtransitions franchissables sur occurrence de l'événement.

La validité d'une subtransition n'est calculée que si cette subtransition peut devenir validée; c'est à dire quand une des subplaces en amont de la subtransition reçoit une marque disponible. La validité est recalculée juste avant le franchissement d' une subtransition, pour confirmer la validité de la subtransition à franchir.



```
Initialisation
    • construire la version depliee
    • lister les subtransitions qui sont franchissables
                qui attendent un evenement
    • lancer la procedure sequence-of-stabilisation

Sequence-of-stabilisation
 • essayer de franchir toutes les subtransitions franchissables non relies a un evenement
 • calculer la validite des subtransitions qui sont en aval des subplaces qui viennent de
   recevoir des marques.
 • si une subtransition non-conditionnee est franchissable alors
       recommencer sequence-of-stabilisation.
   fin si

Enable-a-mark (m: class mark)
   • declarer que m est disponible
   • calculer la validite des subtransition en aval de la subplace qui vient de recevoir
     cette marque.
   • lancer la procedure sequence of stabilisation

Fire-event (st: class subtransition)
    • essayer de franchir st
    • declarer que st n'attend plus d'evenement.
    • lancer la procedure sequence of stabilisation
```

Figure 5.4 Algorithmes des procedures principales.

5.4 Applications.

Les applications ont permis de vérifier si les règles d'évolution des réseaux de Petri colorés étaient respectées. Elles ont permis de mettre au point la forme des règles qui modélisent les conditions et l'occurrence d'événements associées aux transitions.

Elles ont permis aussi de tester si l'évolution se faisait bien en temps réel et de valider la boîte à outils.

6. References.

S. AGAOUA (1987): *Specification et commande des systèmes à événements discrets, le grafcet coloré*, Thèse de doctorat de l'INPG, Grenoble.

K.E. ÅRZEN (1991): "Sequential function charts for knowledge-based, real time application," *Proc. 3rd International Workshop on Artificial Intelligence in Real Time Control, Napa/Sonoma, CA.*

K.E. ÅRZEN (1990): *User manual of Grafchart*, Lund Institute of technology, Lünd, Sweden.

R. DAVID ET H. ALLA (1990): *Autonomous and timed continuous Petri nets*, 11th International conference on application and theory of Petri nets, Paris.

R. DAVID ET H. ALLA (1989): *Du Grafcet aux réseaux de Petri*, Hermes, Paris.

R. DAVID (1991): *Cours de l'E.N.S.I.E.G sur les réseaux de Petri.*, E.N.S.I.E.G., Grenoble.

P. LADET (1991): *Cours de l'E.N.S.I.E.G sur les réseaux de Petri colorés.*, E.N.S.I.E.G., Grenoble.

R.L. MOORE, H. ROSENOF, and G.STANLEY (1991): "Process control using a real time system," *Proc. 11th IFAC World Congress, Vol7, Tallin, Estonia*, pp. 234–239.

R. MERCIER DES ROCHETTES (1988): *Sur l'utilisation des réseaux de Petri pour la commande des systèmes de production - Mise en oeuvre sur un atelier flexible*, Thèse de doctorat de l'INPG, Grenoble.

GENSYM CORPORARTION (1990): *G2 reference manual*, gensym corporation, Cambridge.

A. Appendix A : Petri nets

This part contains a very brief description of Petri net. It is largely inspired by the book written by David and Alla (1989).

A.1 Ordinary Petri nets (PN).

Carl Adam Petri is a German mathematician who in the beginning, of the sixties defined Petri nets, a tool for describing relations between conditions and events. Since then, a lot of research on Petri nets have been performed mainly within Europe. An international conference -the *European Workshop on Application and Theory of Petri Nets* - has taken place each year since 1980. Petri nets are especially used in data-processing and in automatic control.

A.1.1 Brief description.

A Petri net (PN) is a graphical net that is used to describe relations between conditions and events. A Petri net is composed of places and transitions which are linked together with arrows. A PN is shown in Fig A.1.

The state of a PN is given by the marking of the PN. Each place contains an integer number of marks (greater or equal to 0). The changes of the marking are due to the firing of transitions.

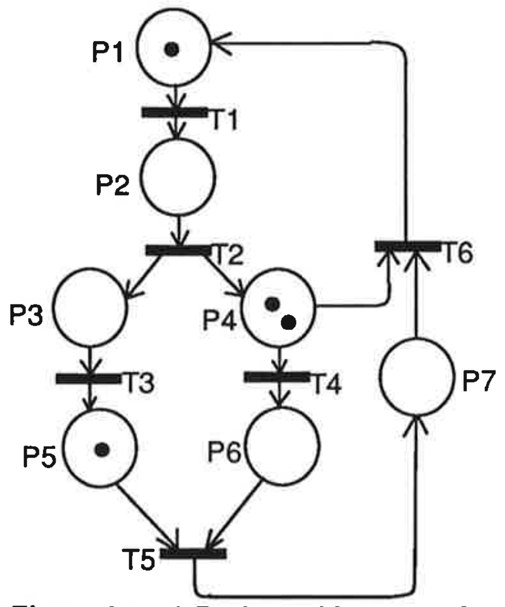


Figure A.1 A Petri net with some marks.

In the PN of Fig A.1, the place P1 is at an **input** of the transition T1 because there is an arrow from P1 to T1. The place P5 is at the **output** of T3 because there is an arrow from T3 to P5. Similarily we say that a transition is at an input or at an output of a place.

A.1.2 Firing of a transition.

The firing of a transition can be carried out only if every place at the input of the transition contains at least one mark. In this case the transition is said to be **enabled**. In the PN of the Fig A.1, T1 and T4 are enabled but T5 is not enabled because there is no mark in P6.

The firing of a transition consists of removing one mark in every place that is at the input of the transition and in creating one mark in every place at the output of the transition. This is illustrated in Fig A.2. Transitions are fired one by one.

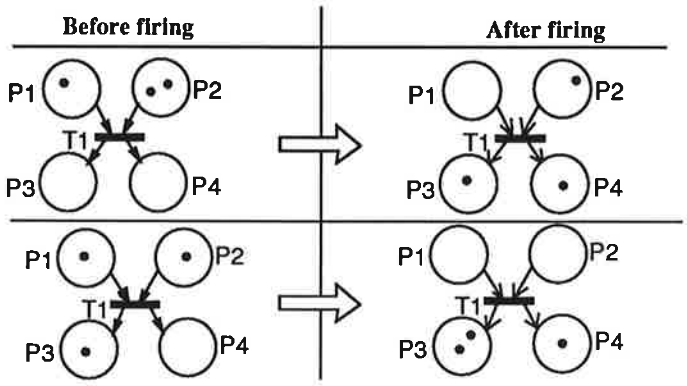


Figure A.2 Firing of a transition.

A.1.3 Autonomous and Non-autonomous Petri nets.

There are two types of PNs : autonomous and non-autonomous Petri nets.
Autonomous Petri nets describe systems that are autonomous: the firing instants are unknown or not indicated. All the PNs presented before are autonomous PNs.
Non-autonomous Petri nets describe systems in which the evolution depends on external events or time. Three kinds of non-autonomous Petri net have been used in the tool box.

- **Synchronized Petri nets.** In synchronized Petri nets, the firing of transitions depends on the occurrence of external events. If a transition is enabled and if the event linked to this transition occurs, then the transition can be fired. An event can be considered as a pulse and the transition is fired when the pulse appears. Hence, the firing is synchronized by the pulse.
- **P-temporized Petri nets.** In P-temporized Petri nets, each place has a delay. When a mark is created in the place, it is not available. It becomes available after waiting an amount of time equal to the delay of the place. When a mark is not available, it is as if it is not present in the place. Only available marks are taken into consideration in determining whether a transition is enabled or not.
- **Interpreted Petri nets:** An interpreted PN is P-temporized and synchronized. In addition, it has an operative part composed of internal variables, conditions linked with transitions and actions linked to places. The actions modify the values of the internal variables. Those variables can be parts of conditions. It is possible to implement actions that affect the external world. Those actions can be continuous actions or pulse actions. A continuous action is an action that is carried out as long as the place

contains at least one mark. A pulse action is carried out only when a mark is created in a place. The conditions can be linked with external variables. An example of an interpreted Petri net is given in Fig A.3.

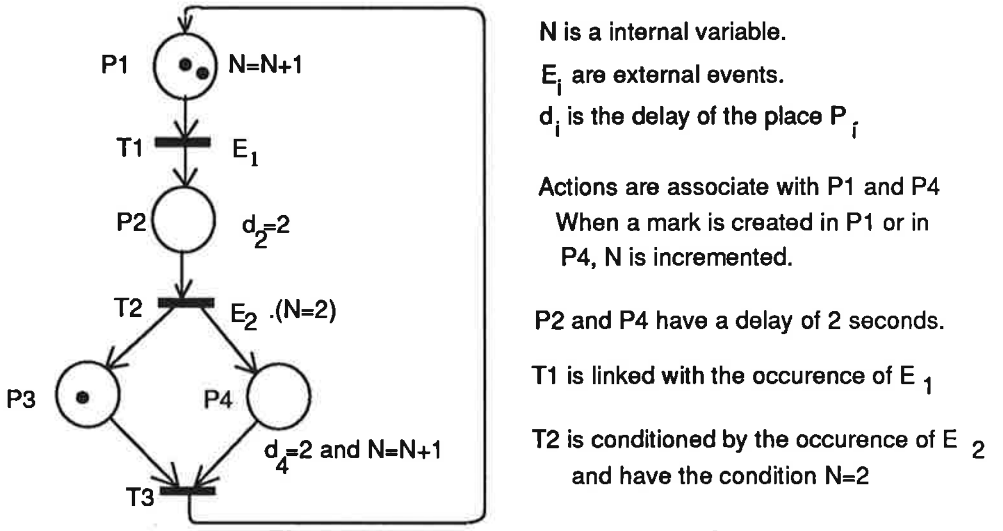

P1
N=N+1
T1
E_1
P2
$d_2=2$
T2
E_2 .(N=2)
P3
P4
$d_4=2$ and N=N+1
T3
N is a internal variable.
E_i are external events.
d_i is the delay of the place P_i
Actions are associate with P1 and P4 When a mark is created in P1 or in P4, N is incremented.
P2 and P4 have a delay of 2 seconds.
T1 is linked with the occurence of E_1
T2 is conditioned by the occurence of E_2 and have the condition N=2

Figure A.3 Example of Interpreted PN.

In Non-autonomous PN, the order of firing is important. The transitions are fired one by one. This can cause many problems, e.g., suppose that several transitions are enabled and are linked to the same event. What is the evolution when the event occurs? All the transitions cannot be fired simultaneously. In this case a firing sequence is defined consisting of all the transitions that are enabled and linked to the event. This firing sequence resolves the possible conflict between several transitions. This is explained in the following section.

A.1.4 Firing sequence and evolution of interpreted PNs.

Firing sequence. When several transitions can be fired in the same time a firing sequence is defined. This sequence indicates in which order the transitions will be fired and resolves the problem of conflicts. This sequence has some properties:

- It is only composed of transitions that can be fired at the time *t*.
- A transition can appear only once in the sequence.

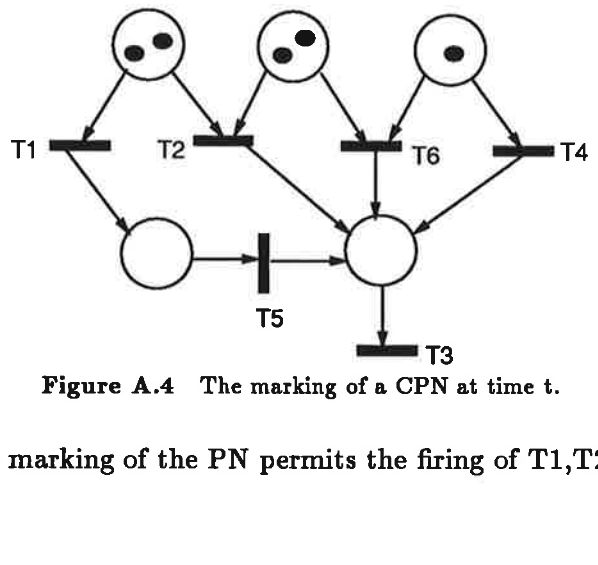


Figure A.4 The marking of a CPN at time t.

In Fig A.4, the marking of the PN permits the firing of T1,T2,T4 and T6. A firing sequence is defined with these transitions. For example T1-T6-T2. This means that first T1 is fired, then T6 is fired and finally T2 is fired. T4 does not belong to the firing sequence because there is a conflict between T4 and T2; only one of these two transitions can be fired.

Transition T5 becomes enabled after the firing of T1. However, T5 must not be fired before the end of the firing sequence. The transitions T1,T2,T4,T6 that can be fired at time t will be considered before the transitions that become fireable after time t.

Hence, at first list the transitions that can be fired at time t.

Secondly, try to fire all these transitions one by one (if there is no conflict all these transitions are fired).

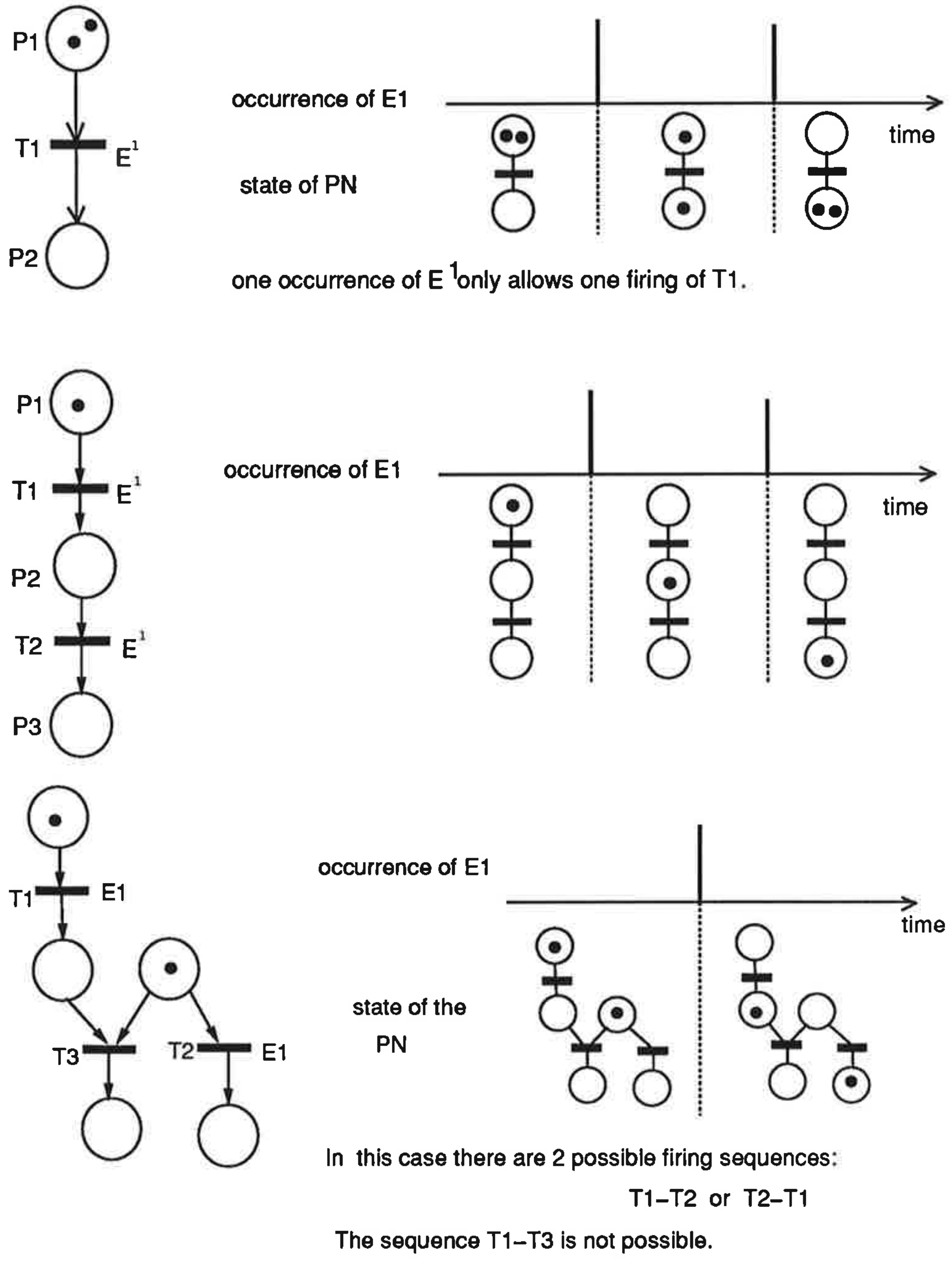
Since the firing of a transition is instantaneous, a firing sequence theoretically lasts 0 second

Evolution in time. When there is no transition that can be fired the PN is said to be **stable**. An evolution corresponds to the firing of transitions that leads to a stable state of a PN. There are only three reasons that can start an evolution of the PN:

- an event occurs,
- a mark becomes available in a place, or
- a condition linked to a transition becomes true.

- When an event occurs, a firing sequence is created with all the transitions that are enabled and that are linked with this event. After this first sequence, the state of the Petri net is checked. If the state is not stable a new firing sequence is carried out. If after this second firing sequence the state of the PN is not stable another sequence is carried out. In fact a series of firing sequences is carried out until a stable state is reached. This series of firing sequences is called stabilization sequence. Theoretically this operation takes no time. Hence no events can occurs during the stabilization sequence. Hence there is only transitions without events in the stabilization sequence.
- When a mark becomes available in a place, it may enable the transitions that are at the output of the place. So a firing sequence is carried out with the transition that can be fired after the mark has become available. If the state of the PN is not stable, this first firing sequence is followed by a series of firing sequences until a stable state is reached.
- If a condition of a transition becomes true and if the transition is enabled, then it can participate in firing sequences in order to reach a stable state. The difference between the condition and an event is that the event is a pulse. It is only true instantanously.

If an action is linked to a place that receives a mark the action must be carried out. Some examples of evolution are shown in Fig A.5.



one occurrence of E^1 only allows one firing of T1.

In this case there are 2 possible firing sequences:
T1–T2 or T2–T1

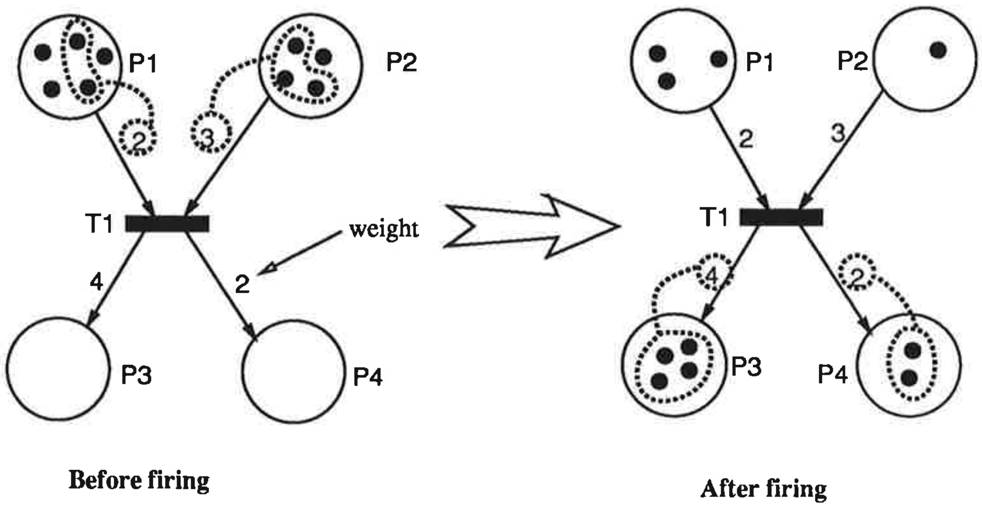
The sequence T1–T3 is not possible.

Figure A.5 Evolution of non-autonomous PN .

A.2 Generalized Petri nets (GPN).

In Generalized Petri Nets, the arrows have a weight. The weight of the arrow is an integer number that is greater or equal to 1.

In GPN, the number of marks in a place required to enable a transition can be different from 1. In the same way, the number of marks created or deleted in the place during the firing of a transition can be different from 1. The number is given by the weight of the arrow that links the transition to the place. An example of a GPN is given in Fig A.6.



T1 is enabled because :

There are 5 marks in P1 (5>2)

and

There are 4 marks in P2 (4>3)

Figure A.6 A firing in a generalized Petri net.

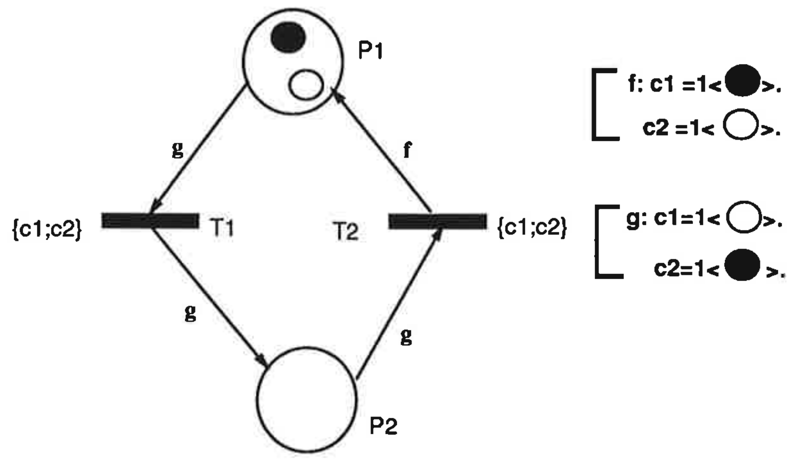
A.3 Colored Petri Nets (CPN).

A.3.1 Description.

In a colored Petri net

- each mark has a color,
- each transition has several firing colors which are linked to different ways of firing the transition, and
- a function which links each firing color of the transition with a set of marks is attached to each arrow.

The differences will be explained in the following. A colored petri net is presented in Fig A.7.



There are colored marks in P1: 1 black and 1 white.
T1 and T2 have two firing colors: c1 and c2.
f and **g** are functions which link firing colors with colored marks.

Figure A.7 Example of a colored PN.

Color of a mark. The color of a mark is an attribute that distinguishes each mark in a place. The color of mark can be simple, complex or with an index.

- A complex color is a color composed of several simple colors. For example a mark can have the color ***red-task***$_1$-***robot***$_2$, and the color indicates that there is a ***red*** piece that must go to ***robot***$_2$ to carry out ***task***$_1$.
- A color with an index makes it possible to describe a system in which elements are numbered. Index colors can be useful for modeling sequences of the same object. For example when modeling a conveyor with ten compartments, it is natural to represent each compartment with a number and hence the color of the mark can be *compartment*$_i$ (i=1 to 10).

Firing color. In a CPN, a transition can be fired in several ways. Each way of firing corresponds to a different color. In Fig A.7, there are two ways to fire T1: with the color C1 or with the color C2.

Firing a transition always consists of removing some marks in each input place of the transition and in creating some marks in each output place of the transition. For each firing color of the transition the number and the color of marks that are removed or created are different and are given by functions attached to the arrows.

A.3.2 Functions.

In a CPN, each arrow contains a function which links each firing color of the transition that is at a end of the arrow to a set of marks. An example is given in Fig A.8. In this example, f(<m1>) = 1<m1>+2<m2> indicates that the set of marks linked to the firing color *m*1 according to the function *f* consists of with 1 mark with the color *m*1 and 2 marks with the color *m*2.

The functions at the input of a transition are used to calculate the enableness of the transition, and to know which marks to remove during the firing.

The functions at the output of a transition are used to determine the marks to be created during the firing.

A lot of predefined functions are used:

- Id is the identity. Ex: Id(<red>)=<red>.
- Dec is the discoloration. It takes off the color.
  Ex: ∀ *color* Dec(*color*)=<.> (<.> represents a mark without any color).
- Succ is the function **successor of.** It is applied to colosr with an index. The function returns the successor of the color.
  Ex: Succ($< car_i >$)=$< car_{i+1} >$.
- Prec is the function **precede.** It is applied to a color with an index.
  Ex: Prec($< c_i >$)=$< c_{i-1} >$.
- For complex colors it is possible to suppress the i^{th} components with the function $Proj_i$. Ex: $proj_2(< c_1, c_2 >)=< c_1 >$.
- For complex colors it is possible to apply the function Succ (Prec) only for the i^{th} component with the function $Succ_i$ ($Prec_i$).
- Numerous functions can be obtained by combining these predefined functions.

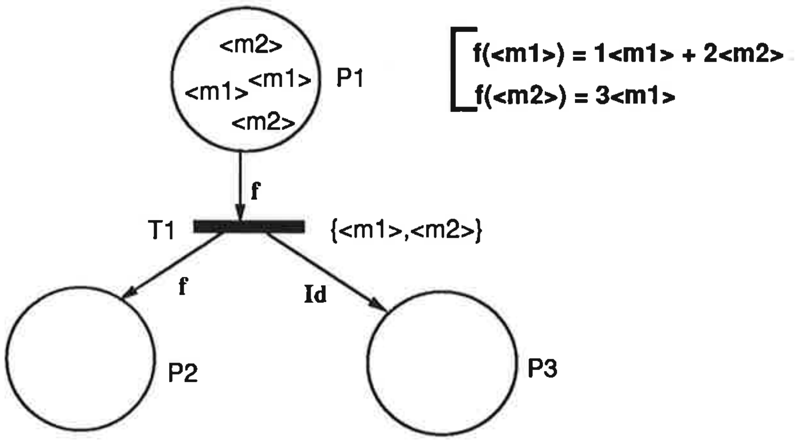


Figure A.8 Example of function.

A.3.3 The enableness of a transition with a firing color.

In a CPN, the same transition can be enabled with different firing colors. The enableness of a transition with a firing color depends on the marking of the places at the input of the transition. Each place must contain at least the marks that are given by the function on the arrow for the firing color. In Fig A.8, the transition T1 is only enabled with the firing color $m1$. f(<m1>)=1<m1>+2<m2> indicates that the set of marks linked to the firing color $m1$ according to the function f is composed of 1 mark with the color $m1$ and 2 marks with the color $m2$. This set of marks is included in the set of the marks that are in P1, so T1 is enabled with the color $m1$.

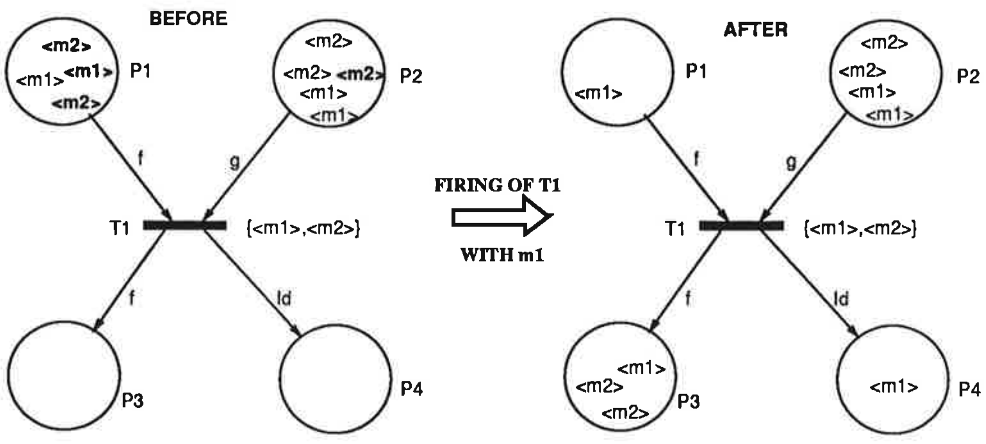
A.3.4 Firing a transition.

In a CPN, a transition is fired with a firing color. A transition can be fired with a firing color only if the transition is enabled with this firing color.

The firing of a transition T with the firing color C consists of removing some marks in each place P_i that is at the input of T and in creating some marks in each place P_o that is at the output of T.

The marks that must be removed in P_i are given by the function that is on the arrow which links P_i to T. This function is applied to C and gives the number and the color of marks that should be removed in P_i.

The marks that must be created in P_o are given by the function that is on the arrow which links T to P_o. This function is applied to C and gives the number and the color of marks that should be created in P_o. An example of firing is shown in Fig A.9.



f(<m1>) = 1<m1> + 2<m2>
f(<m2>) = 3<m1>

g(<m1>) = 1<m2>
g(<m2>) = 1<m1> + 2<m2>

T1 is not enabled with the firing color m2 because there is not 3 marks <m1> in the place P1

Figure A.9 Example of firing in a CPN.

A.3.5 Events, conditions, delays and actions in a CPN.

In a CPN, conditions and events can be linked to transitions in order to obtain an non-autonomous Colored Petri net. These events or conditions are linked to the different firing colors of the transition. Events (conditions) can be different for each firing colors of the transition. Hence, each event (resp:condition) is indexed with a firing color.

Delays can be implemented in each place of a CPN. The delay do not depend on the color of marks. Each mark that is contained in the place waits during the same time.

Actions can be linked to a place. The action to perform depends of the color of the mark.

The following example shows events and delays in a CPN.

EXAMPLE A.1—Events and delays in a CPN .

Assume that a CPN models a machine that can work with two types of pieces: piecel and piece2. The operator has two buttons : B1 to work with piecel and B2 to work with piece2. Whenever the operator pushes a button, if the machine is free and a piece that is asked by the operator is in the input store, the machine becomes busy and a piece linked to the button is loaded in the machine. After 10 minutes, the piece leaves of the machine and the machine becomes free. This problem is modeled by the CPN of Fig A.10. The events of the transition T1 are similar: it is the pulse when the operator pushes a button; but for each firing color the event is different because it comes from a different button.

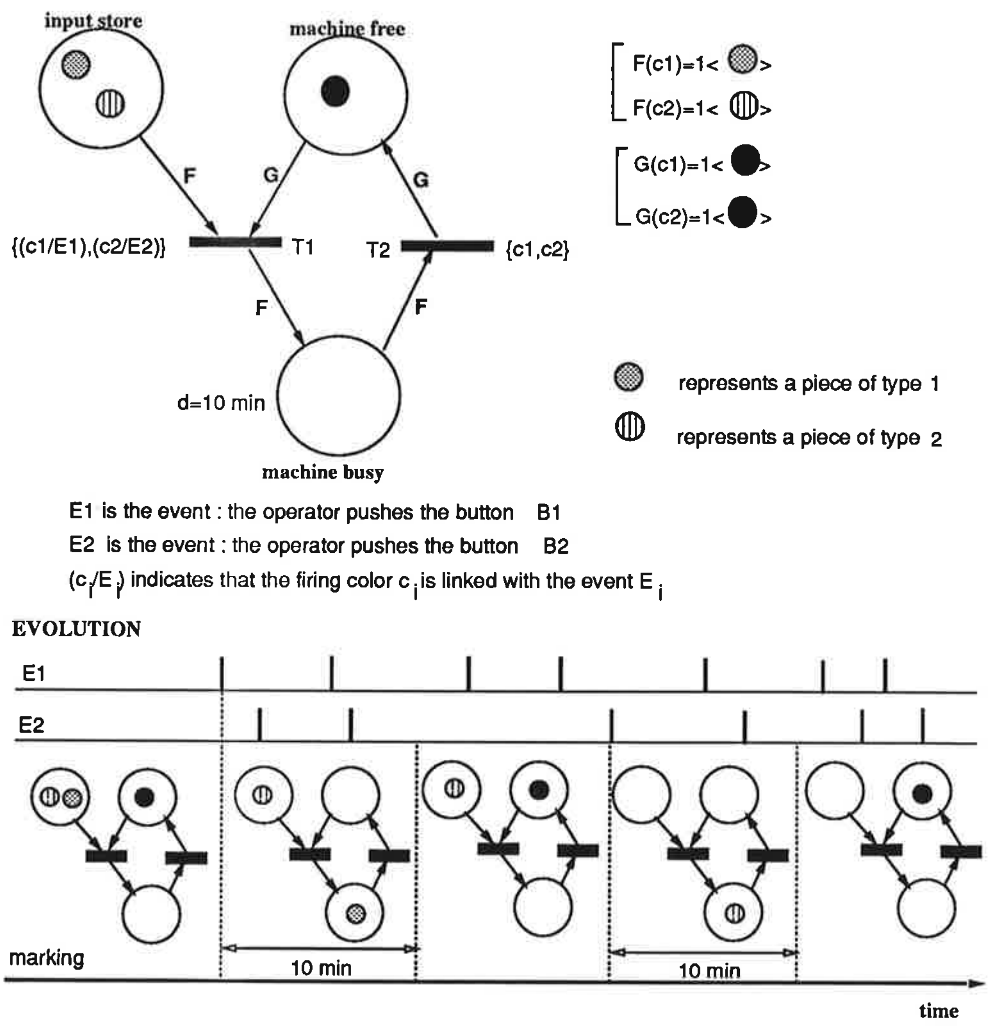

input store
machine free
F
G
G
{(c1/E1),(c2/E2)}
T1
T2
{c1,c2}
F
F
d=10 min
machine busy
F(c1)=1< >
F(c2)=1< >
G(c1)=1< >
G(c2)=1< >
represents a piece of type 1
represents a piece of type 2
E1 is the event : the operator pushes the button B1
E2 is the event : the operator pushes the button B2
(c_i/E_i) indicates that the firing color c_i is linked with the event E_i
EVOLUTION
E1
E2
marking
10 min
10 min
time

Figure A.10 Example of firing in a CPN.

A.3.6 Advantages of CPN.

Every CPN can be considered as a summary of a PN. If the number of colors is finite, then you always can turn a CPN into a PN. In the same way, all PNs can be turned into CPNs. A CPN makes it possible to model a system with fewer places and transitions than in a PN. Hence, CPNs are well adapted to modeling complex systems. For instance if you have to model several boilers where each boiler can be described by the same PN, it is easy to model all the boilers by only one CPN that graphically looks the same as the PN that describes one boiler and where a mark color represents each boiler. An example is given in Fig A.11.

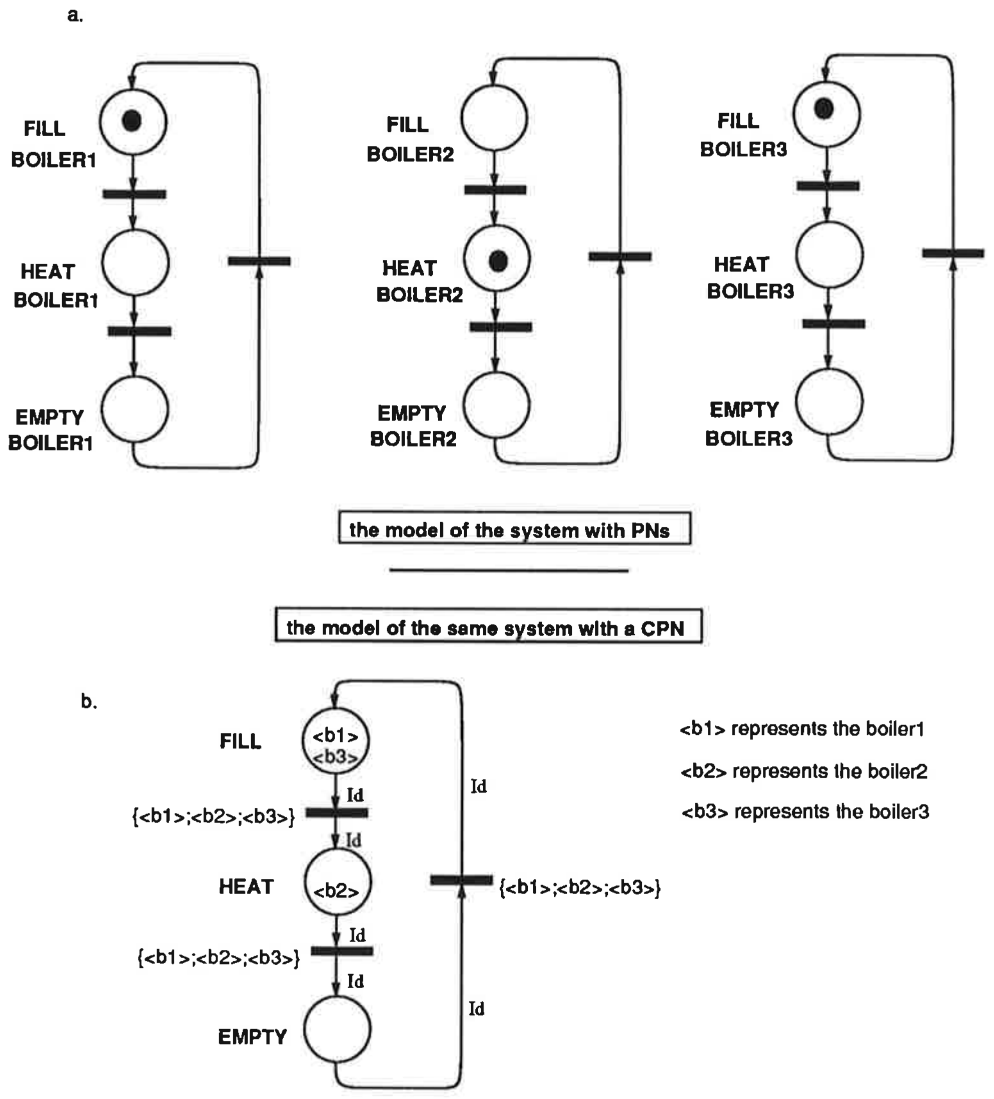


Figure A.11 CPN model.

B. Appendix B : G2

B.1 G2

G2 from Gensym Corporation in Cambridge, MA, is the most widely spread and technically most advanced real-time expert system tool available on the market. The number of licenses sold is over 700.

G2 is implemented in Common Lisp and runs on Sun 3 and 4, HP, Apollo, Vaxstation, Decstation, VAX 8600, TI Explorer and MicroExplorer, Symbolics, Compaq 386, and Mac II. For most machines, 16 MB RAM memory and X-Windows is required. The price ranges from \$ 18,000 to \$ 36,000 depending on computer.

B.1.1 Technical Description

The main parts of G2 are: the knowledge-base, a real-time inference engine, a procedure language, a simulator, the development environment, the operator interface, and optional interfaces to external on-line data servers.

Classes and objects: In G2 everything is an item, i.e, rules, objects, procedures, graphs, buttons, text boxes, etc., are all items. The items are organized into a hierarchy. All items have a graphical representation through which they are manipulated by mouse and menu operations. Operations exist for moving an item, cloning it, changing its size and color, displaying its attribute table, etc. One part of the item hierarchy is the G2 objects. The object is the only part of the item hierarchy that the user has full control over, i.e., can specialize into subclasses, can reference in expressions, etc.

Objects are used to represent the different concepts of an application. They can represent arbitrary concepts, i.e., both physical concepts such as process components and abstract ones. The objects are organized into a class hierarchy, i.e., only single inheritance is allowed. The class definition, or using G2 terminology, the object definition defines the attributes that are specific to the class and the look of the icon. Icons can be created with an interactive icon editor. The attributes describe the properties of the object. The values of an attributes may be

- constants,
- variables,
- lists, or
- other objects.

Constants can be numbers, symbolic values, i.e., the G2 correspondence to the enumeration type, logical values, i.e., true or false, and text strings. Under run-time, constants can only be changed explicitly by the user. Variables are used to represent entities whose values change during run-time. Variables are defined from four basic predefined classes:

- quantitative variables, i.e., real-valued variables,
- symbolical variables,
- logical variables, and
- text variables.

The predefined variable classes can be specialized by the user. With the predefined variable classes come a set of default attributes. These include attributes that determine whether a history should be saved for the variable or not; the current value for the variable; what should be the source of the variable's value, i.e., the inference engine, the simulator, or some external data server; the validity interval of the variable, e.g., how long the current value of the variable should remain valid; etc. The validity interval could be specified to be indefinite, given by a fixed time interval, or dependent on the validity intervals of the variables that were used to calculate the value of the variable. Variables with indefinite validity interval are called parameters and form separate classes. Parameters always have a current value and their initial values can be specified.

Lists may contain arbitrary values. It is possible to have objects as the values of attributes in other objects. In that case, the attribute objects have no iconic representation.

Objects can be static, i.e., they are explicitly created by the developer, or dynamic, i.e., they are created dynamically during run-time. Dynamic objects can also be deleted during run-time. The G2 language contains actions to move, rotate, and change the colour of an object. Using this, animations can be created.

Composite objects, i.e., objects that have an internal structure composed of other objects, can be created using objects as the value of attributes. It is, however, not possible to at the same time have a iconic representation for these objects. If such a representation is desired this has to be implemented using the subworkspace concept. In G2 each object and most items may have an associated subworkspace. On this (sub-)workspace arbitrary items may be positioned. The internal structure of an object can be represented on its subworkspace. It is also possible to connect together objects on the subworkspace with objects connected to the subworkspace's superior object. It is, however, not possible to define that an object should have an internal structure of this type in the class definition.

Connections and Relations: G2 has two ways of defining relations between objects: connections and relations. Connections are primarily used to represent physical connections, e.g., pipes or wires. It is, however, also possible to let connections represent abstract relations among objects. Connections have a graphical representation and may have attributes. They are defined in terms of a connection hierarchy. Both unidirectional and bidirectional connections are allowed. Type checking is performed to allow only connections of the same class to be connected together. Connections can be used in G2 expressions for reasoning about interconnected objects in a variety of ways. A connection is attached to an object either at a pre-specified location, a *port*, or anywhere on the object.

Relations can only be created at run-time and have no graphical representation. They have no corresponding relation hierarchy and cannot have attributes. Relations can be specified as being one-to-one, one-to-many, many-to-one, and many-to-many. The inverse relation of a relation can be specified, as well as whether the relation should be symmetric or not. In the latter case the inverse relation is the same as the relation. Relations can be used in G2 expressions in a similar way as connections.

The inference engine: G2 rules are used to encapsulate an expert's heuristic knowledge of what to conclude from conditions and how to respond to them. Five different types of rules exist.

- If rules
- When rules
- Initially rules
- Unconditional rules
- Whenever rules

When rules are a variant of ordinary 'If' rules that may not be invoked through forward chaining or cause backward chaining. Initially rules are run when G2 is initialized. Unconditional rules are equivalent to 'If' rules with the rule conditions always being true. Whenever rules allow asynchronous rule firing as soon as a variable receives a new value, fails to receive a value within a specified time-out interval, when an object is moved, or when a relation is established or deleted.

The rule conditions contain references to objects and their attributes in a natural language style syntax. Objects can be referenced through connections with other objects. G2 supports generic rules that apply to all instances of a class. The G2 rule actions makes it possible to conclude new values for variables, send alert messages, hide and show workspaces, move, rotate, and change color of icons, create and delete objects, start procedures, explicitly invoke other rules, etc. G2 rules can be grouped together and associated with a specific object, a class of objects, or a user-defined category. This gives a flexible way of partitioning the rule-base. The following is an example of a G2 rule,

```
for any water-tank
 if the level of the water-tank < 5 feet and
 the level-sensor connected to the water-tank is working
 then conclude that the water-tank is empty
 and inform the operator that
 "[the name of the water-tank] is empty"
```

The real-time inference engine initiates activity based on the knowledge contained in the knowledge base, simulated values, and values received from sensors or other external sources. In addition to the usual backward and forward chaining rule invocation, rules can be invoked explicitly in several ways. First, a rule can be scanned regularly. Second, by a focus statement all rules associated with a certain focal object or focal class can be invoked. Third, by an invoke statement all rules belonging to a user defined category, like safety or startup, can be invoked. The scanning of a few vital rules in combination with focusing of attention is meant to reflect the way human operators monitor a plant. It is also an important way to reduce the computational burden on the system.

Internally the G2 inference engine is based on an agenda of actions that should be performed by the system. The agenda is divided into time slots of 1 second's length. After execution, scanned rules are inserted into the agenda queue at the time slot of their next execution. Focus and invoke statements causes the invoked rules to be inserted in the agenda at the current time slot. Rules being invoked by forward chaining is treated in the same way.

A rule is invoked by backward chaining if the rule actions of the rule includes a conclude statement that gives a variable a new value, and if a new value for the variable is needed, i.e., the variable has a default update interval that specifies that the value should be recalculated regularly, the value is needed in a rule condition, or the value is needed in a display. Depth or breadth first backward chaining may be specified as well as the precedence order of the rules.

Simulation: G2 has a built-in simulator which can provide simulated values for variables. The simulator is intended to be used both during development for testing the knowledge base, and in parallel during on-line operation. In the latter case, the simulator can be used, e.g., to implement filters for estimation of signals that are not measured.

The simulator allows for differential, difference, and algebraic equations. The equations can be specific to a certain variable of apply to all instances of a variable class. Each first-order differential equation is integrated individually with individual and user-defined step sizes. The numeric integration algorithms available are a simple forward Euler algorithm with constant step size and a fourth order Runge-Kutta algorithm, also with fixed step size. GSPAN, an interface between G2's simulator and external simulators is available as a separate product.

Procedures: G2 contains a Pascal-style procedural programming language. Procedures are started by rule actions. Procedures are reentrant and each procedure invokation executes as a separate task. Procedures can have attributes and return one or several values. Local variables are allowed within a procedure.

The allowed procedure statements include all the rule actions, assignment of values to local variables, *If-then-else* statements, *case* statements, *repeat* statements, *for loop* statements that can either be numeric or generic for a class, i.e., they execute a statement or set of statements once for each instance of the class, *exit if* statements to exit loops, *go to* statements, and *call* statements to call another procedure and await its result. Procedures can be temporarily halted with a *wait* statement. A wait statement causes G2 to stop executing the procedure until either a specified amount of time has passed or a condition is met. It is possible to specify that two or more statements should be executed in parallel and that all iterations of a loop should be done in parallel.

Procedures are executed by G2's procedure interpreter. The procedure interpreter cannot be interrupted by other G2 processing, i.e., the inference engine or the simulator. Other processing is only allowed when the procedure is in a wait state. A wait state is entered when a wait statement is executed, when the statement *allow other processing* is executed, and when G2 collects data from outside the procedure for assigning to a local variable.

Real-time issues: Unlike the majority of expert system tools, G2 is designed for real-time operation. This shows in a number of different ways. The inference engine automatically sends out requests for variables that have become invalid and waits for new values without halting the system. Priorities and scan intervals can be associated with rules.

Regular scanning of rules and thus updating of information in combination with variables with time-limited validity gives a partial solution to the problem of non-monotonic, time-dependent reasoning. The validity interval of a variable specifies how long the current value of the variable should remain valid. By the possibility to propagate validity intervals to dependent, concluded variables their values will also eventually expire if for some reason sensor values cease to arrive to G2.

Whenever rules can be used to catch asynchronous events such as the arrival of un-requested sensor data from a passive sensor, e.g., representing some alarm in the underlying control system, or that a requested sensor value has failed to arrive to G2 within a specified time-out interval.

G2 has some facilities for temporal reasoning. It is possible to save histories of old values for all variables. Functions for referencing old variable values are available. G2 also has built-in statistical functions operating on quantitative variable histories. These are functions for computing the integral, standard deviation, rate of change, maximum, and minimum values over some time interval.

G2 also has possibilities for reasoning about whether a variable has a current value or not, and can refer to the time when a variable received its current value.

To avoid garbage collection, G2 takes care of the dynamic memory allocation and reallocation internally during run-time.

Development interface: G2 has a nice graphics-based development environment with windows (called workspaces), popup menus, and mouse interaction. Input of rules, procedures, and other textual information is performed through a structured grammar editor. The editor prompts all valid next input statements in a menu. Using this menu the majority of the text can be entered by mouse-clicking. It is, however, also possible to use the keyboard in an ordinary way. The editor has facilities for Macintosh style text selection, cut, paste, undo, redo, etc.

The Inspect facility allows the user to search through the knowledge base for some specified item. The user can go to the item, show all matching items on a temporary workspace, write them out on a report file, highlight them, and make global substitutions.

G2 has facilities for tracing, stepping, and adding breakpoints. The internal execution of G2 can be monitored using meters.

End-user Interface: G2 has facilities for building end-user interfaces. Colors and animation can be used. An object icon is defined as a set of layers whose colors can be changed independently during run-time. The meta-color *transparent* makes it possible to dynamically hide objects. Different user categories can be defined and the behaviour with respect to which menu choices that are allowed can be set for each category. It is also possible to define new menu choices.

G2 contains a set of predefined displays such as readouts, graphs, meters, and dials that can be used to present dynamic data. G2 also has a set of predefined interaction objects that can be used for operator controls. Radio buttons and check boxes can be used to change the values of symbolical and logical variables by mouse clicking. An action button can be associated with an arbitrary rule action which is executed when the button is selected. Sliders can be used to change quantitative variables and type-in boxes are used to type in new variable values.

External interfaces: G2 can call external programs in four different ways: using foreign function calls, and using GFILE, GSPAN, and GSI. On some platforms, external C and Fortran functions may be called from within G2. GFILE is an interface to external data files that allows G2 to read sensor data from the files. GSPAN is the interface between G2 and external simulators. GSI is Gensym's standard interface. It consists of two parts; one part written in Lisp that is connected to G2 and one part written in C to which the user can attach his own functions for data access. On the same machine, the two parts communicate using interprocess communication media such as pipes or mailboxes. On different machines, TCP/IP – Ethernet is used.

GSI is the base for several off-the-shelf interfaces between G2 and conventional control systems, PLC systems, relational databases, etc. The Travel Notes in Appendix B contains reports from User Group Meetings where details about the existing interfaces are presented.

Networking: Several G2s can be used in a network exchanging data over Ethernet. Telewindows is a separate product that allows multiple users to access the same G2 system, giving each user his own window into the application. It is also possible for one Telewindows user to simultaneously access several G2s in a network.

B.1.2 Drawbacks

The main problems with G2 stem from the fact that G2 is a closed system. G2 can only be interfaced with other program modules through the predefined interfaces. The G2 environment in itself is also a quite closed world. It is impossible to modify the way that G2 operates internally. If what G2 provides in terms of, e.g., graphics, class – object structures, etc., is insufficient nothing can be done about it.

G2 can not be modularized. Hence, it requires quite powerful computers even if only a small subset of the functionality is used within an application.

Although G2 is fast compared to many expert system tools, it can be too slow for certain applications. The smallest time unit is one second. For applications that require faster response, G2 is inadequate. Gensym claims that G2 is capable of running between 300 and 500 medium-sized rules per second depending on the machine that is used. These figures are difficult to verify. If the simulator is used, the speed decreases substantially. Gensym are currently developing a run-time version that uses compiled rules and procedures instead of interpreting them, as is currently done.

C. Appendix C : Procedure listing

```
              FIN-INIT, a procedure

fin-init(p-n: class petri-net )
begin
    conclude  that the status of p-n is active;
    change the icon-color of p-n to green;
end

              ACTION, a procedure

action()
    st:class subtransition;
begin
    for st = each subtransition do
        call new-list-subtransi (st);
    end;
    start sequence-stabilization();
end

                RENIEW, a procedure

reniew(p:class place)
    st:class subtransition;
    sp:class subplace;
begin
    for sp = each subplace that is in-relation-
       with p do
        for st=each subtransition connected at an
           output of sp do
            start new-list-subtransi (st);
        end;
    end;
    start sequence-stabilization ();
end

             CREATE-A-MARK, a procedure

create-a-mark (p: class place, c:text )

{ this procedure creates a mark m in the place p
   with the color c;if there is a color-petri
   linked with c then the icon-color of m is this
   color-petri,else the icon-color of m is black}

    m: class mark;
    cp: class color-petri;
    sp:class subplace;
begin
    create a mark m ;
    conclude that the color of m = c;
    sp=call the-subplace (p,c);
if there exists a color-petri cp such that ("[the
   name of cp]" = c) then
    change the icon-color of m to the color of cp
else
    change the icon-color of m to black;

conclude that m is contained-by p;
transfer m to the workspace of p at (around-
   x(p),around-y(p));
conclude that the time-of-creation of m = the
   current time;
if  the delay of p =0 then
 begin
    conclude that the nb-element-available of sp
       =the nb-element-available of sp +1;
    conclude that m is available-in p;
 end
else
    begin
        start enable-mark (m) after the delay of
           p;
    end;
end
```

DELETE-A-MARK, a procedure

```
delete-a-mark (p: class place, c: text)

  {this procedure delete  a mark m which has the
     color c in the place p}

    m: class mark;
begin
      if there exists a mark m that is available-
         in p such that (the color of m =c) then
            begin
                 delete m
            end;
end
```

ZOOMER, a procedure

```
zoomer(pz: class place-zoom)
    p: class place;
    ele:text;
    m: class message;
    ma:class mark;
begin
    for m= each message that is a-message-in pz do
        delete m;
        end;
    p= the place that is zoomed-by pz;

    for ma = each mark that is contained-by p do
       create a  message m;
        change the text of m to the color of ma;
         transfer m to the workspace of pz at
            (around-x (pz), around-y (pz));
        conclude that m is a-message-in pz;
    end;
end
```

ENABLE-MARK, a procedure

```
enable-mark(m:class mark)
    p:class place=the place that is place-
       contening m;
    sp:class subplace;
    st:class subtransition;
begin
    conclude that m is available-in p;
    sp=call the-subplace (p,the color of m);
    conclude that the nb-element-available of sp =
       the nb-element-available of sp + 1;
    for st =each subtransition connected at an
       output of sp do
        call new-list-subtransi (st);
     end;
    start sequence-stabilization ();
end
```

FIRE-SUBTRANSITION, a procedure

```
fire-subtransition(st:class subtransition)
    i-l:class input-sublink;
    o-l:class output-sublink;
    sp:class subplace;
    nb:quantity;
    i:quantity;
    t:class transition= the transition that is
       related-with st;
begin
call calcul-subtransi (st);
if the enable of st then
 begin
   for i-l= each input-sublink connected at an input
```

```
      of st do
       sp=the subplace at the input end of i-l;
       conclude that the nb-element-available of
          sp=the nb-element-available of sp - the
          weight of i-l;
        for i=1 to the weight of i-l do
            call delete-a-mark (the place that is
               related-with sp,the color of sp);
        end;
    end;
    for o-l=each output-sublink connected at an
       output of st do
        sp= the subplace at the output end of o-l;
        for i=1 to the weight of o-l do
            call create-a-mark (the place that is
               related-with sp,the color of sp);
        end;
    end;
    call change-implied (st);
  end;
end
```

FIRE-EVENT1, a procedure

```
fire-event1(st:class subtransition)
    t:class transition = the transition that is
       related-with st;
begin
    conclude that st is not waiting-event-to t;
    call fire-subtransition (st);
    start sequence-stabilization ();
end
```

INITIALISATION, a procedure

```
initialisation(petri:class petri-net)
    t: class transition;
    p: class place;
    sp: class subplace;
    st: class subtransition;
    t-p-l: class transition-place-link;
    p-t-l: class place-transition-link;
    i:quantity;
    w:class kb-workspace=the subworkspace of
       petri;
begin
    activate the subworkspace of petri;
    for t= each transition upon w do
        start trans-tra (t);
        start create-subtransition (t);
        for t-p-l= each transition-place-link
```

```
            connected to t do
               start trans-out (t-p-l);
         end;
         for p-t-l = each place-transition-link
            connected to t do
             start trans-inp (p-t-l);
         end;
     end;
     for p= each place upon w do
         start trans-pla (p);
     end;
     start fin-init(petri);
  end
```

FIRE-EVENT, a procedure

```
fire-event(t: class transition,c:text)
    st:class subtransition;

begin
    st= call the-subtransition (t,c);
    conclude that st is not waiting-event-to t;
    call fire-subtransition (st);
    start sequence-stabilization ();
end
```

NEW-LIST-SUBTRANSI, a procedure

```
new-list-subtransi(st:class subtransition)
    t:class transition;

begin
    call calcul-subtransi (st);
    if the enable of st then
        begin
            if st is with-the-propriety non-event
               then conclude that st is
               subtransition-that is-fireable
            else
             begin
               t= the transition that is related-
                  with st;
               conclude that st is waiting-event-to
                  t;
            end;
        end;
end
```

SEQUENCE-STABILIZATION, a procedure

```
sequence-stabilization()
```

```
    st:class subtransition;
    val: truth-value;
begin
for st=each subtransition that is subtransition-that is-
   fireable do
    call fire-subtransition (st);
    conclude that st is not subtransition-that is-
       fireable ;
end;
for st=each subtransition that is subtransition-that
   must-be-recalculate do
    call new-list-subtransi (st);
    conclude that st is not subtransition-that must-be-
       recalculate ;
end;
if there exists a subtransition st that is
   subtransition-that is-fireable then start sequence-
   stabilization ();
conclude that feu is true;
end
```

THE-SUBPLACE, a procedure

```
the-subplace(p: class place, c:text) = (class
   subplace)
sp:class subplace;
begin
    if there exists a subplace sp that is in-
       relation-with p such that (the color of
       sp=c) then return sp;
end
```

THE-SUBTRANSITION, a procedure

```
the-subtransition(t: class transition, c:text) =
   (class subtransition)
st:class subtransition;
begin
    if there exists a subtransition st that is in-
       relation-with t such that (the color of
       st=c) then return st;
end
```

CHANGE-IMPLIED, a procedure

```
change-implied (st: class subtransition )
    sp:class subplace;
    st2: class subtransition;
begin
conclude that st is subtransition-that must-be-
   recalculate ;
```

```
for sp =each subplace connected at an output of st
   do
    for st2 = each subtransition connected at an
       output of sp do
       conclude that st2 is subtransition-that
          must-be-recalculate ;
        end;
  end;
end
```

CALCUL-SUBTRANSI, a procedure

```
calcul-subtransi(st:class subtransition)
    sp:class subplace;
    i-l:class input-sublink;
begin
 if there exists a input-sublink i-l connected at
    an input to st such that (the nb-element-
    available of the subplace at the input end of
    i-l < the weight of i-l) then
     begin
       conclude that the enable of  st is false;
     end
  else
      begin
        conclude that the enable of st is true;
      end;
end
```

CREATE-SUBTRANSITION, a procedure

```
create-subtransition(t:class transition)
    col:text;
    st:class subtransition;
    val:truth-value=false;
begin
    if there exists a subworkspace sw of t such that
       (there exists a rule nearest to sw) then val =
       true;
    for col=each text in the color-set of t do
        create a subtransition st;
        transfer st to espace-de-calcul;
        conclude that the color of st = col;
        conclude that st is in-relation-with t;
        if val then conclude that st is with-the-
           propriety event else conclude that st is
           with-the-propriety non-event;
     end;
end
```

FIND-SUBPLACE, a procedure

```
find-subplace(p:class place,color:text) = (class
   subplace )
    sp:class subplace;
begin
    if there exists a subplace sp that is in-
       relation-with p such that (the color of
       sp=color) then return sp
     else
        begin
             create a subplace sp;
             transfer sp to espace-de-calcul;
             conclude that sp is in-relation-with
                p;
             conclude that the color of sp =color;
             return sp;
         end;
end
```

CREATE-OUTLINK, a procedure

```
create-outlink(sp:class subplace,st: class
   subtransition,q:quantity)

    o-l: class output-sublink;
begin
    create a connection o-l of class output-
       sublink connected between sp newly locating
       it at bottom 25 and st newly locating it at
       top 25 with style diagonal, with direction
       input ;
    conclude that the weight of o-l =q;
end
```

CREATE-INPUTLINK, a procedure

```
create-inputlink(sp:class subplace,st:class
   subtransition,q:quantity)

    i-l: class input-sublink;

begin
    create a connection i-l of class input-sublink
       connected between sp newly locating it at
       bottom 25 and st newly locating it at top
       25 with style diagonal,with direction
       output;
    conclude that the weight of i-l=q;
end
```

TRANS-PLA, a procedure

```
trans-pla (p:class place)
    formule: text=the initial-marking of p;
    i: quantity;
    part:text;
    x: quantity;
    y: quantity;
begin
  repeat
    exit if  (formule="");
    if  is-contained-in-text("sum",formule)then
      begin
            x= position-of-text ("sum",formule);
            part= get-from-text (formule,x +
               4,length-of-text (formule));
            y= position-of-text (")",part);
            part= get-from-text (part, 1,y - 1);
           call trans-sum (p,part);
         formule= omit-from-text (formule,x,y + x
            + 3);
           go to sortie;
        end;
    x= position-of-text ("<",formule);
    part= get-from-text (formule, 1,x);
    y= quantity (part); formule= omit-from-text
       (formule,1,x);
    x= position-of-text (">",formule);
    part= get-from-text (formule, 1,x - 1);
    for i=1 to y do
        call create-a-mark (p,part);
       end;
    formule= omit-from-text (formule,1,x);
sortie: end;
end
```

TRANS-TRA, a procedure

```
trans-tra(t: class transition)
formule: text=the color-set-def of t;
    i: quantity;
    part:text;
    x: quantity;
    y: quantity;
begin
    repeat
        exit if (formule="");
        if  is-contained-in-
           text("sum",formule)then
      begin
            x= position-of-text ("sum",formule);
            part= get-from-text (formule,x +
                4,length-of-text (formule));
             y= position-of-text (")",part);
             part= get-from-text (part, 1,y - 1);
            call trans-sum2(t,part);
          formule= omit-from-text (formule,x,y + x
             + 3);
            go to sortie;
        end;
        x= position-of-text (">",formule);
        part= get-from-text (formule,2,x - 1);
        formule= omit-from-text (formule,1,x);
        insert part at the end of the text list
           the color-set of t;
sortie:   end;
end
```

TRANS-SUM2, a procedure

```
trans-sum2(t:class transition ,t1:text)
i: quantity;
part:text;
ele: text;
x: quantity;
y: quantity;
begin
    x= position-of-text (",", t1);
    part = get-from-text (t1,1,x - 1);
    y= quantity (omit-from-text (t1,1,x));
    for i=1 to y do
        ele= insert-in-text (part,".[i]",1);
        insert ele at the end of the text list the
           color-set of t;
        end;
end
```

TRANS-SUM, a procedure

```
trans-sum(p:class place,t1:text)
i: quantity;
part:text;
ele: text;
x: quantity;
y: quantity;
begin
    x= position-of-text (",", t1);
    part = get-from-text (t1,1,x - 1);
    y= quantity (omit-from-text (t1,1,x));
    for i=1 to y do
        ele= insert-in-text (part,".[i]",1);
        call create-a-mark (p,ele);
        end;
```

```
end
```

TRANS-OUT, a procedure

```
trans-out (t-p-l: class transition-place-link)
    form-out: text = the output-function-text of
       t-p-l;
    p:class place=the place at an output end of t-
       p-l;
    t:class transition =the transition at the
       input end of t-p-l;
    sp:class subplace;
    st: class subtransition;
    formule,part,color:text;
    x,y: quantity;
begin
if not(is-contained-in-text ("<",form-out)) then
  begin
    for st =each subtransition that is in-
       relation-with the transition connected to
       t-p-l do
        color=call value-function(form-out,the
           color of st);
        sp=call find-subplace (p,color);
        call create-outlink(sp,st,1);
        end;
   end
 else
 begin
  repeat
    exit if (form-out="");
    x = position-of-text (";",form-out);
    formule= get-from-text (form-out, 1,x - 1);
    form-out= omit-from-text (form-out,1,x);
    x = position-of-text (":",formule) ;
    part= get-from-text (formule, 1,x - 1);
    st=call the-subtransition(t,part);
    formule= omit-from-text (formule,1,x);
    repeat
      exit if (formule="");
      x= position-of-text ("<",formule);
      part= get-from-text (formule, 1,x);
      y= quantity (part);
      formule= omit-from-text (formule,1,x);
      x= position-of-text (">",formule);
      color= get-from-text (formule, 1,x - 1);
      sp=call find-subplace (p,color);
      call create-outlink(sp,st,y);
      formule= omit-from-text (formule,1,x);
    end;
    end;
  end;
end
```

TRANS-INP, a procedure

```
trans-inp (p-t-l: class place-transition-link)
    t:class transition=the transition at the
       output end of p-t-l;
    p:class place=the place at the input end of p-
       t-l;
    st:class subtransition;
    sp:class subplace;
    formule,part,color: text;
    form-inp: text = the inp-fonction-text of p-t-
       l;
    x,y: quantity;
begin
if not(is-contained-in-text ("<",form-inp)) then
  begin
    for st=each subtransition that is in-relation-
       with the transition connected to p-t-l do
        color=call value-function(form-inp,the
           color of st);
        sp=call find-subplace (p,color);
        call create-inputlink(sp,st,1);
        end;
   end
else
 begin
  repeat
    exit if (form-inp="");
    x = position-of-text (";",form-inp);
    formule= get-from-text (form-inp, 1,x - 1);
    form-inp= omit-from-text (form-inp,1,x);
    x = position-of-text (":",formule) ;
    part= get-from-text (formule, 1,x - 1);
    st=call the-subtransition(t,part);
    formule= omit-from-text (formule,1,x);
    repeat
      exit if (formule="");
      x= position-of-text ("<",formule);
      part= get-from-text (formule, 1,x);
      y= quantity (part);
      formule= omit-from-text (formule,1,x);
      x= position-of-text (">",formule);
      color= get-from-text (formule, 1,x - 1);
      sp=call find-subplace (p,color);
      call create-inputlink(sp,st,y);
       formule= omit-from-text (formule,1,x);
    end;
    end;
```

```
end;
end
```

VALUE-FUNCTION, a procedure

```
value-function(fu: text,ele: text) = (text)
    def-fu: text;
    nb-fu: quantity;
    fu-inf:text;
    fu-inf2:text;
    pos1:quantity;
    pos2:quantity;
    pos3:quantity;
    result:text;
    result1:text;
    result2:text;
begin
if is-contained-in-text ("add",fu) then
    begin
        result=get-from-text(fu,5, length-of-text
           (fu) - 1);
        return add(ele,result);
    end;
if is-contained-in-text ("inv",fu) then
    begin
        result=get-from-text(fu,5, length-of-text
           (fu) - 1);
        return result;
    end;
case(fu) of
    "id": return ele;
    "dec": return "dec";
otherwise :
begin
nb-fu= quantity (omit-from-text (fu,1,4) );
def-fu = get-from-text (fu,1,4);
case (def-fu) of
    "succ": return  succ(get-color (nb-fu,ele));
    "prec": return  prec(get-color (nb-fu, ele));
    "colo": return get-color (nb-fu, ele);
    "proj": return get-color (nb-fu, ele);
    "comb":
        begin
            if nb-fu=1 then
              begin
                fu-inf= get-from-text (fu,7,
                   length-of-text (fu) - 1);
                result= call value-function (fu-
                   inf,ele);
              end
            else
```

```
                begin
                  pos1= position-of-text ("(",fu);
                  pos2= position-of-text (",",fu);
                  pos3= length-of-text (fu);
                  fu-inf=concate("comb","[nb-fu -
                     1](");
                  fu-inf=concate(fu-inf, get-from-
                     text (fu,pos2 + 1,pos3));
                  fu-inf2= get-from-text (fu,pos1 +
                     1,pos2 - 1);
                  result1= call value-function (fu-
                     inf2,ele);
                  result2= call value-function (fu-
                     inf,ele);
                  result=concate(result1,"-");
                  result= concate (result,result2);
                end;
              return result;
          end;
      end;
end;
end;
end
```