Dipartimento di Informatica
Università del Piemonte Orientale "A. Avogadro"
Spalto Marengo 33, 15100 Alessandria
http://www.di.unipmn.it

FIRB Perf Project supported by MIUR under Grant FIRB-Perf-RBNE019N8N

Performance Evaluation of Complex Systems: Techniques, Methodologies and Tools.

TECHNICAL REPORT TR-INF-2003-10-06-UNIPMN
(October 2003)

Development of a Dynamic Fault Tree Solver based on Coloured Petri Nets and graphically interfaced with *DrawNET*

Author: D. Codetta Raiteri (raiteri@unipmn.it)

Contents

Chapter 1

Introduction

This report is about the realization of a tool to solve *Dynamic Parametric Fault Trees* (DPFT) [1] [2], an extension of traditional Fault Trees (FT) [3] that includes S-dependencies; traditional FT's have gained a widespread acceptance for the dependability and safety analysis of complex systems since they are simple and easy to manipulate, but they have the limitation that basic components are assumed to be s-indipendent. S-dependence in the failure process arises when the failure behaviour of a component depends on the state of the system.

The approach to solve DFT's presented in this report is based on two peculiar features; first, we adopt a parameterization technique, referred to as *Parametric Fault Tree* (PFT) [4] [5] [6], to fold equal subtrees or components and resort them to a more compact representation; parameterization can be applied to DFT too, generating the *Dynamic Parametric Fault Tree* (DPFT) model. Second, DPFT can be modularized and each module [7] [8] translated into a *Coloured Petri net* in the form of *Stochastic Well-formed Net* (SWN) [9].

The analysis of DFT's may require the generation of a very large state space; parameterization, modularization and the conversion to SWN allow a remarkable reduction of the state space dimensions when redundancies and symmetries are present in the system.

The tool is composed by a graphic interface (*DrawNET* [10] [11]), a post processor and a translator from DPFT to SWN; *DrawNET* has been adapted to draw a DPFT, to execute its solver and to collect and visualize the results. The user has only to draw the DPFT and request the results.

1.1 Dynamic Fault Trees

DFT's allow to model systems where there are dependencies between components or where the failure of a subsystem may be caused by something more complicated than a simple combination of failure events as in the traditional FT. In order to obtain all of that, some new gates called *dynamic gates* [1] [2] were introduced and they are:

- *Priority And* (PAND)
- *Functional Dependency Gate* (FDEP)
- *Sequence Enforcing Gate* (SEQ)
- *Warm Spare Gate* (WSP)

1.1.1 Dynamic gates description

Priority And

PAND gate is specified to fail if all of its input events have occured and in a specific order (from left to right); PAND may have two or more input events and is similar to AND gate, but it has one more constraint: the order of the events; if the order is not respected the gate does not fail. PAND gate inputs may be basic events or internal events.

In Fig. 1.1, A, B and C are the input events of a PAND gate that fails when A, B, C have failed in this order.

Functional Dependency Gate

FDEP gate causes simultaneous failures; its input events are a trigger event and a set of dependent components; when the trigger event occurs, the dependent components are forced to fail if they did not before by their own; in this way FDEP models the functional dependency of a set of components on another one. The trigger event is tipically a basic event; the number of dependent events may be one or more and usually they are basic events too.

In Fig. 1.2 we have a FDEP gate whose trigger event is T (connected to the left side of the gate) while its dependent events are A and B; when T fails, A and B are forced to fail.

Sequence Enforcing Gate

SEQ forces its input events to occur in a specified order (from left to right); every component can not fail before its predecessor has not failed yet; this

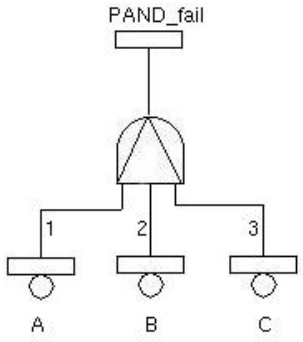

Figure 1.1: PAND gate

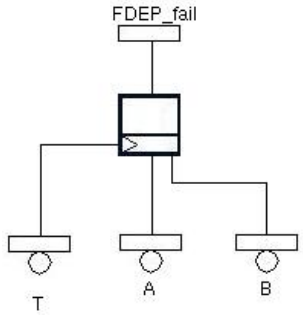

Figure 1.2: FDEP gate

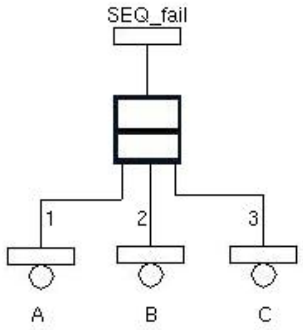

Figure 1.3: SEQ gate

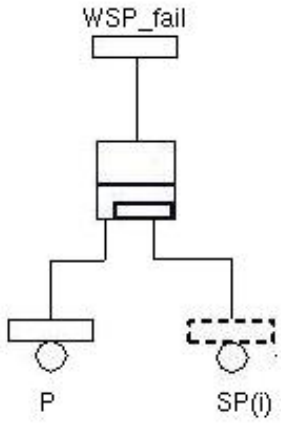

Figure 1.4: WSP gate

gate fails when all of its input events have occured and they can only fail in the specified order. SEQ input events are basic events.

In Fig. 1.3, A, B and C are forced to occur in this order; B can not fail until A has failed and C can not fail until B has failed.

Warm Spare Gate

WSP models the presence of a set of spare components associated with a main component; when the main component fails it can be replaced by one of its spares; a failure rate λ and a dormancy factor α are associated to each spare component that can be in one of these states:

- dormant (stand by),
- working,
- failed.

Every spare component is dormant initially; in this state, it may fail by the failure rate $\alpha\lambda$ ($0 \leq \alpha \leq 1$) smaller than λ in order to reduce the probability of failure of the spare during its dormancy state. When the main component fails, one of the available (not failed and not working) spares changes its state from dormant to working in order to replace the main component; during the working state the spare may fail with the failure rate λ.

This gate fails when the main component and all of its spares are failed; WSP becomes a *Hot Spare Gate* (HSP) when $\alpha = 1$; in this case, the failure rate of the spare is the same during its dormant and working states. Another version of this gate is the *Cold Spare Gate* (CSP) when $\alpha = 0$, so the failure rate during the dormant state is equal to 0 and the spare can only fail during its working state.

WSP can be connected only to basic events; one of them must represent the main component and the other ones must represent the spares; spare components may be connected to more than one WSP gate; in this case, they are shared among several main components.

1.1.2 Parametric Dynamic Fault Trees

PFT [4] [5] [6] formalism can be extended to DFT's in order to give a more compact representation of identical subtrees by connecting (basic) replicator events to both static and dynamic gates and assigning them parameters. When a gate is connected to a set of basic events with the same failure rate,

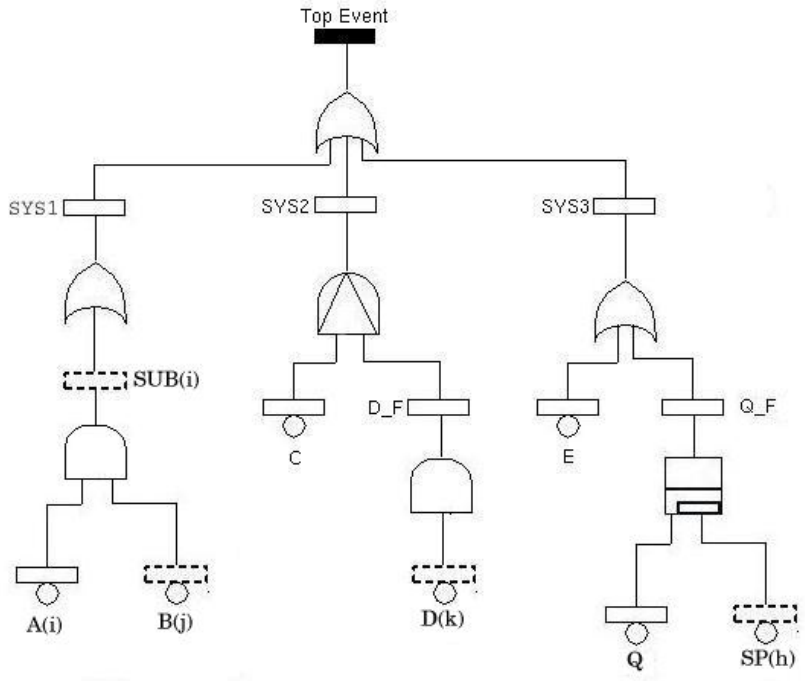

Figure 1.5: Example of DPFT

they can be folded to a unique basic replicator with the same failure rate and a parameter that can assume as many values as the number of folded basic events; if a gate is connected to a set of subtrees with the same structure and failure rates, the identical subtrees can be folded in a unique subtree with the same structure and failure rates and whose root is a replicated event with a parameter that can assume as many values as the number of folded subtrees; this parameter must appear in the events of the new subtree, except in the case that some (basic) events were shared between the identical subtrees.

For instance, in Fig. 1.4 there is the parametric representation of the main component P whose set of m spares is represented by the basic replicator event SP(i) whose parameter i can assume values between 1 and m. When P fails it is replaced by one of the available SP(i); when, after P failure, every SP(i) has failed, the gate fails.

Fig. 1.5 is an example of DPFT: this system is composed by three subsystems called SYS1, SYS2 and SYS3; SYS1 fails if at least one among SUB1, SUB2 and SUB3 (represented as SUB(i)) fails; SUB(i) fails when all of its component (A(i) and B(j)) fail. B(j) is shared among SUB1, SUB2 and SUB3.

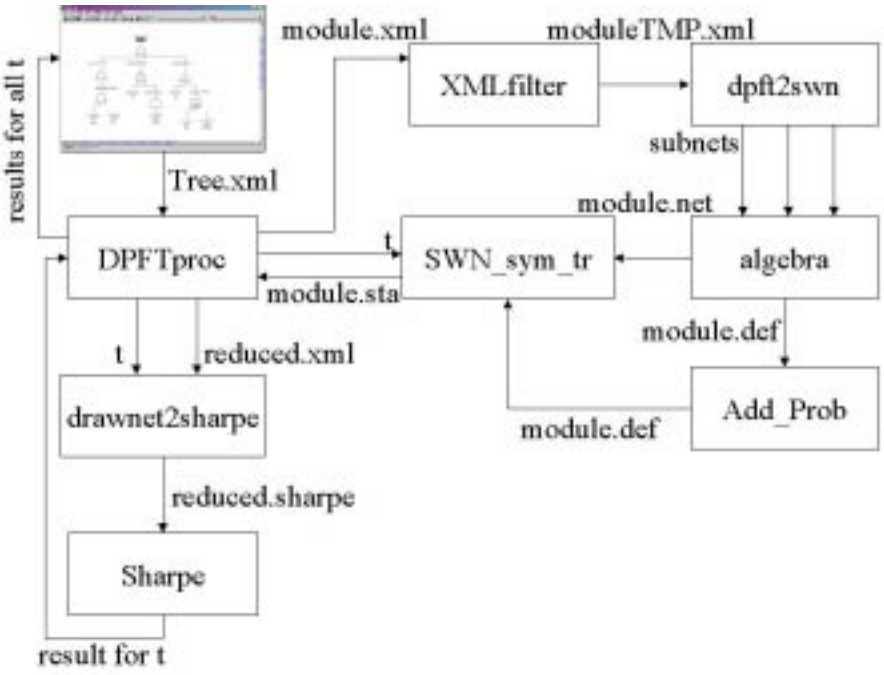

Figure 1.6: Tool overview

SYS2 fails if both C and D_F fail and C does before D_F; D_F fails when every D(k) has failed; SYS3 fails if E or Q_F fails; Q_F fails when Q is failed and there are no spares SP(h) available to replace it.

1.2 Tool overview

The general scheme of the tool is shown in Fig. 1.6; *DrawNET* [10] [11] graphic tool allows us to draw the DPFT, define its features and indicate what type of analysis to be executed on the DPFT (transient analysis). Clicking on **Transient** from the **Execute** menu, *DrawNET* generates the XML representation of the DPFT, saves it into a file and starts the execution of the DPFT processor (*DPFTproc*).

DPFTproc loads the XML representation of the DPFT generated by *DrawNET* and parses it filling its data structures with the information about the DPFT and the time values for the transient analysis; at this point *DPFTproc* detects the modules [7] [8] (indipendent subtrees) of the DPFT and their nature (static or dynamic, minimal or not). For each *Minimal Dynamic Module* (MDM) (a module containing at least one dynamic gate and

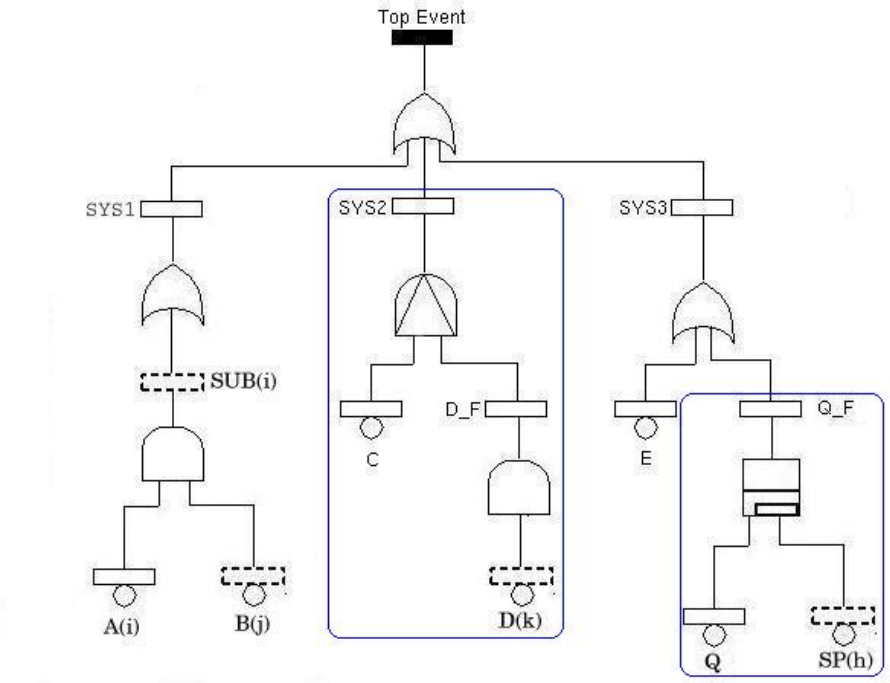

Figure 1.7: DPFT modules

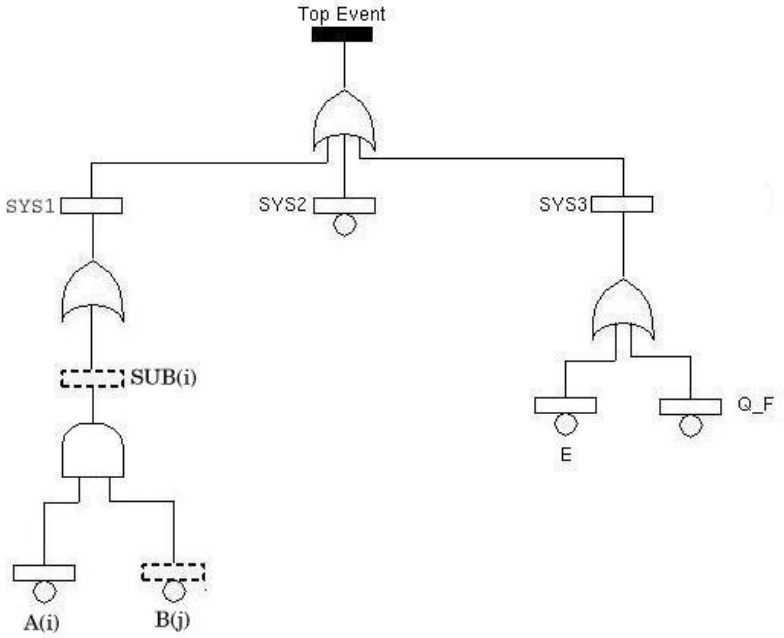

Figure 1.8: Reduced PFT

not containing other modules) *DPFTproc* generates its XML representation according to the same formalism adopted by *DrawNET*. Fig. 1.7 shows the MDM's of the DPFT in Fig. 1.5

Each module XML representation is passed to *XMLfilter* that modifies the XML code according to the formalism adopted by *dpft2swn* (translator from DPFT to SWN), but keeping unaltered the module structure; the new XML representation of each module is saved by *XMLfilter* and loaded by *dpft2swn* that translates the module in its equivalent SWN [9] and saves it in the *GreatSPN* [12] format.

Now, for each requested time value, a transient analysis of each SWN is performed by a specific solver (previously realized for the *GreatSPN* package) asking for the probability of failure of the module at that time; the results are collected by *DPFTproc* that for each time value replaces in the DPFT each module with a basic event whose failure probability is equal to failure probability of the module at that time; then, *DPFTproc* generates the XML representation of the reduced PFT (now it is static).

This XML file is taken in input by *drawnet2sharpe* [13] (previously realized) that translates a PFT expressed in the *DrawNET* formalism in the

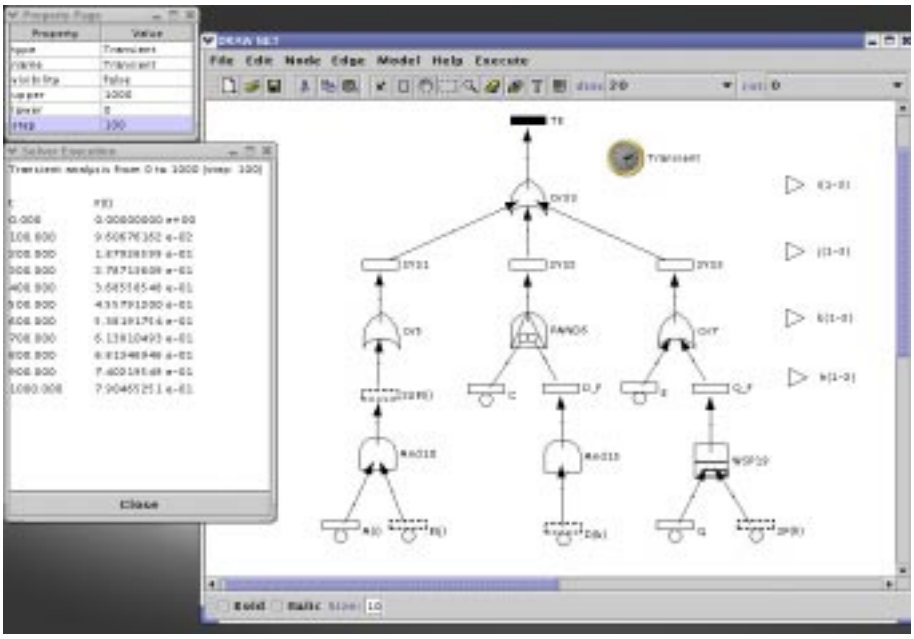

Figure 1.9: *DrawNET* screenshot

Sharpe [14] formalism after having unfolded it; *Sharpe* is a tool for the analysis of several stochastic models and is able to deal with static FT's; *Sharpe* is asked to perform the transient analysis of the reduced PFT (Fig. 1.8) at the current time. The results calculated by *Sharpe* on the reduced PFT for each time value are collected by *DPFTproc* that saves them in a specific file called **tr_results**; now *DPFTproc* ends and *DrawNET* visualizes in a window what **tr_results** contains.

Fig. 1.9 shows how the DPFT in Fig. 1.5 appears in *DrawNET* with the visualization of the results of the transient analysis on it.

1.2.1 Requirements

Since our tool is composed by several parts, also several directories of files are necessary to the tool to work; they are shown in Fig. 1.10. **DPFT2SWN** contains:

- *DPFTproc*, *XMLfilter*, *dpft2swn*, *Add_Prob* and *drawnet2sharpe* executables;
- *DPFTproc*, *XMLfilter*, *dpft2swn*, *Add_Prob* and *drawnet2sharpe* source code;

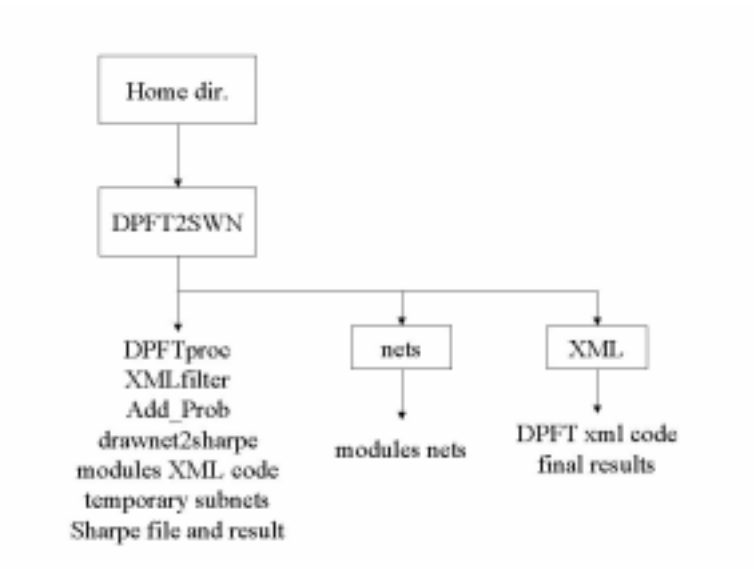

Figure 1.10: Tool directories

- temporary files containing the XML representations of the modules and of the reduced PFT;
- temporary files containing the subnets generated for each gate of a module by *dpft2swn* and later connected;
- the *Sharpe* version of the reduced PFT.

XML is a link to the directory where *DrawNET* is installed and where XML representation of the drawn DPFT is saved by *DrawNET*; **nets** is a link to the directory containing *GreatSPN* nets; here, there are the SWN for each module.

To run the tool some other tools and utilities need to be installed on the system:

- *GreatSPN* package (to solve SWN)
- *algebra* (to compose the SWN's representing modules)
- *Sharpe* tool (to solve static fault trees)
- *drawnet2sharpe* to unfold and translate a PFT in its *Sharpe* version.

Chapter 2

DrawNET as Graphical Interface

DrawNET has been used as graphical interface for our tool to draw the DPFT, to request the results (transient analysis) and to visualize them. *DrawNET* has been adapted to support all of that and more specifically it has been modified in order to:

- load the DPFT formalism when *DrawNET* is run;
- execute the DPFT solver by clicking on **Transient** from the **Execute** menu;
- visualize the results when the analysis has finished.

DrawNET has been developed in Java language and is executed by running the **DPFTedit** class, included in the directory where *DrawNET* is installed; the code of this class (DPFTedit.java) follows:

```
public class DPFTedit
{  public static void main(String[]args)
        {
          SolverDefinition[] SD = {
                new SolverDefinition("Transient", "dpft_tr")};
          JFrame f = new PrincipalFrame(new ToolDefinition(
                new ModelToXML("xml"), "DPFT", SD));
          f.show();
        }
}
```

Using this class, *DrawNET* loads the DPFT formalism contained in the file named DPFT.xml and associate **dpft_tr** command to the click on **Transient** in the **Execute** menu; **dpft_tr** is a script that changes the working directory from the current (where *DrawNET* is installed) to the **DPFT2SWN**

directory and runs *DPFTproc*; when *DPFTproc* ends, the script visualizes the results by means of the command **cat**. The output of this script (the transient analysis results) is shown by *DrawNET* in a window that appears when the results become available.

2.1 DPFT formalism

This is what DPFT.xml contains:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE formalism SYSTEM "formalism.dtd">
<formalism parent="" name="PFT">

  <propertyType name="Title" default=""/>

  <nodeType parent="" name="Event0">
    <propertyType name="Label" default=""/>
    <propertyType name="Parameters" default=""/>
  </nodeType>

  <nodeType parent="Event0" name="Event"/>

  <nodeType parent="Event0" name="BasicEvent">
    <propertyType name="Distribution" default="ALL EXP 0.001"/>
  </nodeType>

  <nodeType parent="Event" name="ReplicatorEvent">
    <propertyType name="DeclaredParameters" default=""/>
  </nodeType>

  <nodeType parent="BasicEvent" name="BasicReplicatorEvent">
    <propertyType name="DeclaredParameters" default=""/>
    <propertyType name="alfa" default="1"/>
  </nodeType>

  <nodeType parent="" name="TopEvent">
    <propertyType name="Label" default=""/>
  </nodeType>

  <nodeType parent="" name="Gate"/>
```

```
<nodeType parent="Gate" name="And"/>

<nodeType parent="Gate" name="Or"/>

<nodeType parent="" name="G2of3"/>

<nodeType parent="Gate" name="KofN">
  <propertyType name="K" default="2"/>
  <propertyType name="N" default="3"/>
</nodeType>

<nodeType parent="" name="PAND"/>

<nodeType parent="" name="SEQ"/>

<nodeType parent="Gate" name="FDEP"/>

<nodeType parent="" name="WSP"/>

<nodeType parent="" name="Parameter">
  <propertyType name="ParameterName" default=""/>
  <propertyType name="Cardinality" default="1"/>
</nodeType>

<nodeType parent="" name="Transient">
  <propertyType name="lower" default="0"/>
  <propertyType name="upper" default="1000"/>
  <propertyType name="step" default="100"/>
</nodeType>

<edgeType parent="" name="Arc">
  <constraint fromType="Gate" fromCardinality="1"
   toType="Event" toCardinality="1"/>
  <constraint fromType="WSP" fromCardinality="1"
   toType="Event" toCardinality="1"/>
  <constraint fromType="G2of3" fromCardinality="1"
   toType="Event" toCardinality="1"/>
  <constraint fromType="PAND" fromCardinality="1"
   toType="Event" toCardinality="1"/>
  <constraint fromType="SEQ" fromCardinality="1"
   toType="Event" toCardinality="1"/>
```

```
    <constraint fromType="Gate" fromCardinality="1"
     toType="TopEvent" toCardinality="1"/>
    <constraint fromType="WSP" fromCardinality="1"
     toType="TopEvent" toCardinality="1"/>
    <constraint fromType="G2of3" fromCardinality="1"
     toType="TopEvent" toCardinality="1"/>
    <constraint fromType="PAND" fromCardinality="1"
     toType="TopEvent" toCardinality="1"/>
    <constraint fromType="SEQ" fromCardinality="1"
     toType="TopEvent" toCardinality="1"/>
    <constraint fromType="Event0" fromCardinality=""
     toType="Gate" toCardinality=""/>
    <constraint fromType="Event0" fromCardinality=""
     toType="G2of3" toCardinality="3"/>
    <constraint fromType="BasicEvent" fromCardinality=""
     toType="WSP" toCardinality="2"/>
  </edgeType>

  <edgeType parent="" name="Order">
    <propertyType name="No." default="0"/>
    <constraint fromType="BasicEvent" fromCardinality=""
     toType="PAND" toCardinality=""/>
    <constraint fromType="Event" fromCardinality=""
     toType="PAND" toCardinality=""/>
    <constraint fromType="BasicEvent" fromCardinality=""
     toType="SEQ" toCardinality=""/>
  </edgeType>

  <edgeType parent="" name="Trigger">
    <constraint fromType="Event" fromCardinality=""
     toType="FDEP" toCardinality="1"/>
    <constraint fromType="BasicEvent" fromCardinality=""
     toType="FDEP" toCardinality="1"/>
  </edgeType>

</formalism>
```

DPFT.xml is an extension of PFT.xml [15] (previously realized) to include the definition of dynamic gates and some special types of arc:

- **Order**: it allows to connect the input events to a PAND gate or a SEQ gate and to assign them an order number (order of failure);

- **Trigger**: it connects a trigger event to a FDEP gate in order to distinguish the trigger event from the dependent events that are connected by an ordinary arc; a FDEP gate can have only one trigger event connected to; a trigger event may be basic or not.

In every other case an ordinary arc is used; the WSP gate must have two input events: a basic event to represent the failure of the main component and a basic replicator event to represent the set of spares.

Respect to the previous PFT formalism (PFT.xml) [15] a new attribute has been defined for the basic replicator event: **alpha**; **alpha** is the dormancy factor of the spare components and must be assigned by the user when a basic replicator event is connected to a WSP gate; if we assign to **alpha** 0, the WSP gate is equal to a *Cold Spare Gate* (CSP) gate; if we assign to **alpha** 1, the WSP gate is equal to a *Hot Spare Gate* (HSP) gate.

This formalism contains also the definition of the transient analysis time values: lower, upper, step.

Colour classes and subclasses do not appear in the formalism and must not be associated to the parameters any more; only parameters must be included in the DPFT drawing, while the corresponding colour classes and subclasses will be generated automatically translating the DPFT modules in SWN.

2.2 *DrawNET* windows

The version of *DrawNET* we use (*DrawNET++*) is composed by two windows: the main window and the *Property Page*; the main window contains the menus and allows us to draw the DPFT and execute the transient analysis; the **Node** menu (Fig. 2.1) contains all of the types of node a DPFT may have, while the **Edge** menu (Fig. 2.2) contains every type of arc. Fig. 2.3 shows all of the available types of events while Fig. 2.4 shows all of the gates (static and dynamic).

Property Page changes its aspect selecting an element of the DPFT (a node or a arc) and shows the attributes relative to the selected element:

- some attributes must be specified for the events as shown from Fig. 2.5 to Fig. 2.8;
- to every parameter (Fig. 2.9) a cardinality must be assigned to specify the number of values the parameter can assume;
- Fig. 2.10 shows the *Property Page* for an edge of type **Order** with the indication of the order number (in this case, 2);

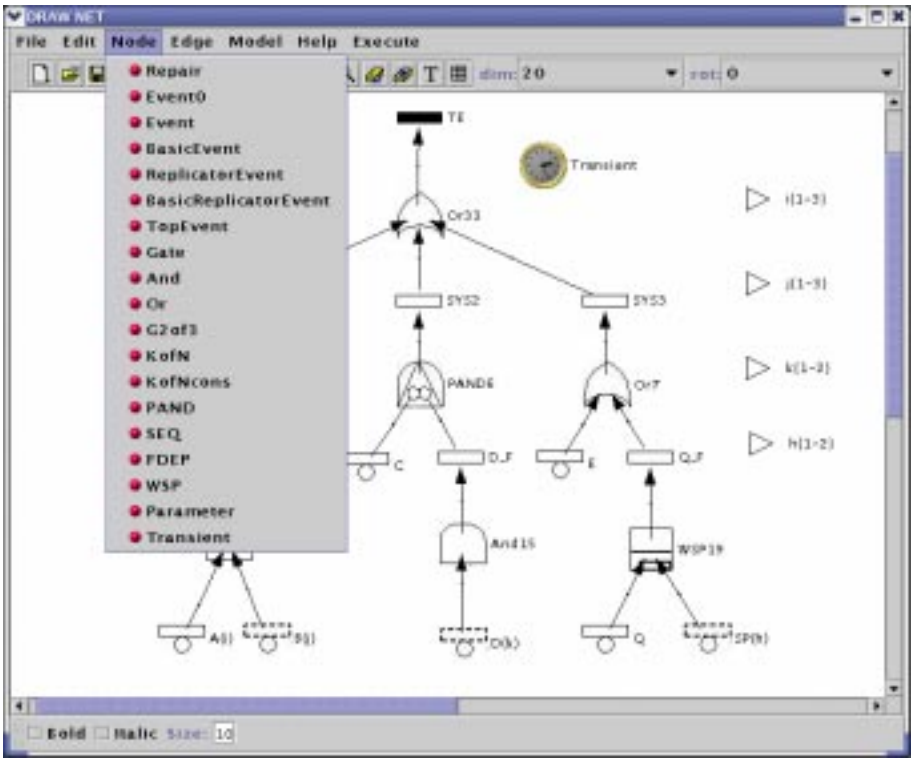

Figure 2.1: All of the types of node

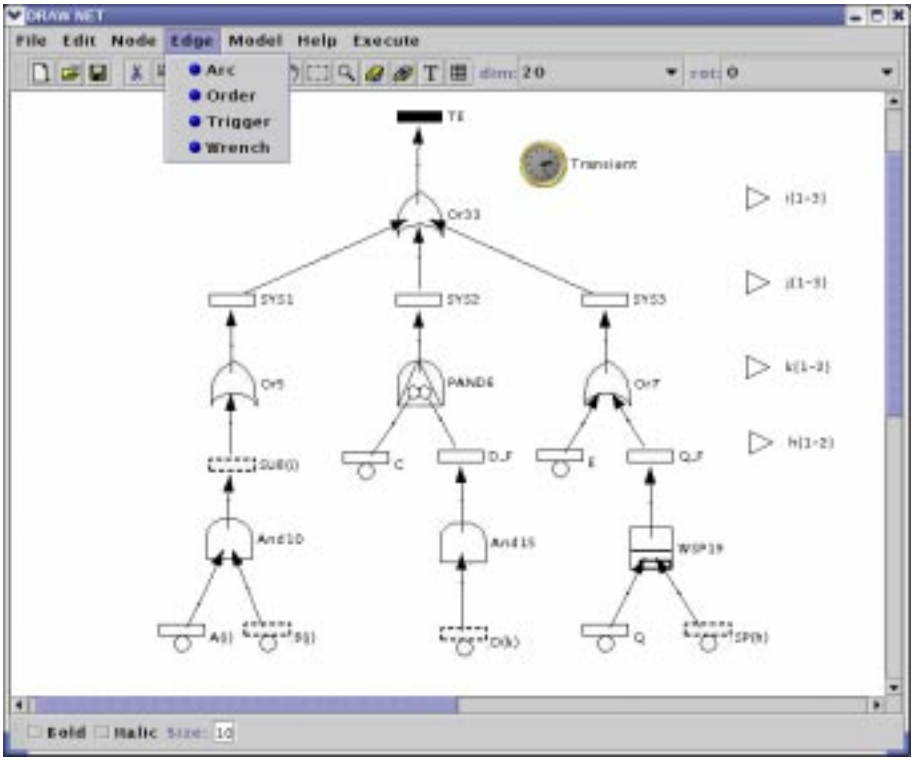

Figure 2.2: All of the types of edge

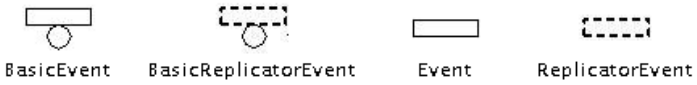

Figure 2.3: All of the types of events

Figure 2.4: All of the types of gate

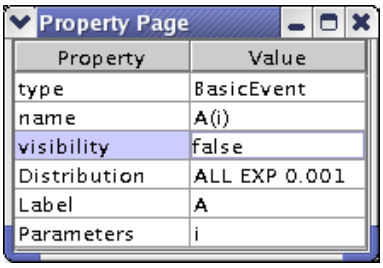

Figure 2.5: *Property Page* for A(i) basic event

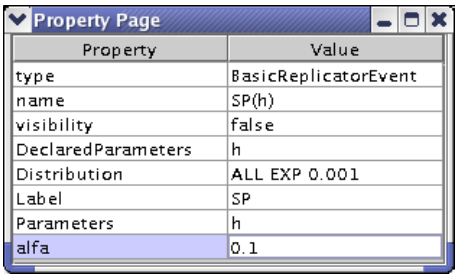

Figure 2.6: *Property Page* for SP(h) basic replicator event

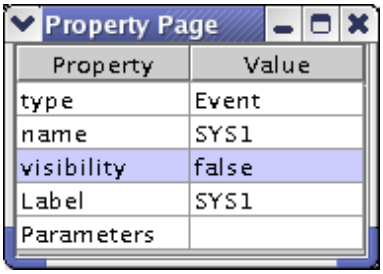

Figure 2.7: *Property Page* for SYS1 event

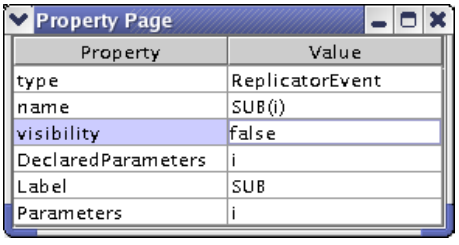

Figure 2.8: *Property Page* for SUB(i) replicator event

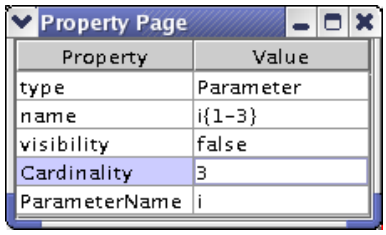

Figure 2.9: *Property Page* for a parameter

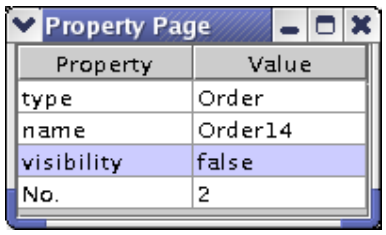

Figure 2.10: *Property Page* for and Order arc

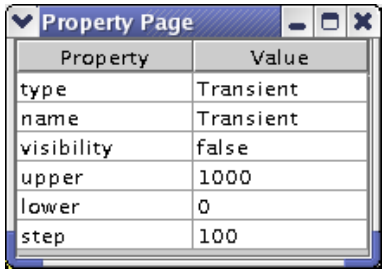

Figure 2.11: *Property Page* for transient analysis time values

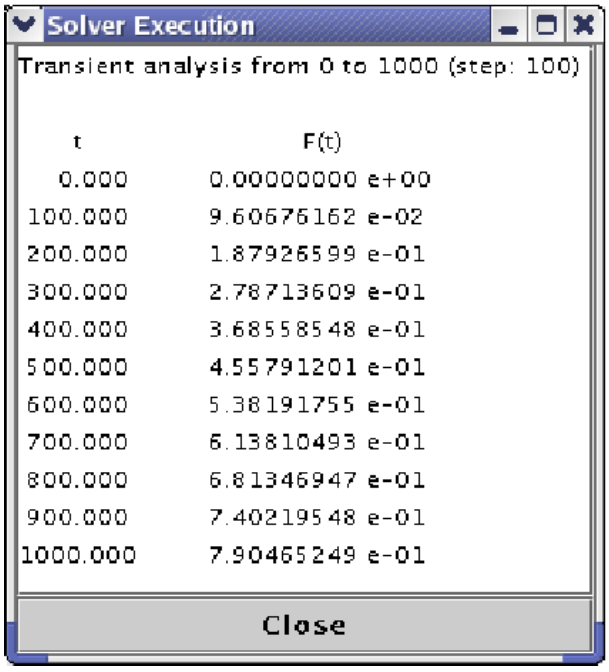

Figure 2.12: Results of the transient analysis

- in order to set the time values for the transient analysis the user has to insert the transient icon (a clock) from the **Node** menu and the *Property Page* will appear as in Fig. 2.11; here the user can modify the time values setting the lower and the upper values and the step.

Clicking on **Transient** from the **Execute** menu, the transient analysis starts and when it ends a window with the results will appear as in Fig. 2.12.

Chapter 3

The DPFT processor

DPFTproc is the core of our tool:

- it detects the modules of the DPFT and their nature;
- for each MDM, *DPFTproc* generates its XML representation and runs the translators and the solver;
- it collects the results and generates the XML representation of the reduced PFT;
- *DPFTproc* runs *Sharpe* on the reduced PFT and collects the final results for each time value.

3.1 *DPFTproc* parser

DPFTproc parser, for each line of the XML representation of the DPFT generated by *DrawNET*, examines the initial tag identifying what that line describes; for every possible tag the parser executes a specific procedure that loads the attributes of the element (a node or a arc) in the corresponding data structure.

The XML code generated by *DrawNET* for the DPFT in Fig. 1.5 follows:

```
<?xml  version='1.0' encoding='UTF-8' standalone='no'?>
<PFT name='report' visibility='false' Title='technical_report'>
  <TopEvent name='TE' visibility='false' Label=''/>
  <Event name='SYS2' visibility='false' Label='SYS2' Parameters=''/>
  <Event name='SYS1' visibility='false' Label='SYS1' Parameters=''/>
  <Event name='SYS3' visibility='false' Label='SYS3' Parameters=''/>
  <Or name='Or5' visibility='false'/>
```

```
<PAND name='PAND6' visibility='false'/>
<Or name='Or7' visibility='false'/>
<Arc name='Arc4' visibility='false' from='Or5' to='SYS1'/>
<Arc name='Arc5' visibility='false' from='PAND6' to='SYS2'/>
<Arc name='Arc6' visibility='false' from='Or7' to='SYS3'/>
<ReplicatorEvent name='SUB(i)' visibility='false'
 DeclaredParameters='i' Label='SUB' Parameters='i'/>
<And name='And10' visibility='false'/>
<BasicEvent name='A(i)' visibility='false' Distribution='ALL EXP 0.001'
 Label='A' Parameters='i'/>
<BasicEvent name='C' visibility='false' Distribution='ALL EXP 0.001'
 Label='C' Parameters=''/>
<Event name='D_F' visibility='false' Label='D_F' Parameters=''/>
<And name='And15' visibility='false'/>
<BasicReplicatorEvent name='D(k)' visibility='false'
 DeclaredParameters='k' Distribution='ALL EXP 0.001'
 Label='D' Parameters='k' alfa='1'/>
<Arc name='Arc7' visibility='false' from='SUB(i)' to='Or5'/>
<Arc name='Arc8' visibility='false' from='And10' to='SUB(i)'/>
<Arc name='Arc10' visibility='false' from='A(i)' to='And10'/>
<Arc name='Arc11' visibility='false' from='And15' to='D_F'/>
<Arc name='Arc12' visibility='false' from='D(k)' to='And15'/>
<Order name='Order13' visibility='false' from='C' to='PAND6' No.='1'/>
<Order name='Order14' visibility='false' from='D_F' to='PAND6' No.='2'/>
<BasicEvent name='E' visibility='false' Distribution='ALL EXP 0.001'
 Label='E' Parameters=''/>
<Event name='Q_F' visibility='false' Label='Q_F' Parameters=''/>
<WSP name='WSP19' visibility='false'/>
<BasicEvent name='Q' visibility='false' Distribution='ALL EXP 0.001'
 Label='Q' Parameters=''/>
<BasicReplicatorEvent name='SP(h)' visibility='false'
 DeclaredParameters='h' Distribution='ALL EXP 0.001'
 Label='SP' Parameters='h' alfa='0.1'/>
<Arc name='Arc15' visibility='false' from='E' to='Or7'/>
<Arc name='Arc16' visibility='false' from='Q_F' to='Or7'/>
<Arc name='Arc17' visibility='false' from='WSP19' to='Q_F'/>
<Arc name='Arc18' visibility='false' from='Q' to='WSP19'/>
<Arc name='Arc19' visibility='false' from='SP(h)' to='WSP19'/>
<Parameter name='i{1-3}' visibility='false'
 Cardinality='3' ParameterName='i'/>
<Parameter name='j{1-3}' visibility='false'
```

```
   Cardinality='3' ParameterName='j'/>
  <Parameter name='k{1-2}' visibility='false'
   Cardinality='2' ParameterName='k'/>
  <Transient name='Transient' visibility='false'
   upper='1000' lower='0' step='100'/>
  <Or name='Or33' visibility='false'/>
  <Arc name='Arc24' visibility='false' from='Or33' to='TE'/>
  <Arc name='Arc25' visibility='false' from='SYS1' to='Or33'/>
  <Arc name='Arc26' visibility='false' from='SYS2' to='Or33'/>
  <Arc name='Arc27' visibility='false' from='SYS3' to='Or33'/>
  <BasicReplicatorEvent name='B(j)' visibility='false'
   DeclaredParameters='j' Distribution='ALL EXP 0.001'
   Label='B' Parameters='j' alfa='1'/>
  <Arc name='Arc28' visibility='false' from='B(j)' to='And10'/>
  <Parameter name='h{1-2}' visibility='false'
   Cardinality='2' ParameterName='h'/>
</PFT>
```

The declarations in the XML code are in the same order the elements have been drawn by the user; after having loaded every attribute in the lines of the XML code, the parser examines every arc and connects the data structure of every gate with the data structures of its input and output events by means of pointers.

First, the trigger arcs are examined, connecting the trigger event of a FDEP gate as the first input event; then the ordinary arcs are considered and finally the order arcs; if the user forgot to assign the order numbers to the order arcs, the input events of a PAND or SEQ gate will be connected in the order the arcs were drawn.

In the XML code there are also the time values (lower, upper, step) specified by the user for the transient analysis on the line identified by the <**Transient**> tag; if the user forgot to specify the time values, *DPFTproc* initialize them as 0, 1000, 100.

3.2 Modules detection

A module is an indipendent subtree; we have a module when a subtree does not share any node with other subtrees and it does not descend from a dynamic gate (dynamic gates establish some kind of dependance on their descendants).

To verify the first condition we use a linear algorithm [7]; this algorithm was thought to be executed on traditional FT's (static and not parametric)

and it does not consider the fact that some nodes may be shared by the effect of the parameterization. For instance, in the DPFT in Fig. 1.5, SUB(i) seems to share no nodes, but if we unfold it we can see that B(j) is shared between SUB1, SUB2 and SUB3; so SUB(i) is not a module.

This algorithm does not even consider the dependencies given by the dynamic gates; in Fig. 1.5, D_F is not a module because it descends from a dynamic gate; a module may be a dynamic or static module if contains or not at least one dynamic gate; a module of any nature is minimal if does not contain any other module; for example SYS3 is a dynamic module but it is not minimal because contains Q_F that is a module too.

A MDM is a module that contains some dynamic gates and does not not contain any other module.

Modules detection starts using a linear time algorithm [7] adapted to check also if an internal event descends or not from a dynamic gate [8] [16]; then for every module detected in this way a further control is done on its parameters to verify if it is really a module: for each parameter associated to the root of the candidate module we verify if it appears in every node of the module; if it does, the candidate module is really a module, else there is some shared nodes and it is not a real module.

At this point for each real module detected we perform another traversal of it counting the number of dynamic gates and the number of modules it contains; if it contains no module it is minimal; if it contains at least one dynamic gate it is dynamic; if both conditions are satisfied it is a MDM.

The MDM for the DPFT in Fig. 1.5 are in Fig. 1.7.

3.3 Transient Analysis by Modularization

For each MDM detected, *DPFTproc* generates its XML representation in the same formalism adopted by *DrawNET*, adding automatically the colour classes and subclasses associated to every parameter appearing in the DPFT; for example the XML code for Q_F module is

```
<?xml version='1.0' encoding='UTF-8' standalone='no'?>
<PFT name='report_Q_F' visibility='false' Title='modulo_SSM_min_di_report'>
  <ParameterType name='ParameterType1' visibility='false'
   ParameterColor='C1' ParameterName='i'/>
  <ParameterType name='ParameterType2' visibility='false'
   ParameterColor='C2' ParameterName='j'/>
  <ParameterType name='ParameterType3' visibility='false'
   ParameterColor='C3' ParameterName='k'/>
```

```
  <ParameterType name='ParameterType4' visibility='false'
   ParameterColor='C4' ParameterName='h'/>
  <ColorClass name='C1' visibility='false' SubClassList='c11'
   Ordered='false'/>
  <ColorClass name='C2' visibility='false' SubClassList='c21'
   Ordered='false'/>
  <ColorClass name='C3' visibility='false' SubClassList='c31'
   Ordered='false'/>
  <ColorClass name='C4' visibility='false' SubClassList='c41'
   Ordered='false'/>
  <ColorSubClass name='c11' visibility='false' ElementList='i{1-3}/>
  <ColorSubClass name='c21' visibility='false' ElementList='i{1-3}/>
  <ColorSubClass name='c31' visibility='false' ElementList='k{1-2}/>
  <ColorSubClass name='c41' visibility='false' ElementList='h{1-2}/>
  <TopEvent name='TopEvent0' visibility='false' Label='TE'/>
  <WSP name='WSP1' visibility='false'/>
  <Arc name='Arc0' visibility='false' from='WSP1' to='TopEvent0'/>
  <BasicEvent name='Q' visibility='false' Distribution='ALL EXP 0.001'
   Label='Q' Parameters=''/>
  <Arc name='Arc1' visibility='false' from='Q' to='WSP1'/>
  <BasicReplicatorEvent name='SP(h)' visibility='false' PredSWN=''
   DeclaredParameters='h' Distribution='ALL EXP 0.001' Label='SP'
   PList='' Parameters='h' alfa='0.1'/>
  <Arc name='Arc2' visibility='false' from='SP(h)' to='WSP1'/>
</PFT>
```

DPFTproc first writes in the XML file the declarations about the parameters and their colour classes and subclasses; all the parameters are declared even though some of them are not used in the module. Then, *DPFTproc* considers the root of the module and declares it as the Top Event; the declarations of the descendent arcs and nodes follow.

Every module XML code is passed to *XMLfilter* that translates it in the *dpft2swn* formalism; in the case of the Q_F module, the new XML code is

```
<?xml version='1.0'?>
<!DOCTYPE TREE SYSTEM 'FaultTreeLast.dtd'>
<TREE name='report_Q_F'>
   <TOP name='TE'/>
   <DECL>
      <CLASS_DECL name='C1' ordered=''>
          <SBC name='c11' descr='i{1-3}'/>
      </CLASS_DECL>
```

```
      <CLASS_DECL name='C2' ordered=''>
          <SBC name='c21' descr='i{1-3}'/>
      </CLASS_DECL>
      <CLASS_DECL name='C3' ordered=''>
          <SBC name='c31' descr='k{1-2}'/>
      </CLASS_DECL>
      <CLASS_DECL name='C4' ordered=''>
          <SBC name='c41' descr='h{1-2}'/>
      </CLASS_DECL>
      <PARAM_DECL name='i' type='C1'/>
      <PARAM_DECL name='j' type='C2'/>
      <PARAM_DECL name='k' type='C3'/>
      <PARAM_DECL name='h' type='C4'/>
   </DECL>
   <BE name='Q'>
      <ALL/>
      <DISTR type='EXP' description='0.001'/>
   </BE>
   <BE name='SP'>
      <PARAM>h</PARAM>
      <ALL/>
      <DISTR type='EXP' description='0.001'/>
   </BE>
   <GATE type='wsp' name='wsp0'>
      <INPUT name='Q'>
      </INPUT>
      <INPUT name='SP'>
         <PARAM>h</PARAM>
         <DECL_PARAM>h</DECL_PARAM>
      </INPUT>
      <OUTPUT name='TE'>
      </OUTPUT>
   </GATE>
</TREE>
```

dpft2swn formalism is more structurated than *DrawNET* formalism even if it describes the same module; this XML file begins with the declarations of colour classes and subclasses followed by the parameters; then, there are basic events declarations with their label and probability distribution; the file ends with gates declarations: for each gate there is the indication of its name and type followed by its input events and its output event.

XMLfilter makes some adaptions too:

- for PAND and SEQ gates it declares their input events in the failure order specified by means of order arcs;
- for FDEP gate it declares first the trigger event followed by the dependent events;
- for WSP gate it declares first the main component, then the spare components set.

Now *DPFTproc* runs *dpft2swn* on each MDM new XML representation in order to generate its SWN representation in *GreatSPN* [12] format; a file with extension .net (containing SWN places and transitions) and a file with extension .def (containing colour classes and subclasses) are created in the **nets** directory for each module; since the .def file must contain the result to calculate on the net too, *DPFTproc* runs Add_Prob on it in order to add the request of the module failure probability.

The .net file contains both the logic and graphic description of a SWN; the .net file for the SWN corresponding to Q_F module follows:

```
|0|
|
f   0   5   0   5   1   0   0
Q_dn    0 3.000000 1.800000 3.000000 1.550000 0
SP_na    0 8.399998 4.700000 8.150000 4.450000 0 8.150000 4.950000 C4
SP_dn    0 8.399998 8.450000 8.150000 8.200000 0 8.150000 8.700000 C4
TE_dn    0 20.899998 1.800000 20.899998 1.550000 0
SP_curr    0 17.299998 3.300000 17.299997 3.050000 0 17.049997 3.550000 C4
G1  0.000000 0.000000 1
Q_f  0.001000   1   0   0 1 1.20 1.80 0.40 1.55 0.00 1.80 0
   1
   1   1   0 0
   1
   1   1   2 0
1.800000 2.400000
2.400000 2.400000
SP_fail_OFF  0.000100   1   0   0 1 3.0 4.7 2.75 4.45 2.75 4.9 0
   2
   1   3   0 0 -0.150000 0.090000 <h>
   1   2   0 0 -0.150000 0.090000 <h>
   1
```

```
   1   2   1 0 -0.150000 0.090000 <h>
3.250000 4.950000
SP_fail_ON  0.001000   1   0   1 0 17.30 1.80 17.05 1.55 17.05 2.00 0
   1   5   0 0 -0.150000 0.090000 <h>
   1
   1   3   2 0 -0.150000 0.090000 <h>
17.120406 2.398641
8.472315 8.039657
   0
Q_spare  1.000000   1   1   1 0 14.60 2.55 14.01 2.20 14.01 2.75 0
   1   1   2 0
13.983502 2.447251
3.410343 1.872317
   3
   1   1   2 0
13.983502 2.447251
3.072317 1.389657
   1   5   0 0 -0.150000 0.090000 <h>
   1   2   2 0 -0.150000 0.090000 <h>
14.841334 3.126526
8.741196 4.460844
   2
   1   2   2 0 -0.150000 0.090000 <h>
15.041939 2.108058
8.472315 4.289657
   1   5   1 0 -0.150000 0.090000 <S>
14.799998 2.350000
TE_fail  1.000000   1   1   2 1 19.10 1.80 18.60 1.45 18.60 2.00 0
   1   2   2 0 -0.150000 0.090000 <S>
18.613277 2.192081
8.810341 4.772317
   1   1   2 0
18.658057 1.358058
3.239156 2.141197
   3
   1   4   0 0
   1   2   2 0 -0.150000 0.090000 <S>
18.613277 2.192081
8.576006 4.322333
   1   1   2 0
18.658057 1.358058
```

```
3.341197 1.560844
   2
   1   5   1 0 -0.150000 0.090000 <S>
19.099998 2.300000
   1   4   1 0
19.599998 2.300000
```

The .def file for the SWN corresponding to Q_F module contains the result to calculate on it and the list of colour classes and subclasses:

```
|256
%
|prob 0.2 0.2 : p{#TE_dn=1};
|
(C1 c 0.500000 0.500000 (@c
u c11
))
(c11 c 0.700000 0.700000 (@c
i{1-3}
))
(C2 c 1.500000 0.500000 (@c
u c21
))
(c21 c 1.700000 0.700000 (@c
i{1-3}
))
(C3 c 2.500000 0.500000 (@c
u c31
))
(c31 c 2.700000 0.700000 (@c
k{1-2}
))
(C4 c 3.500000 0.500000 (@c
u c41
))
(c41 c 3.700000 0.700000 (@c
h{1-2}
))
```

At this point, every MDM is ready to be analyzed as a SWN; *DPFTproc*, for each time value requested by the user makes these operations:

- runs *swn_sym_tr* (SWN symbolic transient solver included in the GreatSPN package) on each MDM SWN at the current time;
- looks for the result for each MDM in the .sta file generated by *swn_sym_tr* in the **nets** directory;
- replaces each MDM with a basic event whose failure probability is equal to such result;
- generates the reduced PFT XML representation according to *DrawNET* formalism; for instance, at time 1000 the reduced PFT (Fig. 1.8) XML representation is

```
<?xml version='1.0' encoding='UTF-8' standalone='no'?>
<PFT name='report_TOP' visibility='false' Title='modulo_CSM_MAX'>
  <ParameterType name='ParameterType1' visibility='false'
  ParameterColor='C1' ParameterName='i'/>
  <ParameterType name='ParameterType2' visibility='false'
   ParameterColor='C2' ParameterName='j'/>
  <ParameterType name='ParameterType3' visibility='false'
   ParameterColor='C3' ParameterName='k'/>
  <ParameterType name='ParameterType4' visibility='false'
   ParameterColor='C4' ParameterName='h'/>
  <ColorClass name='C1' visibility='false' SubClassList='c11'
   Ordered='false'/>
  <ColorClass name='C2' visibility='false' SubClassList='c21'
   Ordered='false'/>
  <ColorClass name='C3' visibility='false' SubClassList='c31'
   Ordered='false'/>
  <ColorClass name='C4' visibility='false' SubClassList='c41'
   Ordered='false'/>
  <ColorSubClass name='c11' visibility='false' ElementList='i{1-3}/>
  <ColorSubClass name='c21' visibility='false' ElementList='i{1-3}/>
  <ColorSubClass name='c31' visibility='false' ElementList='k{1-2}/>
  <ColorSubClass name='c41' visibility='false' ElementList='h{1-2}/>
  <TopEvent name='TopEvent0' visibility='false' Label='TE'/>
  <Or name='Or1' visibility='false'/>
  <Arc name='Arc0' visibility='false' from='Or1' to='TopEvent0'/>
  <Event name='SYS1' visibility='false' Label='SYS1' Parameters=''/>
  <Or name='Or2' visibility='false'/>
  <Arc name='Arc1' visibility='false' from='Or2' to='SYS1'/>
  <ReplicatorEvent name='SUB(i)' visibility='false' PredSWN=''
```

```
 DeclaredParameters='i' Label='SUB' PList='' Parameters='i'/>
<And name='And3' visibility='false'/>
<Arc name='Arc2' visibility='false' from='And3' to='SUB(i)'/>
<BasicEvent name='A(i)' visibility='false' Distribution='ALL EXP 0.001'
 Label='A' Parameters='i'/>
<Arc name='Arc3' visibility='false' from='A(i)' to='And3'/>
<BasicReplicatorEvent name='B(j)' visibility='false' PredSWN=''
 DeclaredParameters='j' Distribution='ALL EXP 0.001' Label='B'
 PList='' Parameters='j' alfa='1'/>
<Arc name='Arc4' visibility='false' from='B(j)' to='And3'/>
<Arc name='Arc5' visibility='false' from='SUB(i)' to='Or2'/>
<Arc name='Arc6' visibility='false' from='SYS1' to='Or1'/>
<BasicEvent name='SYS2' visibility='false'
 Distribution='ALL DET 0.168386965990' Label='MOD_SYS2'
 Parameters=''/>
<Arc name='Arc7' visibility='false' from='SYS2' to='Or1'/>
<Event name='SYS3' visibility='false' Label='SYS3' Parameters=''/>
<Or name='Or4' visibility='false'/>
<Arc name='Arc8' visibility='false' from='Or4' to='SYS3'/>
<BasicEvent name='E' visibility='false' Distribution='ALL EXP 0.001'
 Label='E' Parameters=''/>
<Arc name='Arc9' visibility='false' from='E' to='Or4'/>
<BasicEvent name='Q_F' visibility='false'
 Distribution='ALL DET 0.098805271089' Label='MOD_Q_F'
 Parameters=''/>
<Arc name='Arc10' visibility='false' from='Q_F' to='Or4'/>
<Arc name='Arc11' visibility='false' from='SYS3' to='Or1'/>
</PFT>
```

SYS2 and Q_F modules have been replaced by a basic event whose failure probability is equal to the failure probability of the module at time 1000.

- *DPFTproc* runs *drawnet2sharpe* on the reduced PFT XML representation, in order to generate the *Sharpe* version of it with the indication of the current time; at time 1000 this is the *Sharpe* version:

```
ftree report_TOP

repeat MOD_SYS2 prob(0.168386965990)
repeat E exp(0.001)
```

```
repeat MOD_Q_F prob(0.098805271089)
repeat B1 exp(0.001)
repeat B2 exp(0.001)
repeat B3 exp(0.001)
repeat A1 exp(0.001)
repeat A2 exp(0.001)
repeat A3 exp(0.001)

or SYS3 E MOD_Q_F
and SUB1 A1 B1 B2 B3
and SUB2 A2 B1 B2 B3
and SUB3 A3 B1 B2 B3
or SYS1 SUB1 SUB2 SUB3
or TOP SYS1 MOD_SYS2 SYS3
end

format 8

eval(report_TOP) 1000.00 1000.00 1000.00
end
```

Sharpe does not support PFT, so the PFT must be unfolded to the corrisponding FT by *drawnet2sharpe*;

- *DPFTproc* runs *Sharpe* on the reduced PFT performing the transient analysis at the current time; for 1000, *Sharpe* returns:

```
system report_TOP
        t             F(t)

 1.00000000 e+03  7.90465249 e-01
```

- *DPFTproc* looks for the *Sharpe* result and appends it in the file named **tr_results** in the **XML** directory containing the final result for each requested time value.

At the end of this cycle the transient analysis has been completed and *DPFTproc* execution ends; **tr_results** is the file that *DrawNET* visualizes now in a window; in the case of our example, what this file contains is shown in Fig. 2.12.

There are two special cases where modularization can not be applied:

- the whole DPFT is a minimal dynamic module (for instance the Top Event is the output of a dynamic gate);
- the whole DPFT is a maximal static module (if the tree contains no dynamic gate; it is really a PFT or a FT).

In the first case *DPFTproc* generates the SWN of the DPFT and performs the transient analysis directly on it without modularization and subtrees replacements; in the second case the transient analysis of the PFT will be performed directly by *Sharpe* after having translated the PFT in its *Sharpe* version by means of *drawnet2sharpe*.

Chapter 4

Dynamic Gates Translation in SWN

This chapter is about the way *dpft2swn* translates *dynamic gates* into SWN; for each dynamic gate we will see and comment its representation in SWN, while about static gates (AND, OR, K:N) translation, some information can be found in [15] (*dpft2swn* is an extension of *pft2swn* tool, previously realized, able to translate PFT in SWN).

4.1 Priority And

Fig. 4.1 shows the Petri net for the gate in Fig. 1.1; the components A, B and C are represented by the places with the same name; for instance, if a token appear in **A** by means of **A_fail** timed transition, it means that A is failed; the firing rate of **A_fail** is equal to the failure rate of component A.

A token in the place named **AB** indicates that A and B are failed and that A failed before B; a token in the place named **ABC** indicates that A, B, C failed and they did in such an order (the gate has failed); a token in the place named **Oper** indicates that the failure order has not been respected so the gate has not failed (operative).

The transitions named **pand_0** and **pand_1** bring the token to **ABC** if the order is respected; if it is not respected the transition **pand_2** or **pand_3** puts a token in **Oper** inhibiting **pand_0** and **pand_1**; this logic can be applied to a PAND gate with two, three, four or more input events; for example, if we exclude C from the PAND input events, the Petri net will appear as in Fig. 4.2.

An input event to a PAND may be a basic event or an internal event; in the second case we will have in the Petri net a subnet connected to the place

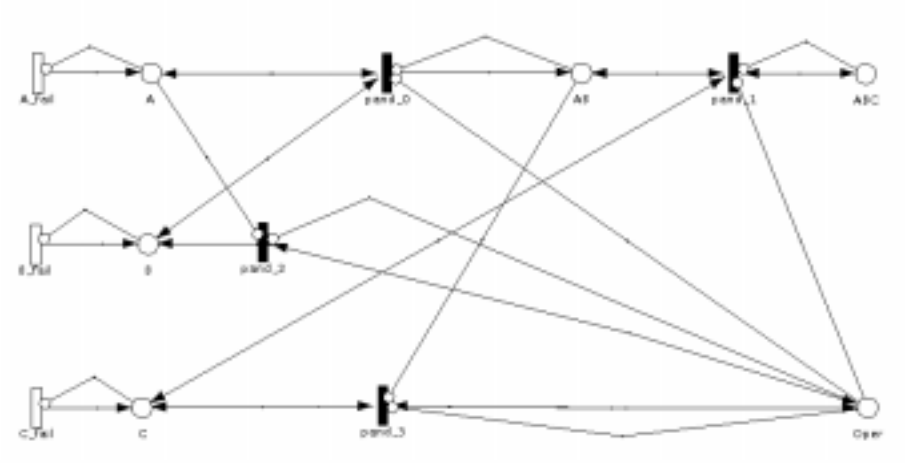

Figure 4.1: Petri net for PAND gate with three input events

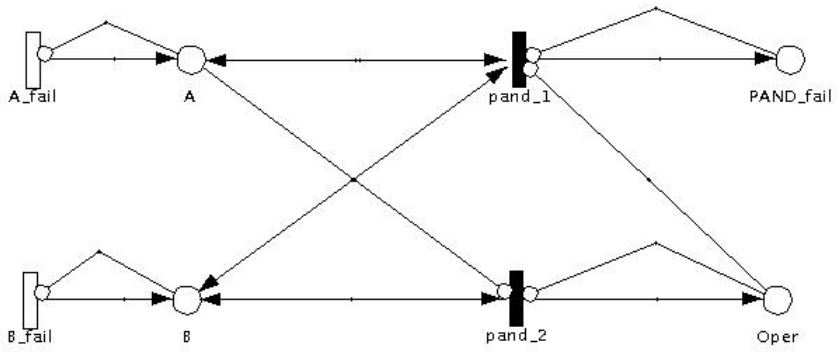

Figure 4.2: Petri net for PAND gate with two input events

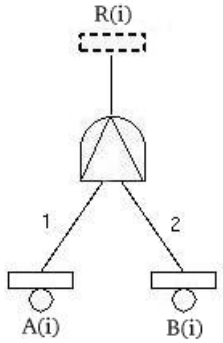

Figure 4.3: PAND gate with parameters

representing the internal event, instead of a timed transition.

dpft2swn does not support yet a PAND gate whose input event is a replicated event; anyway a replicated event may be its output event or a PAND gate may descend from a replicated event; in these cases, the input and output events of the PAND gate may have some parameters associated to.

An example is the PAND gate in Fig. 4.3 where the output of the gate is the replicator event R(i) whose parameter is i and the input basic events are A and B having i as parameter too; this means that for each value of i, A(i) must occur before B(i) to have the failure of R(i). Assuming that the colour class associated with i is C1, the SWN for such a gate is shown in Fig. 4.4.

4.2 Functional Dependency Gate

Fig. 4.5 shows the Petri net for the gate in Fig. 1.2; a token representing the failure may appear in **A** or **B** place (dependent components) by means of **fdep_2** or **fdep_3** transition that fires when a token in T appears (T fails); a token in **A** or **B** place may already be present because the component has failed by its own.

fdep_1 transition fires when T fails, to represent the failure of the gate whose state is equal to the state of the trigger.

FDEP gate may have a replicated dependent event as in Fig. 4.6 whose corresponding Petri net is shown in Fig. 4.7; in this case, the function on the arcs connecting **fdep_2** transition to the coloured place **D** means that every still working D(i) fails when the trigger event occurs.

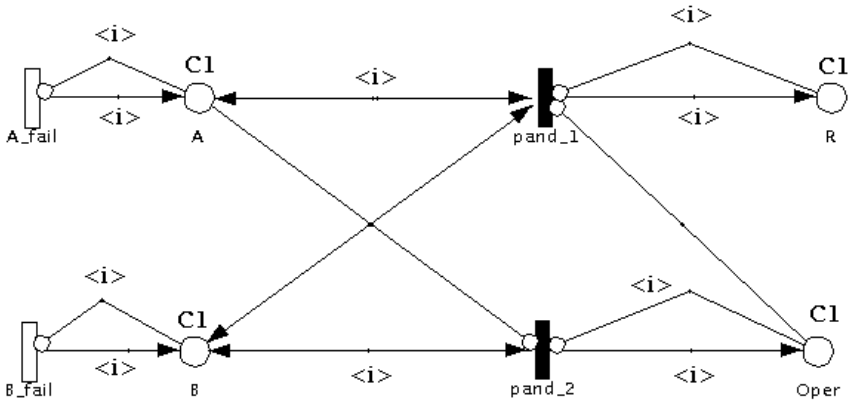

Figure 4.4: SWN for a PAND gate with parameters

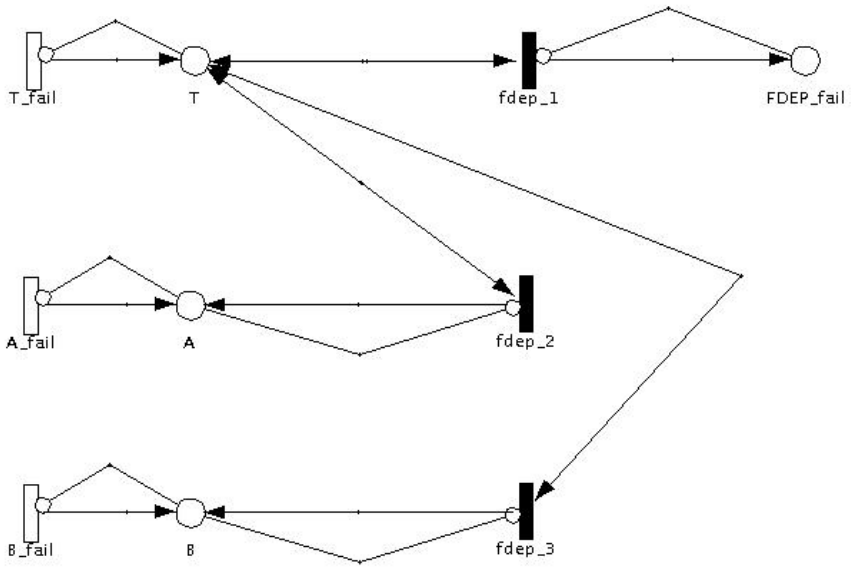

Figure 4.5: Petri net for FDEP gate with two dependent input events

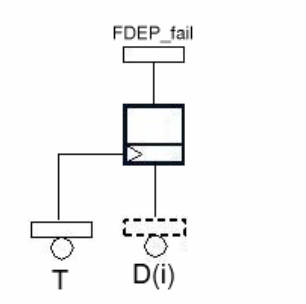

Figure 4.6: FDEP with a replicated dependent event

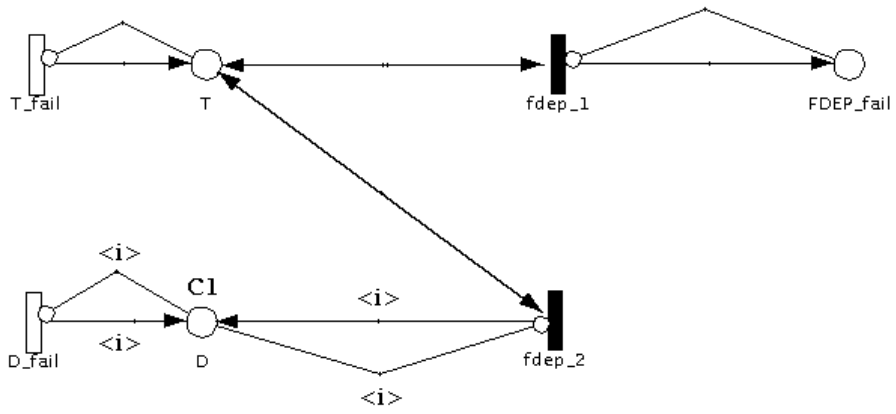

Figure 4.7: Petri net for FDEP with a replicated dependent event

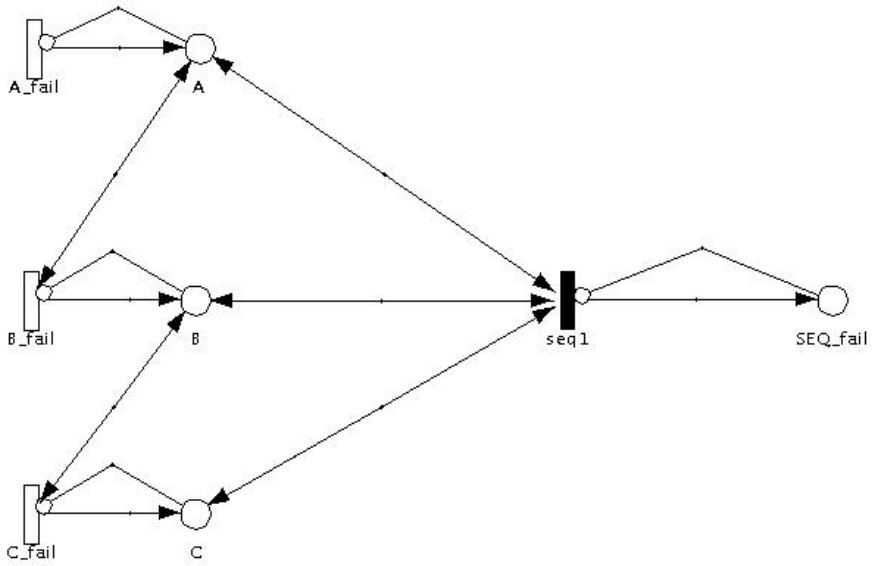

Figure 4.8: Petri net for SEQ gate

4.3 Sequence Enforcing Gate

Fig. 4.8 shows the Petri net for the gate in Fig. 1.3; the firing of the transition representing the failure of B is enabled by the presence of a token in place **A** (A is failed); in the same way the failure of C is enabled by the fault state of B.

seq_1 transition fires when all input events occur, to represent the gate failure; in such a Petri net the first input event to occur may be an internal event instead of a basic event; in this case, a subnet would replace the timed transition for the first input event.

dpft2swn does not support yet a SEQ gate whose input event is a replicated event; anyway a replicated event may be its output event or a SEQ gate may descend from a replicated event; in these cases, the input and output events of the SEQ gate may have some parameters associated to.

An example is the SEQ gate in Fig. 4.9 where the output of the gate is the replicator event R(i) whose parameter is i and whose input basic events are A, B and C having i as parameter too; this means that for each value of i, B(i) is forced to occur after A(i) and C(i) is forced to occur after B(i). Assuming that the colour class associated with i is C1, the SWN for such a gate is shown in Fig. 4.10.

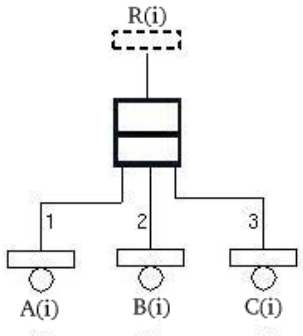

Figure 4.9: SEQ gate with parameters

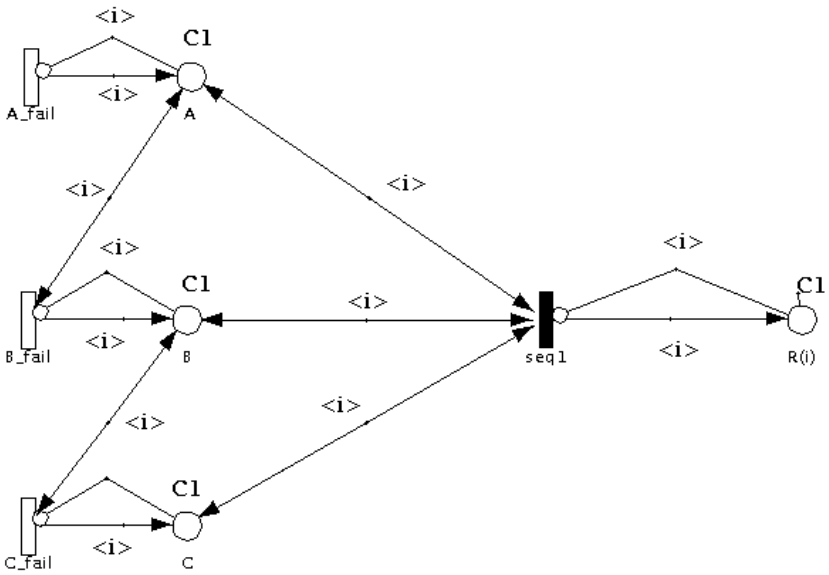

Figure 4.10: SWN for a SEQ gate with parameters

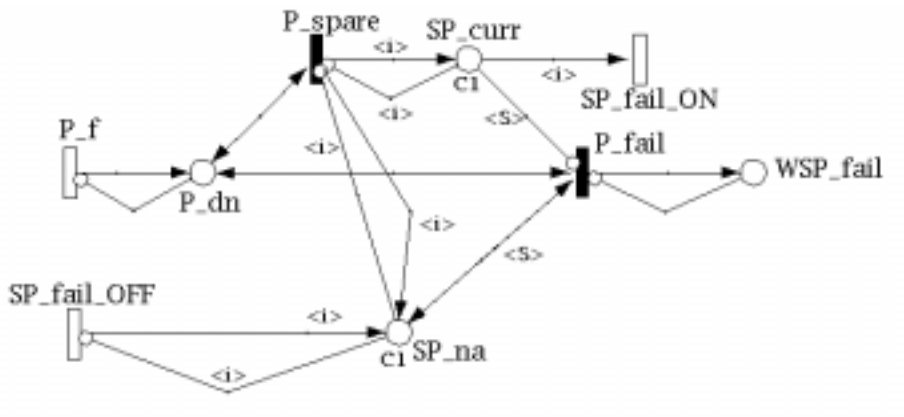

Figure 4.11: Petri net for WSP gate

4.4 Warm Spare Gate

Fig. 4.11 shows the Petri net for the gate in Fig. 1.4; in this case, the main component is represented by the basic event P while the set of spares is modeled as the basic replicator event SP whose parameter is i. We say that a spare is available if it is not already failed or it is not already working.

Let's consider the places of this SWN:

- **P_dn**; a token here represents the failure of the main component;
- **SP_curr** is a coloured place with the same colour class of the set of spares (C1); the presence here of a token whose colour is i means that in this moment the main component is replaced by the i-th spare;
- **WSP_fail**; a token here means that the gate has failed;
- **SP_na** contains the coloured tokens relative to the spares that are not available to replace the main component because they are already working or they are failed.

Let's consider now the transitions of the SWN and their effects:

- **P_f** is a timed transition whose firing means the main component failure; it puts a token in **P_dn** and it does not fire any more;
- **P_spare** fires immediately when P fails and if there is at least an available spare component; **P_spare** puts the coloured token corresponding to an available spare in **SP_curr** and in **SP_na** in order

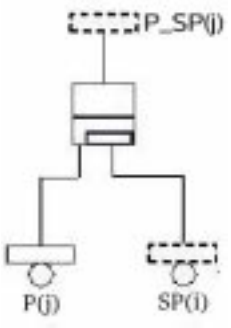

Figure 4.12: WSP gate with shared spares

to model that the main component has been replaced and that the reserved spare has changed to the working state; while there is a token in **SP_curr**, **P_spare** can not fire another time;

- **SP_fail_ON** models the failure of the currently working spare; its firing rate is equal to the working failure rate of the spare; when this transition fires the coloured token disappear from **P_curr**, so **P_spare** can fire again if there is at least an available spare;

- **SP_fail_OFF** is a timed transition to model the failure during the dormancy state of an available spare; its firing rate is equal to the stand-by failure rate of the spare; its effect is putting in **SP_na** a coloured token (the spare is now not available).

Assuming that for the basic replicator event representing the spares, λ is the failure rate and α is the dormancy factor, the firing rate for **SP_fail_OFF** transition will be $\alpha\lambda$; if $\alpha = 0$, **SP_fail_OFF** will not appear in the SWN. The firing rate for **SP_fail_ON** will be λ.

In the case of Fig. 1.4, there is one main component and a set of spares; the spares may be shared by several main components as in the case of the system Fig. 4.12; here, there is the parametric representation of a set of main components that share a set of spares; a spare may replace any of the main component and P_SP(j) occur when P(j) is failed and there are no available spares.

The SWN for this gate has the same structure of the SWN in Fig. 4.11, but some places are now coloured and some arc functions have been added, as shown in Fig. 4.13 where the parameter j is associated to the colour class C2; the modified places are:

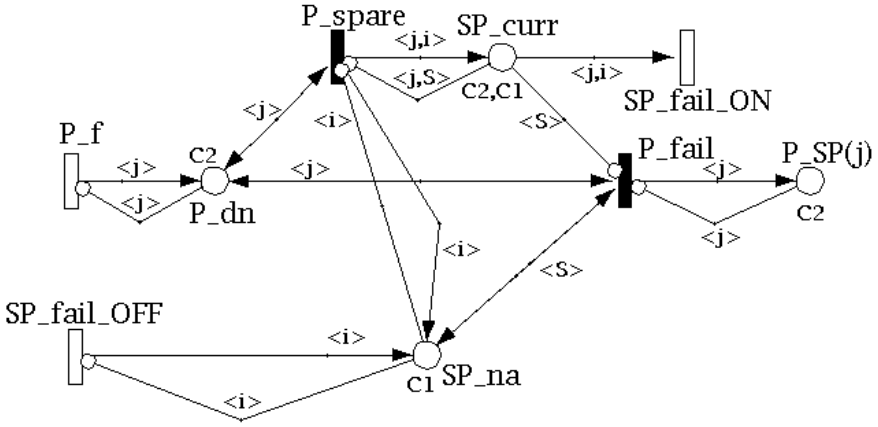

Figure 4.13: SWN for WSP gate with shared spares

- **P_dn** has colour C2 and contains the coloured tokens relative to the failed main components;
- **SP_curr** has now two colours in this order: C2, C1; when the j-th main component fails, the transition **P_spare** puts a token in **SP_curr** whose colour is $<j,i>$ to indicate that the j-th main component is now replaced by the i-th spare; at the same time, **P_spare** puts a token whose colour is $<i>$, in **SP_na** to indicate that the i-th spare is now not available; this transition can not fire again for the same j until the current spare replacing the j-th main component does not fail; this happens when the transition **SP_fail** fires;
- **P_SP** whose colour is C2 contains the tokens relative to the main components that can not be replaced because there are no available spares.

In the case of shared spares we may have more than one spare working at the same time, but any main component is always replaced by only one spare.

4.5 Modules composition

In the previous sections we explained how *dpft2swn* translates in SWN every dynamic gate, but in our tool we must translate in SWN some modules, not only dynamic gates; *dpft2swn* generates a SWN for each gate of a module; at this point the tool called *algebra* is run to connect all the SWN's together composing the SWN representing the whole module.

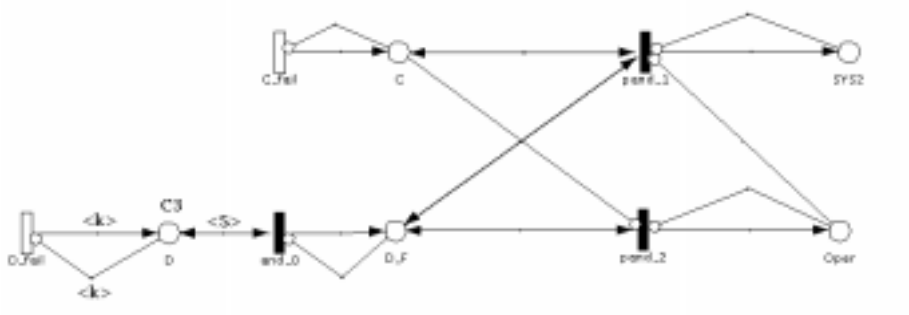

Figure 4.14: SWN for module SYS2

Let's consider the DPFT in Fig. 1.7 and its MDM named SYS2; it is composed by two gates: PAND, AND; its resulting SWN is shown in Fig. 4.14 where the SWN relative to the AND gate is connected to the SWN relative to the PAND gate by the place named **D_F**.

On this SWN the transient analysis will be performed for every requested time value.

Bibliography

[1] J. Bechta Dugan, S.J. Bavuso, and M.A. Boyd. Dynamic fault-tree models for fault-tolerant computer systems. *IEEE Transactions on Reliability*, 41:363–377, 1992.

[2] R. Manian, D.W. Coppit, K.J. Sullivan, and J.B. Dugan. Bridging the gap between systems and dynamic fault tree models. *Proceedings IEEE Annual Reliability and Maintainability Symposium*, pages 105–111, 1999.

[3] W.G. Schneeweiss. *The Fault Tree Method*. LiLoLe Verlag, 1999.

[4] A. Bobbio, G. Franceschinis, R. Gaeta, and L. Portinale. Parametric fault-tree for the dependability analysis of redundant systems and its high level Petri net semantics. *IEEE Transactions Software Engineering*, 29:270–287, 2003.

[5] A. Bobbio, G. Franceschinis, L. Portinale, and R. Gaeta. Dependability Assessment of an Industrial Programmable Logic Controller via Parametric Fault-Tree and High Level Petri Net. In *Proceedings 9th International Workshop on Petri Nets and Performance Models - PNPM01*, pages 29–38. IEEE Computer Society, 2001.

[6] A. Bobbio, G. Franceschinis, L. Portinale, and R. Gaeta. Exploiting Petri Nets to Support Fault-Tree Based Dependability Analysis. In *8-th International Conference on Petri Nets and Performance Models - PNPM99*, pages 146–155. IEEE Computer Society, 1999.

[7] Y. Dutuit and A. Rauzy. A linear-time algorithm to find modules of fault tree. *IEEE Transactions on Reliability*, 45:422–425, 1996.

[8] A. Anand and A. Somani. Hierarchical analysis of fault trees with dependencies, using decomposition. *Proceedings IEEE Annual Reliability and Maintainability Symposium*, pages 69–75, 1998.

[9] G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. Stochastic Well-Formed coloured nets and multiprocessor modelling applications. In K. Jensen and G. Rozenberg, editors, *High-Level Petri Nets. Theory and Application*. Springer Verlag, 1991.

[10] V.Vittorini G.Franceschinis M.Gribaudo M.Iacono C.Bertoncello. DrawNet++: a Flexible Framework for Building Dependability Models. *Proceedings International Conference on Dependable Systems and Networks*, 2002.

[11] V. Vittorini, G. Franceschinis, M. Gribaudo, M. Iacono, and N. Mazzocca. DrawNet++: Model objects to support performance analysis and simulation of complex systems. In *Proceedings 12th International Conference on Modelling Tools and Techniques for Computer and Communication System Performance Evaluation (TOOLS 2002)*, pages 233–238, London, 2002. Springer Verlag - LNCS, Vol 2324.

[12] G. Chiola, G. Franceschinis, R. Gaeta, and M. Ribaudo. GreatSPN 1.7: Graphical Editor and Analyzer for Timed and Stochastic Petri Nets. *Performance Evaluation*, 24:47–68, November 1995.

[13] D. Codetta Raiteri. drawnet2sharpe & sharpe2astra User's manual. http://143.225.250.111/Iside/, 2002.

[14] R.A.Sahner K.S.Trivedi A.Puliafito. *Performance And Reliability Analysis Of Computer Systems; An Example-Based Approach Using the SHARPE Software Package*. Kluwer Academic Publishers, 1996.

[15] C.Bertoncello. ptf2swn: technical report, http://143.225.250.111/iside/, 2001.

[16] D. Codetta Raiteri. Sviluppo di Formalismi per Alberi dei Guasti con Nodi Dipendenti e Riparabili. Master's thesis, Università degli Studi del Piemonte Orientale "Amedeo Avogadro", A. A. 2001-2002.