Fault tolerance Adequation in Syndex

Thomas Lévêque

September 15, 2004

Contents

Chapter 1

Conception Document

1.1 Required data

- an algorithm graph
- an architecture graph

To make Adequation :

- Faillures patterns which want to tolerate

To view schedule :

- one or more effective fails

1.2 Result data

- One schedule which tolerate faillures patterns

1.3 Global Adequation Algorithm

Five points :

1. Build of an architecture reduce for each failure pattern
2. Generate a simple schedule for each failure pattern
3. Union of all graphs
4. Make a pseudo-topological order for the graph
5. Calculate start time, end time and timeout for each task

1.3.1 Build of an architecture reduce for each failure pattern

The procedure `failures_patterns_check ()` checks failures patterns , in the others words, they don't generate disjoint architectures and remove the subsets of another failure pattern. We should use it before make any adequation.

1.3.2 Generate a simple schedule for each failure pattern

During adequation, we don't allow to use a component included in the failure pattern. The function `routes o1 o2 exclusives_routes` returns the shortest route list between `o1` and `o2` which not contain a component included in `exclusives_routes`. The function `best_opr_esfs_sp operation operators graph fp` returns optimal esfs, the best operator for this operation `operation` included in `operators` and not included in `fp`. If there isn't operator, an exception is throwed. The `fault_tolerant_adequation fp graph` function call generate adequation of the graph `graph` with fail components are the set `fp`.

1.3.3 Union of all graphs

The function `union_graph graph_list` returns a graph which is the union of `graph_list`. To make union, we add each graph of the list to the first . The procedure `union_2_graph g1 g2` add `g1` to `g2`. For all the nodes (operation type) of graph, add them with procedure `operation_add` which change operation name (same name with `#operator` at the end) if it is necessary if they are not in the other graph (same operation name and same operator). The function `op_exist` returns a boolean which represents the presence of the node. Finally, for all nodes of the graph, we update successors and predecessors. Be carefull, we must update four lists of each task : `t_opn_dependences_predecessors`, `t_opn_dependences_successors`, `t_opn_predecessors` and `t_opn_successors`. Use function `dependence_add` to update these four lists and add a data dependency on graph.

Algorithm 1 *Build graph list union*

```
let union_graph graph_list =
  let union_2_graph g1 g2 =
    let op_exist operation graph =
      let op_list = (Hashtbl.find_all graph (identifier_of_operation operation))
           @ (Hashtbl.find_all graph ((identifier_of_operation operation)^"#"
              ^(name_of_operator (operator_of_operation operation))))
      in
      List.exists (
         fun op -> (name_of_operator (operator_of_operation op))=
                   (name_of_operator (operator_of_operation operation))) op_list
     in
    Hashtbl.iter (fun name_operation -> fun operation ->
               if not(op_exist operation g1)
               then operation_add g1 operation
               ) g2;
    Hashtbl.iter (fun name_operation -> fun operation ->
               %Update successors et predecessors of g1% ) g2;
   in
  List.iter (fun g -> union_2_graph (List.hd graph_list) g) graph_list;
  List.hd graph_list
```

1.3.4 Make a pseudo-topological order for the graph

Let G graph which we would like calculate a pseudo topological order ϕ and N the noeud list of G.

Algorithm 2 *Calculate a pseudo-topological order for the graph*

Properties :

1. Any predecessor `w'` of `w` which his operation is different from `w` operation and there isn't edge between `w'` and `w` in the original algorithm graph not ordinate.
2. There a predecessor `w'` ordinate which his operation is equal to `w` or an edge between `w'` and `w` in the original algorithm graph.

3. For all data dependency dp which is predecessor of w in the original algorithm graph, there is a ordinate predecessor which corresponding to dp.

4. w must not be included in X.

```
begin
   let k = ref 1 and X = ref [] in
   let checkPred w = %Prop 1% and
       checkDep w = match operator_of_operation w with
                    |Operator _ -> true
                    |Media m -> %Prop 2% and
       checkOp w = match operator_of_operation w with
                   |Media _ -> true
                   |Operator op -> %Prop 3% and
       checkNotOrd w = %Prop 4% in
   let check w = checkNotOrd w && checkPred w && checkDep w && checkOp w in
   while !k<=length(N) do
     let w = List.find check N in
     X:= w :: !X;
     phi.(k) := w;
     k := !k + 1;
   done
   phi
end
```

1.3.5 Calculate start time, end time and timeout for each task

We try to find a fix point for timeout.

Algorithm 3 *Calculate start time, end time and timeout of a graph*

```
begin
   forall (node v) do teta.(0).v:=0;
   i:=0;
   repeat
      i:=i+1;
      forall (failurePattern F) do
        if isStarving v F then teta.(F).v = teta.(i-1).v
            else teta.(F).v = infinite;
        let (alpha.(F),beta.(F),epsilon.(F)):= minimalExecution teta.(F);
      end for
      forall (node v) do teta.(i).v:=(max alpha v )+ 1
   until teta.(i) = teta.(i+1)
end
```

The procedure `minimal_execution_computation phi teta starvingNodes` computes minimal execution. It needs a pseudo topological order `phi`, timeout `teta` and the list of the nodes which not receive all their datas `starvingNodes`. It updates all the esfs, start time of each task. We use four datas to make this computation :

- `beta` represents all the end times.
- `epsilon` checks if an operation is executed.
- `notCalc` checks if an operation is already calculated.
- `empty_times` is the set of all the times where each operator or media is not used.

In the order of the pseudo-topological order, we compute start time and end time for each operation. This computation may fail if all the predecessors aren't already calculated, we indicate that this operation is not calculated. The procedure `alpha_computation` does this computation . When this computation not fails, we compute all the predecessors in the order which have not been calculated.

1.4 Failures patterns files

This files have .fp extension and are text files with this rules :

- a failure pattern by each line
- no empty line
- a failure pattern is represent by each name of fail component separated by exactly one space
- each line doesn't begin or end by a delimiter

Two failure patterns exemple : one line describe `proc1` and `proc2` as failed and another line with `com3`, `operator6` and `operator8` out of order :

```
proc1 proc2
com3 operator6 operator8
```

1.5 Bugs

- Nothing ! So far !

1.6 Future work

- Use progress box during fusion proceed

1.7 Adequation Tests

We generate randomly 50 algorithms for each CCR (Communication to Computation Ratio) and number of tasks with `link`.
Tested CCR :

- CCR = 0.1
- CCR = 1
- CCR = 10

Tested number of tasks :

- 25
- 50
- 75
- 100

We compute the length of the critical path for each produced schedule and compare with those obtained with other adequation algorithms (FTBAR). To speed up making tests, we call the function `minimal_execution_computation` one time with an infinite timeout for each operation and no starving operation (empty list) instead of computing timeout. Then we look after the highest end time of all operations and return it. These changes are to do in the file `fault_tolerance_with_fp.ml`. The following process is to automate the tests. We insert the following code in the main loop of the file `ihm_ctk.ml` :

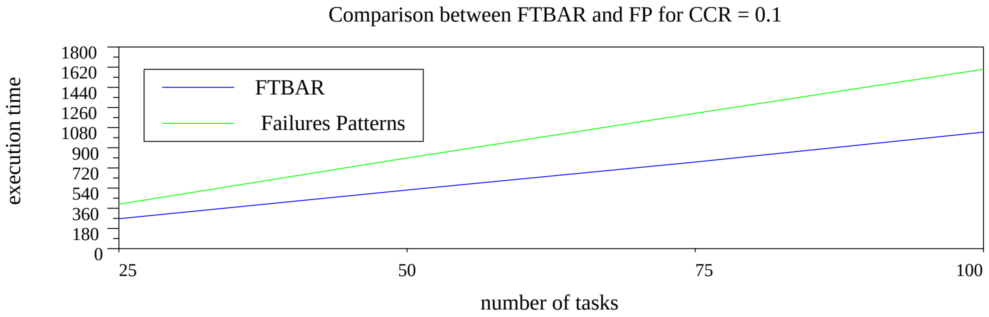
```
for k = 1 to 4 do
  let number_of_task = (25*k) in
    for j = 1 to 3 do
      let ccr = match j with
| 1 -> 0.1
| 2 -> 1.
| 3 -> 10.
      in
for num = 1 to 50 do
  let test_file = ("alg"^(string_of_int (number_of_task/25))
             ^"_red"^(string_of_int num)^"_ccr"
     ^(string_of_float ccr)^"_archi_p4.sdx") in
    open_file_name test_file ;
    let time_test = (fault_adeq_test ()) in
    ps (string_of_float time_test);
        done;
   done;
done
```

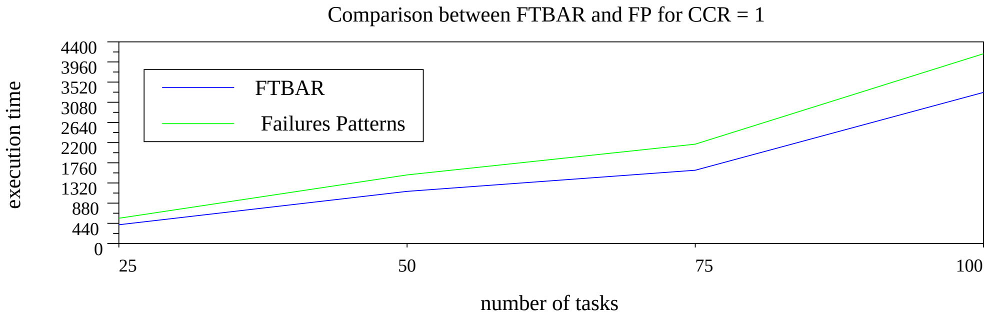
The `test_file` value is the name of the files which have to make tests. The `fault_adeq_test` function is a call of the desire adequation function and then the computation of execution time. To make following graphs, we use Scilab with the following script :

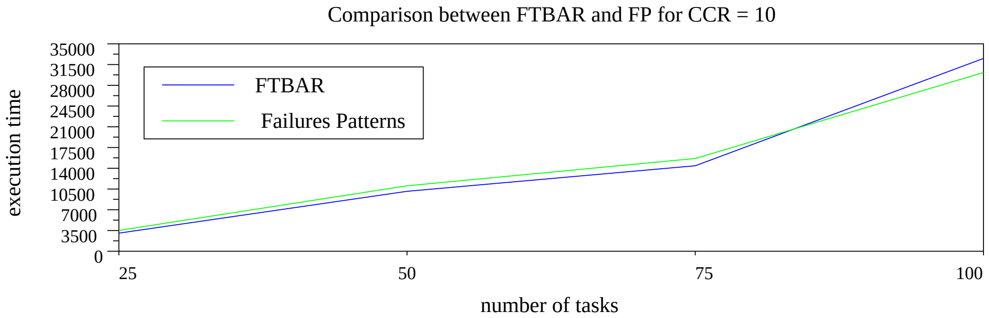
```
NBTASKS=[25 50 75 100];
// FTBAR tests read
FT25C01=fscanfMat('25_tasks_ccr_0.1.ftbar');
FT25C1=fscanfMat('25_tasks_ccr_1.ftbar');
FT25C10=fscanfMat('25_tasks_ccr_10.ftbar');
FT50C01=fscanfMat('50_tasks_ccr_0.1.ftbar');
FT50C1=fscanfMat('50_tasks_ccr_1.ftbar');
FT50C10=fscanfMat('50_tasks_ccr_10.ftbar');
FT75C01=fscanfMat('75_tasks_ccr_0.1.ftbar');
FT75C1=fscanfMat('75_tasks_ccr_1.ftbar');
FT75C10=fscanfMat('75_tasks_ccr_10.ftbar');
FT100C01=fscanfMat('100_tasks_ccr_0.1.ftbar');
FT100C1=fscanfMat('100_tasks_ccr_1.ftbar');
FT100C10=fscanfMat('100_tasks_ccr_10.ftbar');
// Failures Patterns tests read
T25C01=fscanfMat('25_tasks_ccr_0.1.test');
T25C1=fscanfMat('25_tasks_ccr_1.test');
T25C10=fscanfMat('25_tasks_ccr_10.test');
T50C01=fscanfMat('50_tasks_ccr_0.1.test');
T50C1=fscanfMat('50_tasks_ccr_1.test');
T50C10=fscanfMat('50_tasks_ccr_10.test');
T75C01=fscanfMat('75_tasks_ccr_0.1.test');
T75C1=fscanfMat('75_tasks_ccr_1.test');
T75C10=fscanfMat('75_tasks_ccr_10.test');
T100C01=fscanfMat('100_tasks_ccr_0.1.test');
```

```
T100C1=fscanfMat('100_tasks_ccr_1.test');
T100C10=fscanfMat('100_tasks_ccr_10.test');
// functions
deff('[x]=average(y)','x=sum(y)/length(y)');
// averages
FTAV=list([average(FT25C01);average(FT50C01);average(FT75C01);average(FT100C01)],...
[average(FT25C1);average(FT50C1);average(FT75C1);average(FT100C1)],...
[average(FT25C10);average(FT50C10);average(FT75C10);average(FT100C10)]);
FPAV=list([average(T25C01);average(T50C01);average(T75C01);average(T100C01)],...
[average(T25C1);average(T50C1);average(T75C1);average(T100C1)],...
[average(T25C10);average(T50C10);average(T75C10);average(T100C10)]);
// graphs
xsetech([0.,0.,1.,0.33],[-1,1,-1,1]);
rect=[0,0,100,1500];
leg="FTBAR@Failures Patterns";
plot2d(NBTASKS,[FTAV(1) FPAV(1)],[2 3],"111",leg,rect)
xtitle("Comparison between FTBAR and FP for CCR = 0.1","number of tasks","execution time")
xsetech([0.,0.33,1.,0.33],[-1,1,-1,1]);
rect=[0,0,100,4000];
leg="FTBAR@Failures Patterns";
plot2d(NBTASKS,[FTAV(2) FPAV(2)],[2 3],"111",leg,rect)
xtitle("Comparison between FTBAR and FP for CCR = 1","number of tasks","execution time")
xsetech([0.,0.66,1.,0.34],[-1,1,-1,1]);
rect=[0,0,100,35000];
leg="FTBAR@Failures Patterns";
plot2d(NBTASKS,[FTAV(3) FPAV(3)],[2 3],"111",leg,rect)
xtitle("Comparison between FTBAR and FP for CCR = 10","number of tasks","execution time")
```

`NBTASKS` is the set of the different numbers of tasks. For each case (number of tasks, CCR and adequation type), we create a vector of results. We define the function `average` as the average of all elements of a vector. Then we create an average list for each adequation type. The View parameters are `rect` which represents scale, `leg` which represents caption, `[2 3]` define two colors for differents graphs, the `xsetech` command define the position of graphs in the frame and the `xtitle` allow to change graph title and axe labels.







FTBAR seems to be the best adequation for execution times. We can notice that for CCR = 10, Failures Patterns adequation (FPTOL) are better than FTBAR one time of four.

Chapter 2

Making manual

2.1 required software

- compiler ocaml v3.07 or more
- patch camltk-hidden-state.patch
- Tcl/Tk v8.3 or more
- Syndex source code

2.2 Install and make a distrib of Syndex

- Change `COMMON_OBJS` of `Makefile` :

```
COMMON_OBJS=version.cmo symbolic.cmo coord.cmo port.cmo types.cmo \
  application.cmo algorithm.cmo architecture.cmo adequationtypes.cmo \
  parserexpressioninit.cmo lexerexpression.cmo parserexpression.cmo \
  sdx_lexer.cmo sdx_parser.cmo write.cmo \
  transformation.cmo camltk/progress_box.cmo \
  adequation_core.cmo fault_tolerance_adequation_core.cmo \
  latency_adequation.cmo fault_tolerance_adequation.cmo \
  fault_tolerance_with_fp.cmo read.cmo genexec.cmo
```

- Change directories in `Makefile.config.[SYSTEM]`.
- Verify the version of ocaml compiler (enter `ocamlc -v`)
- verify the existence of patch. We should observe `'+2'` on the right side of versus number of Objective Caml (exemple : Objective Caml version 3.07+2). Enter `patch -p1 otherlibs/labltk/Widgets.src < camltk-hidden-state.patch` if the patch is not present.
- Go in Syndex directory.
- Enter `make`
- Optional : to use Syndex command line , enter `make tui`

2.3 Links

- Objective Caml : `http://caml.inria.fr/ocaml/distrib.html`
- patch camltk-hidden-state.patch : `http://pauillac.inria.fr/camltk/`
- Tcl/Tk : `ftp://ftp.scriptics.com/pub/tcl/`

2.4 Modified files for this adequation

The interfaces (.mli) of the next files have been modified.

- `Makefile` to compile new files
- `ihm_ctk.ml` and `ihmcommon.ml` for HCI
- `fault_adequation_with_fp.ml` for adequation functions
- `fault_tolerance_adequation_core.ml` to view schedule specific functions
- `tui.ml` for command line
- `application.ml` to save failures patterns

2.5 Troubleshouting

If your system Linux is recognize as Windows during compilation, you must set the environnement variable `OSTYPE` to `linux` before compiling.

Chapter 3

User manual

This chapter is about "Fail Tolerance" in Syndex. For more informations of Syndex, go to url : `http://www-rocq.inria.fr/s`

3.1 Why Fail tolerance ?

Fail tolerance is very important for critical embedded systems ! The hardware components may be out of order. A controler of car brakes may become failed. We need some mecanisms to prevent the failure of one or more component of him.

3.2 What's a schedule which tolerate failures patterns ?

A failure pattern is a set of components (operator or media) which can become out of order at the same moment. A schedule which tolerate failures patterns is a schedule with replicas of tasks and data dependency which returns the correct result even if a failure pattern appear. Generally, it doesn't support two failures patterns in the same moment.

3.3 Create failures patterns

You should have create an main architecture graph. There is two way to obtain Failures Patterns frame :

- Click on `''Failures patterns Adequation''` in Adequation menu
- Press `F9`

All the components of the main architecture are in the left list of the frame. Select components you want to define as failed then click on the button `Create` to create a new failures pattern. A new item appear in the central list. His name contain all the components name of the failure pattern. There are only the max failures patterns, i.e., a failure pattern which is a subset of another of them is removed.

3.4 Save and restore failures patterns

All this proceed is available in `''Failures Patterns''` frame.

3.4.1 Save a failures pattern

Select one or more failures patterns then click on the button `Save F.P.` to save this failures patterns. A frame is opened to choose the file will contain the save then click on `Save`.

3.4.2 Restore failures patterns

Click on `Load F.P.`. A frame is opened to choose what file contains the failures patterns save. Click on `Open` to add this failures patterns. If failures patterns contain component which are not included in the main architecture, or they are subset of a failures pattern which exists, or they generate disjoint architecture then they are not added !

3.5 Failures patterns files

This files have .fp extension and are text files with this rules :

- a failure pattern by each line
- no empty line
- a failure pattern is represent by each name of fail component separated by exactly one space
- each line doesn't begin or end by a delimiter

Two failure patterns exemple : one line describe `proc1` and `proc2` as failed and another line with com3, operator6 and operator8 out of order :

```
proc1 proc2
com3 operator6 operator8
```

3.6 Generate and view simple schedule of each failures pattern

This type of schedule is a schedule corresponding to selected failures patterns. All this proceed is available in `‘‘Failures Patterns’’` frame.

3.6.1 Generate simple schedule for each failures pattern

Select one or more failures patterns in the central list then click on the button `Calc Simple Adequation`. Another schedule appear in the right list for each selected item.

3.6.2 View simple schedule for each failures pattern

Select one or more simple schedule which you want to view then click on the button `View Simple Schedule`. A frame is opened for each schedule.

3.7 Generate and view schedule which tolerate each failures pattern

This type of schedule tolerate any failure patterns of the list. All this proceed is available in `‘‘Failures Patterns’’` frame.

3.7.1 Generate a schedule which tolerate each failures pattern

Click on the button `Calc Global Adequation` to make adequation. The `global adequation` element represents a schedule which tolerate all the failures patterns where no fail appears during execution.

3.7.2 View schedule which tolerate each failures pattern

Select `global adequation` then click on the button `View Schedule`. A frame is opened which show this adequation.

3.8 Use Syndex command line

To use command line with syndex, enter `./syndex-tui.bin [input_file] [output_file] -fp [fp_file]`. `input_file` is the file which contains architecture and algorithme graphs, `fp_file` contains the failures patterns and `output_file` is the file which will contain the adequation.

3.9 Unsupported with this adequation

It supports operator fails (all types) point-to-point media fails. This adequation don't support multi-point medias (the bus) and disjoint architectures !

Chapter 4

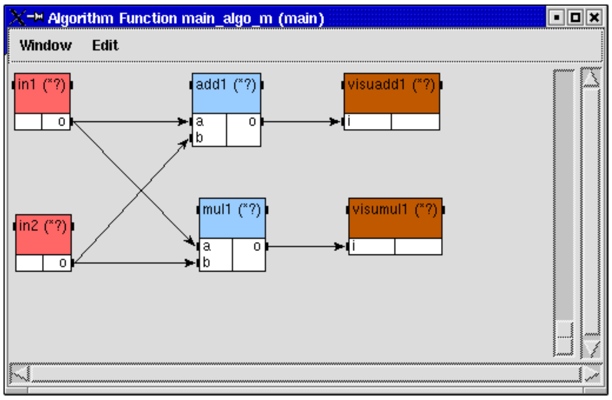
Tutorial

This chapter teach you how generate a fail tolerant schedule. it is necessary to know how create an algorithm graph and an architecture graph with Syndex. For more informations on this operations, you can read the Syndex documentation (`http://www-rocq.inria.fr/syndex/`).

4.1 Graphic use

4.1.1 Create algorithm

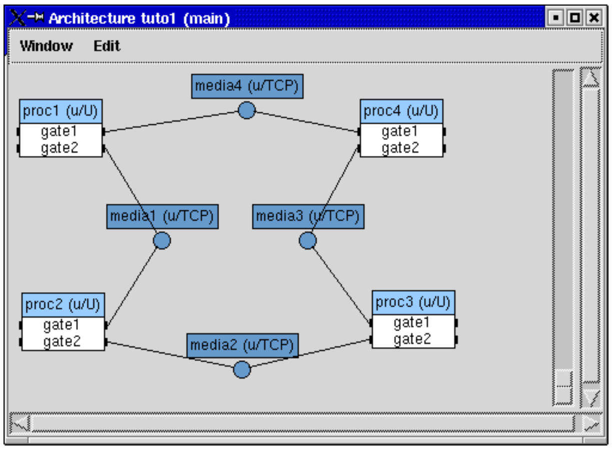
Use `int` librairy to create an algorithm `algo` with the operations `in1` (input), `in2` (input), `add1` (Arit_add), `mul1` (Arit_mul), `visuadd1` and `visumul1` (output). Finally, create the dependency between the ref which looklikes to next figure :



4.1.2 Create architecture

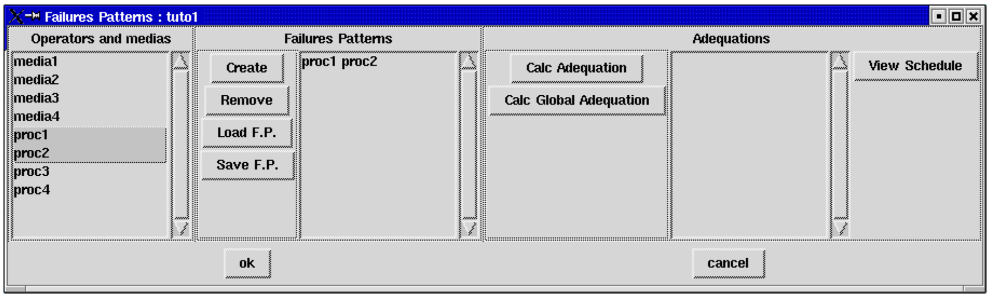
- Use library `U` to create four computing units (called operators in Syndex), `proc1` , `proc2`, `proc3`, and `proc4`.
- Use library `U` to create four medias, `media1` , `media2` , `media3` and `media4`.
- Create connections between operators and the medias.
- Define this architecture as main.

The architecture looks like next figure :



4.1.3 Create failure patterns

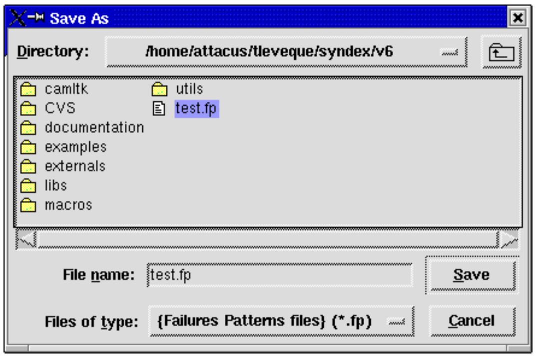
- Click in the menu `Adequation` on the item `Failures Patterns Adequation`.
- Select the operators `proc1` and `proc2` in the left list. Use `CTRL` to do multiple selection.



- Click on the button `Create` to add the new failure pattern. You can observe it in the middle list.
- Repeat previous operation for the two operators `proc3` and `proc4`.

4.1.4 Save failure patterns

Select item `proc1 proc2` in the failures patterns list (central list) then click on the button `Save F.P.`. A frame is opened to choose the save destination file, enter the name `test.fp`. Click on `Save` to save.

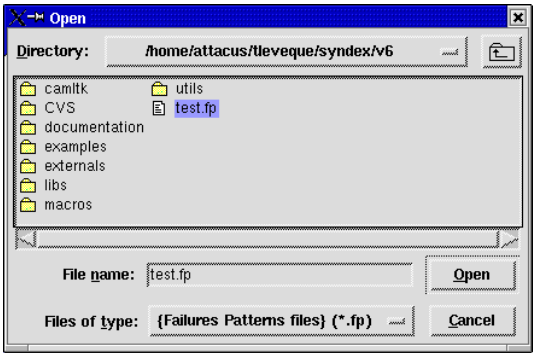


4.1.5 Remove a failure pattern

Select item proc1 proc2 in the failures patterns list (central list) then click on the button Remove.

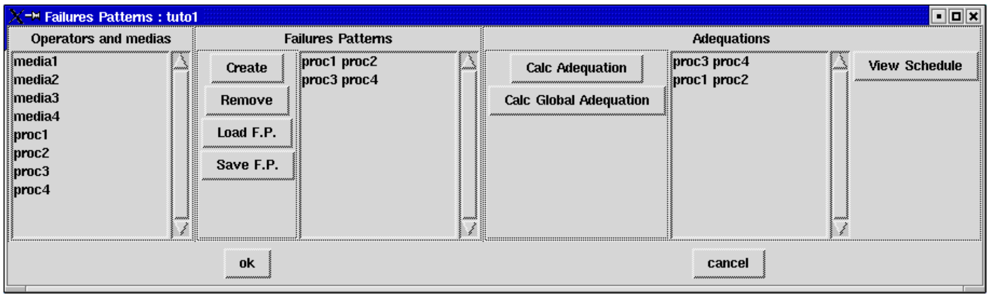
4.1.6 Load a failure pattern

Click on the button Load F.P.. A Frame is opened to choose the loaded file, choose the file test.fp. Click on Open to load the file. The failures pattern proc1 proc2 is added to failures patterns list.



4.1.7 Generate simple schedules

Select the item proc1 proc2 in the failures patterns list (central list) then click on the button Cacl Adequation. Another item with the same name is added to the schedule list (right list).

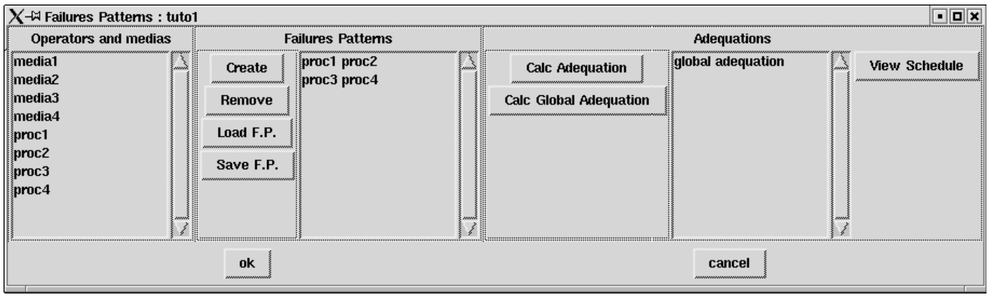


4.1.8 View simple schedules

Select the item proc1 proc2 in the schedule list (right list) then clicks on the button View Simple Schedule. A Frame corresponding to the schedule of selected item proc1 proc2 is opened.

4.1.9 Generate global schedules

Clicks on the button Calc Global Adequation. The set of schedules are calculated. Then a unique item global adequation is added to the schedule list (right list).



4.1.10 View global schedules

Select the element `global adequation` in schedule list then click on the button `View Schedule`. Another frame corresponding to this schedule is opened.

4.2 Command line use

After saving algorithme and architecture graphs in test.sdx, enter `./syndex-tui.bin test.sdx output.sdx -fp test.fp`. The file output.sdx contains your adequation.

Bibliography

[1] C. Dima, A. Girault, and Y. Sorel. Static fault-tolerant scheduling with "pseudo-topological" orders. In *Joint Conference on Formal Modelling and Analysis of Timed Systems and Formal Techniques in Real-Time and Fault Tolerant System, FORMATS-FTRTFT'04*, volume ???? of *LNCS*, Grenoble, France, September 2004. Springer-Verlag.

[2] Christian Rolland. *LaTeX : Par la pratique.* O'Reilly, 1999.

[3] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Développement d'applications avec OCAML.* O'Reilly, 2000.

[4] Andrew Tanenbaum and Maarten Van Steen. *Distributed System : principles and paradigms.* Prentice Hall, 2002.