Deadlock Checker User Guide and Technical Manual

Jeremy Martin
Department of Mathematics, Statistics, and Computer Science
University of Buckingham, MK18 1EG, U.K.

December 19, 1995

1 Introduction

This document contains information about how to install and run the program 'Deadlock Checker', described in [Martin 1995]. It also contains a listing of the ML source code, and some technical details that would be of use to future developers.

The program will only run on a Sun4 or Sun5 computer, with version 1.4 of FDR installed.

This document is intended to serve as a companion to [Martin 1995], and includes no information about the algorithms employed by Deadlock Checker, nor any information about the design rules that it checks. A working knowledge of FDR, the CSP refinement checking program, is assumed [Formal Systems 1993].

2 Installing and Running Deadlock Checker

The source code is split into two files; `compile.ml`, and `check.ml`. These contain standard ML code. The former, which controls network compilation, can only be used on top of FDR. The latter, which performs deadlock and livelock analysis, should work with any standard ML compiler or interpreter.

To run the network compiler first start up FDR

`jupiter%` *fdr*

Note that `jupiter%` is just the unix prompt on one of the machines that was used to develop Deadlock Checker. Don't type that.

Next from within fdr, load the file `compile.ml` `fdr>` *use "compile.ml";*

Once the program has been loaded, you may compile the Dining Philosophers network, `phils.net`, as follows

`fdr>` *compile "phils.csp" "phils.net";*

The compilation should now take place. A certain amount of information will appear on the screen to inform you whether or not this is running smoothly.

Unfortunately it is not possible to load file `check.ml` as well at this stage without running into memory problems. So, once the compilation is complete, you should exit from FDR as follows.

`fdr>` *quit ();*

Now you can proceed with the deadlock analysis. First restart fdr.

`jupiter%` *fdr*

From within FDR, load program `check.ml`

`fdr>` *use "compile.ml";*

Normally you will be placed straight into an interactive deadlock analysis session. The following prompt will now appear

```
Welcome to Deadlock Checker
Command (h for help, q to quit):
```

Typing `h` reveals the following menu

```
Command (h for help, q to quit):h
 h              - help: display this menu
 l <file>       - load network file
 n              - display list of networks in memory
 s <name>       - select network
 c              - display currently selected network
 p              - display list of processes in current network
 d              - decompose network deadlock analysis
 v              - check for acyclic deadlock-freedom (SDD algorithm)
 x              - check for acyclic deadlock-freedom (CSDD algorithm)
 f              - check for acyclic deadlock-freedom (FSDD algorithm)
```

```
 o             - check for deadlock in cyclic-po network
 w             - check for deadlock in client-server network
 a             - check for resource allocation protocol
 r             - restrict network to its vocabulary
 t             - test for livelock-freedom (Roscoe's rule)
```

Note that FSDD stands for "Flashing State Dependence Digraph", which is referred to on page 101 of [Martin 1995].

To load the recently compiled Dining Philosophers network, type

`Command (h for help, q to quit):`*l phils.net*

Now check it for adherence to the Resource Allocation Protocol

`Command (h for help, q to quit):`*a*

```
Network phils.net is busy
Network phils.net is triple-disjoint
Process FORK(4) acts as a resource
Process FORK(3) acts as a resource
Process FORK(2) acts as a resource
Process FORK(1) acts as a resource
Process FORK(0) acts as a resource
Process PHIL(4) is not a resource
User process PHIL(4) obeys resource allocation protocol
User process PHIL(3) obeys resource allocation protocol
User process PHIL(2) obeys resource allocation protocol
User process PHIL(1) obeys resource allocation protocol
User process PHIL(0) claims resource FORK(4) still holding FORK(0)
```

Quit the interactive session as follows

`Command (h for help, q to quit):`*q*

If you need to restart the interactive session, type

`fdr>` *teletype();*

Otherwise quit FDR.

`fdr>` *quit ();*

3 Format of a Compiled Network

The application of program `compile` to a machine readable CSP network results in the creation of a file with the following grammar, to be used by program `check`.

$$
\begin{array}{rcl}
\mathit{Network} & = & \mathit{Encoding} \quad \mathit{Processes} \\
\mathit{Endcoding} & = & \texttt{ENCODING} \quad \hookleftarrow \\
 & & \mathit{Mappings} \\
 & & \texttt{END ENDCODING} \quad \hookleftarrow \\
\mathit{Mappings} & = & \mathit{Number} \quad \mathit{Event} \quad \hookleftarrow \\
 & \mid & \mathit{Number} \quad \mathit{Event} \quad \hookleftarrow \\
 & & \mathit{Mappings} \\
\mathit{Processes} & = & \mathit{Process} \\
 & \mid & \mathit{Process} \quad \mathit{Processes} \\
\mathit{Process} & = & \texttt{PROCESS} \quad \hookleftarrow \\
 & & \mathit{Name} \quad \hookleftarrow \\
 & & \texttt{ALPHABET} \quad \hookleftarrow \\
 & & \mathit{Numbers} \quad \hookleftarrow \\
 & & \mathit{States} \\
 & & \texttt{END PROCESS} \quad \hookleftarrow \\
\mathit{Numbers} & = & \mathit{Number} \mid \mathit{Number} \quad \mathit{Numbers} \\
\mathit{States} & = & \mathit{State} \mid \mathit{State} \quad \mathit{States} \\
\mathit{State} & = & \texttt{STATE} \quad \hookleftarrow \\
 & & \mathit{Number} \quad \hookleftarrow \\
 & & \mathit{ListofRefusals} \\
 & & \mathit{ListofTransitions} \\
 & & \texttt{ENDSTATE} \quad \hookleftarrow \\
\mathit{ListofRefusals} & = & \texttt{REFUSALS} \quad \hookleftarrow\hookleftarrow \\
 & \mid & \texttt{REFUSALS} \quad \hookleftarrow \\
 & & \mathit{Refusals} \\
\mathit{ListofTranstions} & = & \texttt{TRANSITIONS} \quad \hookleftarrow \\
 & \mid & \texttt{TRANSITIONS} \quad \hookleftarrow \\
 & = & \mathit{Transitions} \\
\mathit{Refusals} & = & \mathit{Numbers} \quad \hookleftarrow \\
 & \mid & \mathit{Numbers} \quad \hookleftarrow \\
 & & \mathit{Refusals}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{Transitions} & = & \mathit{Number} \quad \mathit{Number} \quad \hookleftarrow \\
& | & \mathit{Number} \quad \mathit{Number} \quad \hookleftarrow \\
& & \mathit{Transitions}
\end{array}
$$

Where *Number* ranges over integers, *Name* ranges over process names, and *Event* ranges over event names. The newline character is represented as $\hookleftarrow$.

As an example we include part of the compiled form of the Dining Philosophers network

```
ENCODING
0 eats.0
1 eats.1
2 eats.2
3 eats.3
4 eats.4
5 takes.0.0
6 takes.0.1
7 takes.0.2
8 takes.0.3
9 takes.0.4
... etc
30 drops.0.0
31 drops.0.1
32 drops.0.2
33 drops.0.3
34 drops.0.4
35 drops.1.0
...etc
998 tick
999 tau
END ENCODING
PROCESS
PHIL(0)
ALPHABET
0 5 9 30 34
STATE
0
REFUSALS
9 0 34 30
TRANSITIONS
5 1
ENDSTATE
STATE
1
REFUSALS
5 0 34 30
TRANSITIONS
9 2
ENDSTATE
```

```
STATE
2
REFUSALS
5 9 34 30
TRANSITIONS
0 3
ENDSTATE
STATE
3
REFUSALS
5 9 0 30
TRANSITIONS
34 4
ENDSTATE
STATE
4
REFUSALS
5 9 0 34
TRANSITIONS
30 0
ENDSTATE
END PROCESS
PROCESS
PHIL(1)
... etc
```

References

[Martin 1995] J. M. R. Martin *The Design and Construction of Deadlock-Free Concurrent Systems*, Internal Report, University of Buckingham (Department of Mathematics, Statisitics and Computer Science) 1995.

[Formal Systems 1993] *FDR User Manual and Tutorial* Formal Systems (Europe) Ltd. 3 Alfred Street, Oxford OX1 4EH

A Program `compile.ml`

```
(*                        DEADLOCK CHECKER

       A suite of tools for deadlock analysis of CSP networks.
    ( Extra code for FDR to produce network compilation reports )

               Author:  Jeremy Martin, January 1995                *)

(*-----------------------------------------------------------------|
|  This portion of code supplied by Dave Jackson, Formal Systems   |
|----------------------------------------------------------------*)

(* This code is copyright (c) 1995 Formal Systems (Europe) Ltd. It may not be
copied, modified, or distributed in any form without prior written permission
of the copyright owners. It is supplied free of charge, and consequently
```

```
is distributed "as is" without warranty of any kind, either expressed or
implied, including, but not limited to, the implied warranties of
merchantability and fitness for a particular purpose.
*)

fun normalize fname p =
   let
val sp = simplify p;
val _  = alphadump sp (tempfile "spec.ab");
val _  = dump sp (tempfile "spec.imp");
val _  = System.system (FILES.command((!normal_program)^" "^
      (!normal_flags)^
                              (tempfile "spec.ab ")^
                              (tempfile "spec.imp >")^fname));
    in
()
end;

(*------------------------------------------------------------------|
|  Code for producing reports                                       |
|------------------------------------------------------------------*)

fun input_line is =
let
  fun getstring s =
  case input(is, 1) of
    "" => s
  | "\n" => s
  | c => getstring(s^c)
in
  getstring ""
end;

fun nat_to_string (n) =
let
  fun split n =
  if n < 10  then
    [n]
  else
    ((n mod 10)::(split (n div 10)))
in
  implode (map (fn i => chr(i + ord"0")) (rev (split n)))
end;

(* Separate a sequence of process definitions *)

fun separate_terms ([],_,_,read) = if read <> "" then [read] else []
  | separate_terms (c::not_read,n,n',read) =
if c = "(" then
  separate_terms (not_read,n+1,n',read^c)
else if c = ")" then
  separate_terms (not_read,n-1,n',read^c)
else if c = "{" then
  separate_terms (not_read,n,n'+1,read^c)
else if c = "}" then
  separate_terms (not_read,n,n'-1,read^c)
else if n = 0 andalso n' = 0 andalso c = "," then
  (if read <> "" then [read] else [])@separate_terms(not_read,n,n',"")
else
  separate_terms(not_read,n,n',read^c);

fun read_network i file =
let
  fun no_blanks xs = filter (fn x => x<> " ") xs
  fun write_string s =
```

```
  (System.system ("echo '"^s^"' >> "^file);())
  fun write_process p =
  let
    val _ = System.system ("echo \"PROCESS\" >> "^file)
    val _ = System.system ("echo \""^p^"\" >> "^file)
    val _ = normalize (tempfile "temp") (process p)
    val _ = System.system ("cat "^(tempfile "spec.ab")^" >> "^
     (tempfile "net.ab"))
    val _ = System.system ("cat "^(tempfile "temp")^" >> "^file)
    val _ = System.system ("echo \"END PROCESS\" >> "^file)
  in
     ()
  end
in
  if (end_of_stream i)then
     ()
  else
     (case (explode (input_line i)) of
       ("-"::"-"::"+"::ys) => (map write_process (separate_terms
       (no_blanks(ys),0,0,""));())
     | ("-"::"-"::"-"::ys) => write_string ("---"^implode(ys))
     | _ => ();
    read_network i file)
end;

fun get_alphabet file =
let
  val alph = ref ""
  val i = ref 0
in
  while !i < 1001 do
     (alph := (print_value (DecodeEvent (!i)) handle CodingError(s) => "");
    if (!alph <> "") then
       (System.system ("echo \""^(nat_to_string (!i))^
       " "^(!alph)^"\" >> "^file);())
    else
       ();
    i := !i + 1)
end

(* Takes a file containing standard FDR process definitions, and also
   comments beginning --+ followed by instances of processes which
   make up our network. Creates an output file containing the FDR event
   encoding, plus the normalised transition system for each process in
   the network. Comments beginning --- are copied to the output file
   in context *)

fun compile file1 file2 =
let
  val i = open_out file2
  val _ = close_out i (* make sure file2 is empty *)
  val i = open_out (tempfile "net.ab")
  val _ = close_out i (* make sure this file exists and is empty *)
  val _ = ParseFile file1
  val i = open_in file1
  val _ = System.system ("echo \"ENCODING\" >> "^file2)
  val _ = get_alphabet file2
  val _ = System.system ("echo \"END ENCODING\" >> "^file2)
  val _ = read_network i file2
  val _ = close_in i
in
  ()
end;
```

B Program check.ml

```
(*                      DEADLOCK CHECKER

       A suite of tools for deadlock analysis of CSP networks.

               Author:  Jeremy Martin, January 1995

                                                                      *)

(*--------------------------------------------------------------------|
|  Standard definitions                                               |
|--------------------------------------------------------------------*)

(* Function composition *)
infix o
fun (f o g) x = f (g x);

exception Hd;
fun hd (x::_) = x
  | hd [] = raise Hd;

exception Tl;
fun tl (_::xs) = xs
  | tl [] = raise Tl;

fun left (x,_) = x;

fun right (_,x) = x;

fun flat [] = []
  | flat(l::ls) = l @ flat ls;

(* Split a list of pairs into two separate lists *)
fun split [] = ([],[])
  | split((x,y)::pairs) =
let
  val (xs,ys) = split pairs
in
  (x::xs, y::ys)
end;

(* Combine two lists into a list of pairs *)
fun combine ([],[]) = []
  | combine (x::xs, y::ys) = (x,y)::combine(xs,ys);

(* Remove those items of a list which do not satisy a predicate *)
fun filter pred [] = []
  | filter pred (x::xs) =
if pred(x) then x :: filter pred xs else filter pred xs;

(* Take the first element from a list that satisfies a predicate, if
   such an element exists. A list of length one or zero is returned *)
fun get pred [] = []
  | get pred (x::xs) =
if pred(x) then [x] else get pred xs;

(* Count the number of occurences of an item in a list *)
fun cnt (x,[]) = 0
  | cnt (x,y::ys) = if x=y then (1+(cnt(x,ys))) else cnt(x,ys);

local
  fun length1 (n, [ ])  = n
    | length1 (n, x::l) = length1 (n+1, l)
```

```
in
  fun length l = length1 (0,l)
end;

(* Truncate a list after the first n elements *)
fun take (n, []) = []
  | take (n, x::xs) =  if n>0 then x::take(n-1,xs)
else  [];

(* Remove the first n elements from a list *)
fun drop (_, []) = []
  | drop (n, x::xs) = if n>0 then drop (n-1, xs)
else x::xs;

(* Take the nth element from a list *)
exception Nth;
fun nth (n, []) = raise Nth
  | nth (n, x::xs) = if n=1 then x else if n>0 then nth(n-1,xs)
else raise Nth;

(* Check whether there exists an element in a list, satisfying a supplied
   predicate *)
fun exists pred [] = false
  | exists pred (x::xs) = (pred x) orelse exists pred xs;

(* Check whether all elements in a list satisfy a supplied predicate *)
fun forall pred []      = true
  | forall pred (x::xs) = (pred x) andalso forall pred xs;

fun nat_to_string (n) =
let
  fun split n =
  if n < 10 then
    [n]
  else
    ((n mod 10)::(split (n div 10)))
in
  implode (map (fn i => chr(i + ord"0")) (rev (split n)))
end;

exception Min;
fun min ([]:int list) = raise Min
  | min [x] = x
  | min (x::y::xs) = if x<y then min(x::xs) else min(y::xs);

(*---------------------------------------------------------------------|
|  Set operations                                                      |
|--------------------------------------------------------------------*)

infix member;
fun x member [] = false
  | x member (y::l) = (x=y) orelse (x member l);

fun newmember(x,xs) = if x member xs then xs else x::xs;

fun setof [] = []
  | setof (x::xs) = newmember(x, setof xs);

infix union;
fun [] union ys = ys
  | (x::xs) union ys = newmember(x, xs union ys);

infix intersect;
fun [] intersect ys = []
  | (x::xs) intersect ys = if x member ys then x::(xs intersect ys)
```

```
else (xs intersect ys);

infix minus_set;
fun [] minus_set ys = []
  | (x::xs) minus_set ys = if x member ys then xs minus_set ys
else x::(xs minus_set ys);

infix subset;
fun [] subset ys = true
  | (x::xs) subset ys = (x member ys) andalso (xs subset ys);

infix equals_set;
fun xs equals_set ys = (xs subset ys) andalso (ys subset xs);

infix disjoint;
fun xs disjoint ys = (xs intersect ys) = [];

infix member_setofsets;
fun x member_setofsets [] = false
  | x member_setofsets (y::l) = (x equals_set y) orelse
(x member_setofsets l);

infix subset_setofsets;
fun [] subset_setofsets ys = true
  | (x::xs) subset_setofsets ys = (x member_setofsets ys) andalso
(xs subset_setofsets ys);

infix equals_setofsets;
fun xs equals_setofsets ys = (xs subset_setofsets ys) andalso
(ys subset_setofsets xs);

(* Select the maximal elements of a set of sets *)

fun maximal_sets s=
filter (fn x=> not (exists (fn y => x subset y) (s minus_set [x]))) s;

(*-------------------------------------------------------------------|
|  Graph operations                                                  |
|-------------------------------------------------------------------*)

(* Topological sort of a digraph - detects presence of circuits *)

local
  fun nexts (a, []) = []
    | nexts (a, (x,y)::pairs) = if a=x then  y::nexts(a,pairs)
  else nexts(a,pairs);
in
  fun find_circuit digraph =
  let
    fun upto ([], x) = []
      | upto (p::path, x) = if x=p then [p] else (p::upto(path,x))
    fun sort ([], path, visited) = (visited,[])
      | sort (x::xs, path, visited) =
    if x member path then
      (visited, rev(x::upto(path, x)))
    else if x member visited then
      sort(xs, path, visited)
    else
      let
        val (v,c) = sort(nexts(x,digraph),x::path,x::visited)
      in
        if c <> [] then
          (v,c)
        else
          sort(xs, path, v)
```

```
      end
    val (xs,_) = split digraph
  in
    sort(xs, [], [])
  end
end;

fun circuit_free digraph = right(find_circuit digraph)=[];

(* Calculate the minimal elements of a set under a relation *)
fun minimal_elements (xs,digraph) =
let
  fun non_minimal_elements (xs, []) = []
    | non_minimal_elements (xs, (y,z)::g) =
  if (z member xs) then
  y::non_minimal_elements (xs, g)
  else non_minimal_elements (xs, g)
in
  xs minus_set non_minimal_elements(xs,digraph)
end;

(* Calculate the nodes of a digraph which lie on a circuit, or from
   which there is a path to a circuit, using M. Goldsmith's algorithm *)
local
  fun successors (y,digraph) = map right (filter(fn (x,_) => x=y)
  digraph)
  fun check_path([],_,no_ends,ends,g)=(no_ends,ends)
    | check_path(x::path,[]::pendinglist,no_ends,ends,g)=
  check_path(path,pendinglist,no_ends,ends union [x],g)
    | check_path(x::path,(y::pending)::pendinglist,no_ends,ends,g)=
  if y member no_ends orelse y member (x::path) then
    (no_ends union (x::path),ends)
  else if y member ends then
    check_path(x::path,pending::pendinglist,no_ends,ends,g)
  else
    check_path(y::x::path,successors(y,g)::pending::pendinglist,no_ends,
    ends,g)
  fun find_nodes_without_end([],no_ends,ends,g)=no_ends
    | find_nodes_without_end(n::pending,no_ends,ends,g)=
  if n member ends orelse n member no_ends then
    find_nodes_without_end(pending,no_ends,ends,g)
  else
    let
      val (no_ends',ends')=
      check_path([n],[successors(n,g)],no_ends,ends,g)
    in
      find_nodes_without_end(pending,no_ends',ends',g)
    end
in
  fun nodes_without_end digraph =
  let
    val nodes_with_successors = setof (map left digraph)
  in
    find_nodes_without_end(nodes_with_successors,[],[],digraph)
  end
end;

(* Determine whether two nodes are connected in an undirected graph.
   This function also returns a list of certain edges, which have been
   found to lie on a circuit during the analysis *)
local
  fun single_pass(visited,dest,[],edges,extra)=
  (false,visited,edges,extra)
    | single_pass(visited,dest,(x,y)::pending,edges,extra)=
  if x member visited then
```

```
    if y=dest then
      (true, visited, edges minus_set [(x,y)],extra union [(x,y)])
    else if y member visited then
      single_pass(visited,dest,pending,edges minus_set [(x,y)],extra
      union [(x,y)])
    else
      single_pass(visited union [y],dest,pending,edges minus_set [(x,y)],
      extra)
  else if y member visited then
    if x=dest then
      (true,visited,edges minus_set [(x,y)],extra union [(x,y)])
    else
      single_pass(visited union [x],dest,pending,edges minus_set [(x,y)],
      extra)
  else
    single_pass(visited,dest,pending,edges,extra)
  fun find_route(visited,dest,[],extra)=(true,extra)
    | find_route(visited,dest,pending,extra)=
  let
    val (connects,visited',pending',extra')=
    single_pass(visited,dest,pending,pending,extra)
  in
    if connects then
      (false, extra')
    else
      if(visited' equals_set visited)then
        (true,extra')
      else
        find_route(visited',dest,pending',extra')
  end
in
  fun connected(x,y,graph)=find_route([x],y,graph,[])
end;

(* Determine whether two nodes are connected in a digraph, if so
   return the path  (Note:x<>y) *)
local
  fun nexts (a, []) = []
    | nexts (a, (x,y)::pairs) = if a=x then  y::nexts(a,pairs)
  else nexts(a,pairs);
in
  fun get_path(x,y,digraph) =
  let
    fun sort ([], path, visited) = ([],visited)
      | sort (z::zs, path, visited) =
    if z = y then
      (rev(y::path), visited)
    else if (z member path orelse z member visited) then
      sort (zs, path, visited)
    else
      let
        val (p,v) = sort (nexts(z,digraph),z::path,z::visited)
      in
        if p <> [] then
          (p,v)
        else
          sort(zs,path, v)
      end
  in
    sort ([x],[],[])
  end
end;

(* Determine the disconnecting edges of an undirected graph. The algorithm
   used here is different from the one described in my thesis. It is less
```

```
   efficient but easier to implement for this prototype version of the
   software *)
local
  fun find_disconnecting_edges([],g)=[]
    | find_disconnecting_edges((x,y)::pending,g)=
  let
    val (disconnects,extra)=connected(x,y,g minus_set [(x,y)])
  in
    if disconnects then
      (x,y)::find_disconnecting_edges(pending minus_set extra,g)
    else
      find_disconnecting_edges(pending minus_set extra,g)
  end
in
  fun disconnecting_edges graph = find_disconnecting_edges(graph,graph)
end;

(* Determine the connected components of an undirected graph. The algorithm
   used here is different from the one described in my thesis. It is less
   efficient but easier to implement for this prototype version of the
   software *)
local
  fun single_pass(e,n,[],unused)=(e,n,unused)
    | single_pass(e,n,(x,y)::pending,unused)=
  if x member n orelse y member n then
    single_pass( (x,y)::e, n union [x,y], pending, unused)
  else
    single_pass( e, n, pending, (x,y)::unused)
  fun form_component (e,n,[]) = (e,n,[])
    | form_component (e,n,m) =
  let
    val (e',n',m') = single_pass (e, n, m, [])
  in
    if n=n' then
      (e',n',m')
    else
      form_component (e',n', m')
  end
  fun form_disjoint_components([],components)=components
    | form_disjoint_components((x,y)::remainder,components)=
  let
    val(edges,new_component,new_remainder)=
    form_component([(x,y)],[x,y],remainder)
  in
    form_disjoint_components(new_remainder,new_component::components)
  end
in
  fun disjoint_components graph = form_disjoint_components (graph,[])
end;

(*-----------------------------------------------------------------|
|  Sorting                                                         |
|-----------------------------------------------------------------*)

(* Top-down merge sort of integers *)
fun merge([],ys) = ys : int list
  | merge(xs,[]) = xs
  | merge(x::xs, y::ys) = if x<=y then x::merge(xs, y::ys)
else y::merge(x::xs, ys);

fun tmergesort [] = []
  | tmergesort [x] = [x]
  | tmergesort xs =
let
  val k = length xs div 2
```

```
in
  merge(tmergesort (take(k,xs)), tmergesort (drop(k,xs)))
end;

(*-------------------------------------------------------------------|
|  Equivalence classes and partitions                                |
|-------------------------------------------------------------------*)

(* Partition a set with an equivalence relation *)
local
  fun add (x, [], equivalent) = [[x]]
    | add (x, class::classes,equivalent)=
  if equivalent(x, hd(class))then
    (x::class)::classes
  else
    class::add(x,classes,equivalent)
in
  fun partition ([], equivalent)=[]
    | partition (x::remainder, equivalent)=
  add(x,partition(remainder,equivalent),equivalent)
end;

(* Determine the fixed point of successive applications of a function
   to a partition. If no such point exists this function will diverge *)
fun fixed_point f part =
let
  val part' = f part
in
  if part equals_setofsets part' then
    part
  else
    fixed_point f part'
end;

(*-------------------------------------------------------------------|
|  Data types, parameters and global variables                       |
|-------------------------------------------------------------------*)

(* Data type to represent normal form states: note that refusal sets
   are currently used instead of acceptance sets. *)
type state = {number:int, transitions:(int*int) list,
refusals:int list list };

type process = {number:int, alphabet:int list, initial_state:int,
states:state list, data:string};

(* Initialisations of global constants *)
val divergence_state = ~1;

val tau_event = 999;

(* Initialisation of global variables *)
(* Correspondence between integer keys and event names *)
val event_encoding = ref (ref ([]:(int*string) list));

(* Correspondence between integer keys and process names *)
val process_encoding = ref (ref ([]:(int*string) list));

val process_buffer = ref ([]:(int*string) list);

(* Number of processes read *)
val process_num = ref 0;

(* Currently selected network *)
val user_net = ref (ref ([]:process ref list));
```

```
(* Name of currently selected network *)
val net_name = ref ("":string);

(* Stack of networks. Format of each record is
   Network name : List of processes : Event name encoding : Process name
   encoding : Deadlock status (Deadlocks/Deadlock-free/Unknown) *)
val net_list = ref ([]:(string*(process ref list ref)*
((int*string) list ref)*((int*string) list ref)*string) list);

(* Number of networks on the stack *)
val net_num = ref ~1;

(* Network dependence list. Format of each record is:
   Network A : [Network A_1, Network A_2, ... Network A_n], meaning that
   if all the networks A_1, ... A_n are proven deadlock-free then it
   follows that network A is also deadlock-free. *)
val net_dependence = ref ([]:(string*(string list)) list);

(* Log file *)
val log = ref (open_out "check.log");

(* Close the log file. Reopen when we need it *)
val _ = close_out (!log);

(*-------------------------------------------------------------------|
|  Parse normal-form process definition file                         |
|-------------------------------------------------------------------*)
(* The following code is used to load in a normal-form process
   definition file, such as is produced by the program "compile.ml",
   which sits on top of FDR *)
fun string_to_int (s:string) =
let
  fun string_to_int1 ("-"::xs) = ~(string_to_int1 xs)
    | string_to_int1 xs =
  let
    fun number s = ord(s) - ord("0")
    fun string_to_int2 [] = 0
      | string_to_int2 (x::xs) =
    (number x) + (10*(string_to_int2 xs))
  in
    (string_to_int2 (rev xs))
  end
in
  string_to_int1 (explode s)
end;

(* This function is used in conjunction with 'explode' to separate
   an input line into words *)
fun make_list (xs:string list) =
let
  fun form_words(ws,w,[])=
  if w <> "" then
    ws@[w]
  else
    ws
    | form_words(ws,w,x::xs)=
  if x = " " then
    form_words(ws@[w],"",xs)
  else
    form_words(ws,w^x,xs)
in
  form_words([],"",xs)
end;
```

```
(* Read a line from a file *)
fun input_line is =
let
  fun getstring s =
  case input(is,1) of
    "" => s
  | "\n" => s
  | c => getstring(s^c)
in
  getstring ""
end;

fun ignore_line i = (input_line i;());

fun read_int_list i =
(map string_to_int (make_list (explode (input_line i))));

fun read_refusals i =
if lookahead(i) = "T" then
  (ignore_line i;[])
else
  (read_int_list i)::(read_refusals i);

fun read_transitions i =
if lookahead(i) = "E" then
  (ignore_line i;[])
else
  ((fn [x,y] => (x,y)) (read_int_list i))::(read_transitions i);

fun read_states i =
if (lookahead(i)="E")then
  (ignore_line i;[])
else
  let
    val _ = ignore_line i
    val n = hd(read_int_list i)
    val _ = ignore_line i
    val r = read_refusals i
    val t = read_transitions i
  in
    ({number=n,refusals=r,transitions=t}:state)::(read_states i)
  end;

fun read_data i = (* read any comment lines following a process definition *)
let
  fun no_blanks s = filter (fn x => x<>" ") (explode s)
in
  if lookahead(i) <> "-" then
    ""
  else (* Remove initial "---" and any space characters *)
    (implode(tl(tl(tl(no_blanks(input_line i))))))^(read_data i)
end;

fun read_nf i =
let
  val _ = ignore_line i
  val process_name = input_line i
  val _ = ignore_line i
  val a = read_int_list i
  val s = read_states i
  val d = read_data i
in
  (process_num := !process_num + 1;
  process_buffer := (!process_num,process_name)::(!process_buffer);
  ref ({number=(!process_num), alphabet=a, states=s, initial_state=
```

```
  (#number (hd s)),data=d}:process))
end;

fun read_network i net  =
if lookahead(i) = "-" then
  (ignore_line i; read_network i net) (* ignore initial comments *)
else if lookahead(i) = "P" then
  read_network i (net@[(read_nf i)]) (* read next process *)
else
  net (* finished *);

fun read_encoding xs i =
let
  val line = input_line i
in
  if(line = "END ENCODING")then
    xs
  else
    let
      val ys = make_list(explode(line))
    in
      read_encoding ((string_to_int(hd ys),hd(tl ys))::xs) i
    end
end;

fun load_network file =
let
  val i = open_in file
  val _ = ignore_line i
  val encoding_ref = ref (read_encoding [] i)
  val _ = process_buffer := []
  val net = read_network i []
  val process_ref = ref (!process_buffer)
  val net_ref = ref net
  val _ = close_in i
in
  event_encoding := encoding_ref;
  process_encoding := process_ref;
  user_net := net_ref;
  net_num := !net_num + 1;
  net_name := file;
  net_list := (!net_list)@[(!net_name, net_ref, encoding_ref,
  process_ref, "unresolved")];
  ()
end;

(*----------------------------------------------------------------------|
|  Routines for communication with the user                             |
|----------------------------------------------------------------------*)

fun print (s:string) = (output (std_out, s);output (!log, s));

fun list_networks () =
(map (fn x => print( (#1 (x))^" ("^(#5 (x))^")\n")) (!net_list);());

fun process_name (n:int) =
right( hd (get (fn (n',_) => n=n') (!(!process_encoding))));

fun pname (p:process ref) = process_name (#number (!p));

fun list_processes () =
(map (fn p => print( (pname p)^"\n" )) (!(!user_net));());

fun current_network () = print ((!net_name)^"\n");
```

```
fun event (n:int) =
right( hd (get (fn (n',_) => n=n') (!(!event_encoding))));

fun event_list [] = ""
  | event_list (x::[]) = (event x)
  | event_list (x::xs) = (event x)^","^(event_list xs);

fun event_list_list [] = ""
  | event_list_list (x::[]) = "<"^(event_list x)^">"
  | event_list_list (x::xs) = "<"^(event_list x)^">,\n "^
(event_list_list xs);

fun string_set [] = ""
  | string_set (s::[]) = s
  | string_set (s::ss) = s^",\n "^(string_set ss);

fun event_graph [] = ""
  | event_graph ((x,y)::[]) = "("^(event x)^" > "^(event y)^")\n"
  | event_graph ((x,y)::g) = "("^(event x)^" > "^(event y)^"),\n"^
event_graph(g);

(*-------------------------------------------------------------------|
|  Routines for managing stored networks and derived information      |
|-------------------------------------------------------------------*)

(* Mark network "b" as being deadlock-free. *)
fun deadlocks b =
let
  val s = if b then
    (print("Network "^(!net_name) ^" deadlocks\n");"deadlocks")
  else
    (print("Network "^(!net_name) ^" is deadlock-free\n");"deadlock-free")
in
  net_list := map (fn (a,b,c,d,e) => if a= (!net_name) then (a,b,c,d,s) else
  (a,b,c,d,e))
  (!net_list)
end;

fun select_network (name:string) =
let
  val x = get (fn x => (#1 x) = name) (!net_list)
in
  if x = [] then
    (print ("No such network\n");())
  else
    (user_net := (#2 (hd(x)));
    event_encoding := (#3 (hd(x)));
    process_encoding := (#4 (hd(x)));
    net_name := (#1 (hd(x)));
    ())
end;

(* Check for any knock-on effect of a particular process being found to
   be deadlock-free. Global variable "net_dependence", maintains a
   structure of deadlock dependencies *)
fun update_network_status () =
(map
(fn (x,ys) =>
if (forall
  (fn y => (#5 (hd(get (fn (a,_,_,_,_) => a=y ) (!net_list))))
  = "deadlock-free") ys) then
    (net_list := (map (fn (a,b,c,d,e) => if a=x then
      (if e<>"deadlock-free" then
        print("Network "^a^" is deadlock_free\n")
       else ();(a,b,c,d,"deadlock-free"))
```

```
      else (a,b,c,d,e)) (!net_list));())
  else ())
(!net_dependence);());

fun make_subnetwork (xs:int list) name =
let
  val net = filter (fn p => (#number (!p)) member xs) (!(!user_net))
in
  (net_num := !net_num + 1;
  net_list := (!net_list)@[(name, ref net, !event_encoding, !process_encoding,
  "unresolved")];
  ())
end;

(* Add a list of subnetworks to the network stack (each described by a
   list of process ids) Also add a record to global list net_dependence
   to state that if all these subnetworks are proven deadlock-free then
   so must be the network which was selected at the time of creation of
   these subnetworks *)
fun make_subnetworks (ys:int list list) =
let
  val dependence =
  (!net_name,
  map (fn xs =>
  let
    val name = !net_name^"_"^nat_to_string(!net_num)
  in
    make_subnetwork xs name; name
  end
  ) ys)
in
  net_dependence := (dependence::(!net_dependence));
  ()
end;

(* Add a single network to the network to the network stack. If this is
   subsequently proven deadlock-free then so is the network which was
   selected at the time of its creation *)
fun make_derived_net (net: process ref list) =
let
  val net_ref = ref net
in
  net_num := !net_num + 1;
  net_dependence := (!net_name,[(!net_name)^"_"^nat_to_string(!net_num)])::
  (!net_dependence);
  net_name := (!net_name)^"_"^nat_to_string(!net_num);
  user_net := net_ref;
  net_list := (!net_list)@[(!net_name,net_ref, !event_encoding,
  !process_encoding, "unresolved")];
  ()
end;

(*------------------------------------------------------------------|
|  Basic operations on networks, processes, and their states         |
|-----------------------------------------------------------------*)

(* Construct the digraph of all state transitions of a process which
   are events of a particular set *)
fun digraph(p:process ref, gamma)=
let
  fun sub_digraph ([],d) = d
    | sub_digraph ((s:state)::statelist, d) =
  sub_digraph(statelist, d@
  (map (fn (e,n) => (#number s, n))
  (filter (fn (e,n) => e member gamma ) (#transitions s) )))
```

```
in
  sub_digraph(#states (!p), [])
end;

(* Extract a state of a particular number from a list *)
exception Extract_state;
fun extract_state (n, []) = raise Extract_state
  | extract_state (n, (s:state)::ss ) = if (#number s) = n then s
else extract_state (n,ss);

(* Initial events of a state *)
fun initials (s:state) = setof (map left (#transitions s));

(* Successor states numbers of a particular state *)
fun successors (s:state) = setof (map right (#transitions s));

(* Renumber the states of a process *)
fun renumber_states (from, to, p:process ref)=
let
  fun convert x =
  if x member from then
    to
  else
    x
in
  ref ({number=(#number (!p)),alphabet=(#alphabet (!p)),initial_state=
  convert(#initial_state (!p)),states=( map
  (fn (s:state) => {number=convert(#number s), refusals=(#refusals s),
  transitions=(map (fn (e,n) => (e, convert n)) (#transitions s))})
  (#states (!p))),data=(#data (!p))}:process)
end;

(* Evaluate the communication graph and vocabulary of a network (we
   assume that the network is known to be triple-disjoint). This is less
   efficient than the algorithm described in my thesis *)
local
  fun get_communication_graph_and_vocabulary2 ([],e'',v'') = (e'',v'')
    | get_communication_graph_and_vocabulary2
  ((p::net): process ref list,e'',v'') =
  let
    val alpha_p = (#alphabet (!p))
    fun new_edges_and_vocab ([],e,v) = (e,v)
      | new_edges_and_vocab ((p'::net'):process ref list,e,v) =
    let
      val v' = ((#alphabet (!p')) intersect alpha_p)
    in
      if v' = [] then
        new_edges_and_vocab (net',e,v)
      else
        new_edges_and_vocab (net', (#number (!p),#number (!p'))::e,v'@v)
    end
    val (new_edges, new_vocab) = new_edges_and_vocab (net,[],[])
  in
    get_communication_graph_and_vocabulary2
    (net,new_edges@e'', new_vocab@v'')
  end
in
  fun get_communication_graph_and_vocabulary (net:process ref list) =
  get_communication_graph_and_vocabulary2(net,[],[])
end;

(* Extract a process of a particular number from a list *)
exception Extract_process;
fun extract_process (n,[]) = raise Extract_process
  | extract_process (n,(p:process ref)::net) =
```

```
if (#number (!p)) = n then
  p
else
  extract_process (n,net);

(* Extract the backwards channels for each process within a network
   i.e. those channels connected to preceding processes in the list *)
fun extract_backwards_communications (net: process ref list) =
let
  fun extract_backwards_channels(xs,[])=xs
    | extract_backwards_channels(xs,x::(ys: process ref list))=
  extract_backwards_channels(xs@[(x,(#alphabet (!x)) intersect
   (flat (map (fn x => (#alphabet (!(left x)))) xs)))],ys)
in
  extract_backwards_channels([],net)
end;

(*------------------------------------------------------------------|
|  Normalisation of transition systems - using Roscoe's algorithm     |
|-------------------------------------------------------------------*)

(* Evaluate all divergent state numbers of a process that may be either
   partially normalised, or not normalised at all. We form a digraph of
   all tau events in the transition system of the process. Any arc
   which has its destination node already marked as divergent is
   replaced by a loop from its node of origin. We then form a list of
   nodes from which there is a path to a circuit in the digraph. *)
fun divergent_states (p:process ref) =
nodes_without_end (map
(fn (x,y)=>if y=divergence_state then (x,x) else (x,y))
(digraph (p, [tau_event])));

fun mark_divergent_states p = renumber_states (divergent_states p,
divergence_state, p);

(* Expand a set of state numbers of a process to incorporate all those
   states which may be reached using only event tau *)
fun tau_closure (seeds, p:process ref)=
let
  fun tau_successors seeds =
  let
    val all_transitions=
    flat(map (fn x=>
     (#transitions (extract_state(x,#states (!p))))) seeds)
  in
    setof(map right (filter (fn (e,n)=> e=tau_event) all_transitions))
  end
  fun form_tau_closure([],found)=found
    | form_tau_closure(pending,found)=
  let
    val extra=tau_successors(pending)
  in
    form_tau_closure(extra minus_set found, found union extra)
  end
in
  form_tau_closure(seeds,seeds)
end;

(* From a set of state numbers, determine the maximal refusal sets of
   the stable states, i.e. those states with no tau transition *)
fun form_refusals(seeds, p:process ref)=
maximal_sets(flat ( map (fn s=> #refusals s)
(filter (fn s=> not (exists (fn (e,n)=>e=tau_event) (#transitions s)))
(map (fn n=> extract_state(n,#states (!p))) seeds))));
```

```
(* Prenormalise a process, which has already had its divergent states
   marked *)
local
  fun process_seeds([],new_states:(state*(int list)) list,
  t,pending',num_used,_)=
  (t,pending',num_used)
    | process_seeds((e,seeds)::pending,new_states,t,pending',num_used,
  (n,current_seeds))=
  if divergence_state member seeds then
    process_seeds(pending,new_states,(e,divergence_state)::t,pending',
    num_used,(n,current_seeds))
  else
    let
      val x=map (fn y=> #number(left(y)))
      (get(fn (s,seeds')=>seeds' equals_set seeds) new_states)
      val x=
      if x=[] then
        map left
        (get (fn (n,seeds')=>seeds' equals_set seeds) pending')
      else
        x
      val x=
      if (x=[] andalso seeds=current_seeds) then
        [n]
      else
        x
    in
      if x=[] then
        process_seeds(pending,new_states,(e,num_used+1)::t,
         (num_used+1,seeds)::pending',num_used+1,(n,current_seeds))
      else
        process_seeds(pending,new_states,(e,hd x)::t,pending',
        num_used,(n,current_seeds))
  end
  fun combine_states([],new_states:(state*(int list)) list,
  num_used,p:process ref)=map left new_states
    | combine_states((n,seeds)::pending,new_states,num_used,p)=
  let
    val pa=partition (
    flat(map (fn x=>
    (filter (fn (e,n) => not(e = tau_event)))
    (#transitions (extract_state (x,#states (!p))))) seeds),
    (fn ((e,n),(e',n'))=>e=e') )
    val pa'=map (fn x=>(let val(y,z)=split x in
    (hd y,(tau_closure(z,p))) end)) pa
    val (t,pending',num_used')=process_seeds(pa',new_states,
    [],[],num_used,(n,seeds))
  in
    combine_states(pending@pending',
    ({number=n,transitions=t,refusals=form_refusals(seeds,p)},seeds)::
    new_states,num_used',p)
  end
in
  fun prenormalise (p:process ref)=
  if (#initial_state (!p) = divergence_state)then
    ref ({number=(#number (!p)),alphabet=(#alphabet (!p)),initial_state=
    divergence_state,states=[],data=(#data (!p))}:process)
  else
    let
      val seeds=tau_closure([#initial_state (!p)],p)
    in
      ref ({number=(#number (!p)), alphabet=(#alphabet (!p)), initial_state=0,
      states=combine_states([(0,seeds)],[],0,p),data=(#data(!p))}:process)
    end
end;
```

```
(* Equivalence relation to form initial partition of states, in state
   contraction phase of normalisation *)
fun same_marking (s,s') =
((#number s) = divergence_state andalso (#number s') = divergence_state)
orelse ((initials s) equals_set (initials s') andalso
(#refusals s) equals_setofsets (#refusals s'));

(* Iterative function applied to state partition. in state contraction
   phase of normalisation *)
local
  fun same_class([],n,n')=false
    | same_class((class:state list)::classes,n,n')=
  ((exists (fn s => (#number s) = n) class) andalso
  (exists (fn s => (#number s) = n') class)) orelse
  same_class(classes,n,n')
  fun rel part' (s:state,s':state) =
  (#number s = divergence_state)orelse
  forall
  (fn (e,n) =>
  let
    val n'=right(hd(get (fn (e',_)=>e'=e) (#transitions s')))
  in
    same_class (part',n,n')
  end) (#transitions s)
in
  fun next_partition part =
  flat(map (fn class => partition (class, rel part)) part)
end;

(* Contraction of states of prenormalised process *)
fun compact (p:process ref) =
let
  val initial_partition = partition(#states (!p),same_marking)
  val final_partition = fixed_point next_partition initial_partition
  val new_states=filter (fn s=>not(#number s = divergence_state))
  (map hd final_partition)
  fun renumber ([],p:process ref)=p
    | renumber ((class:state list)::classes, p)=
  renumber_states( map (fn x => #number x) class,#number(hd class),
  renumber(classes,p))
  val p'=ref ({number=(#number (!p)),alphabet=(#alphabet (!p)),initial_state=
  (#initial_state (!p)),states=new_states,data=(#data(!p))}:process)
in
  renumber(final_partition,p')
end;

fun normalise(p:process ref) =
compact(prenormalise(mark_divergent_states p));

(*-----------------------------------------------------------------|
|  Hiding events in networks and processes                         |
|-----------------------------------------------------------------*)

(* Apply the CSP hiding operation to a normal form transition system.
   This is done by relabelling any hidden transitions as tau, removing
   hidden events from refusal sets, and then renormalising the resultant
   transition system *)
local
  fun map_to_tau gamma (p:process ref) =
  let
    fun convert e=
    if e member gamma then
      tau_event
    else
```

```
      e
  in
    ref ({number= (#number (!p)),alphabet= (#alphabet (!p))
    minus_set gamma, initial_state= #initial_state (!p),
    states=map (fn s=> {number= #number s, refusals=
    if gamma disjoint (initials s) then (*stable*)
      maximal_sets (map (fn x => x minus_set gamma) (#refusals s))
    else (*unstable*)
      [[]],
    transitions=map(fn(e,n)=>(convert e,n)) (#transitions s)})
    (#states (!p)),data=(#data(!p))}:process)
  end
in
  infix hide
  fun (p:process ref) hide [] = p
    | p hide gamma = normalise(map_to_tau gamma p)
end;

(* Restrict a network to its vocabulary *)
fun restrict_to_vocabulary ()=
let
  val gamma = right(get_communication_graph_and_vocabulary (!(!user_net)))
  val net' = map (fn p => p hide ((#alphabet (!p)) minus_set gamma))
  (!(!user_net))
in
  make_derived_net net'
end;

(*-------------------------------------------------------------------|
|  Checks for network prerequisites                                  |
|-------------------------------------------------------------------*)

(* Check for triple-disjointedness: concatenate the process alphabets,
   sort the resulting list, then look for a sub-sequence of three
   repeated terms. This assumes that no process alphabet contains a
   repeated term. This is a less efficient algorithm than that which is
   documented in my thesis *)
fun triple_disjoint (net : process ref list) =
let
  fun no_three_sequence [] = true
    | no_three_sequence (x::y::z::xs) =
  if x = y andalso y = z then
    (print("Network "^(!net_name)^" is not triple-disjoint:\n");
    print("Event "^event(x)^" appears in more than two processes:\n");
    map (fn p => if x member( #alphabet (!p)) then
      print ("    "^(pname p)^"\n")
    else () ) net;
    false)
  else
    no_three_sequence (y::z::xs)
    | no_three_sequence xs = true
  fun alphabets ([] : process ref list) = [ [] ]
    | alphabets (x::xs) = ((#alphabet (!x))::alphabets xs)
in
  if no_three_sequence (tmergesort (flat (alphabets (net)))) then
    (print("Network "^(!net_name)^" is triple-disjoint\n");true)
  else
    false
end;

(* Check for business. Need to check that each process is non-
   terminating, deadlock-free and divergence-free, i.e there is no state
   of any process with a refusal set equal to its alphabet nor is there
   any divergence state *)
fun busy_process (p : process ref) =
```

```
let
  fun busy_state (s: state) =
  not (exists (fn (_,n) => if n = divergence_state then
  (print("Process "^(pname p)^" is divergent\n");true) else false)
  (#transitions s))
  andalso
  not (exists (fn x => if x equals_set (#alphabet (!p)) then
  (print("Process "^(pname p)^" is not busy\n");true) else false)
  (#refusals s))
in
  (if (#initial_state (!p)) <> divergence_state then true else
  (print("Process "^(pname p)^" is divergent\n");false)) andalso
  forall busy_state (#states (!p))
end

fun busy (net : process ref list) =
if forall busy_process net then
  (print("Network "^(!net_name)^" is busy\n");true)
else
  (print("Network "^(!net_name)^" is not busy\n");false);

(*-------------------------------------------------------------------|
|  Check for conflict-freedom of a pair of processes                 |
|------------------------------------------------------------------*)

(* Consider the state of two processes composed in parallel, and
   determine the set of state pairs that they can move to together by
   performing a single event. Either they will simultaneously perform an
   event in the intersection of their alphabets, or one process only
   will perform an event outside the alphabet of the other *)
local
  fun make_pair(transitions,e,n) =
  map (fn (e',n')=>(n,n') ) (get (fn (e',_) => e'=e ) transitions)
  fun new_pairs([],transitions)=[]
    | new_pairs((e,n)::pending,transitions)=
  new_pairs(pending,transitions)@make_pair(transitions,e,n)
in
  fun successor_pairs(s:state,s':state,alpha_p,alpha_p')=
  let
    val t = #transitions s
    and t' = #transitions s'
    val t1 = filter (fn x => not ((left x) member alpha_p')) t
    and t1' = filter (fn x => not ((left x) member alpha_p)) t'
    val t2 = t minus_set t1
    and t2' = t' minus_set t1'
  in
    setof((map (fn (e,n) => (n,#number s')) t1) @
     (map (fn (e',n') => (#number s, n')) t1') @ new_pairs(t2,t2'))
  end
end;

(* We wish to determine whether two processes are conflict free with
   respect to a set of events, gamma. We need to check that every pair
   of states, that P and Q can be in simultaneously, satisfies the
   following condition. If P wants to communicate with Q and vice-
   versa, but there is no event that they can agree on, then either
   P or Q can perform an event outside gamma. *)
fun conflict_free_refusal_set_pair(x,x',alpha_p,alpha_p',gamma)=
let
  val r=alpha_p minus_set x
  and r'=alpha_p' minus_set x'
in
  not(not(r disjoint alpha_p') andalso not(r' disjoint alpha_p) andalso
  r disjoint r') orelse not (r subset gamma andalso r' subset gamma)
end;
```

```
fun conflict_free_state_pair(s:state,s':state,alpha_p,alpha_p',gamma)=
forall (fn x' => forall (fn x => conflict_free_refusal_set_pair(x,x',
alpha_p,alpha_p',gamma)) (#refusals s)) (#refusals s');

local
  fun checked_state_pairs([],_,_,_,_)=true
    | checked_state_pairs((n,n')::pending,done,p:process ref,
  p':process ref,gamma)=
  let
    val s = extract_state(n,#states (!p))
    and s' = extract_state(n',#states (!p'))
    and alpha_p = #alphabet (!p)
    and alpha_p' = #alphabet (!p')
    and done' = (n,n')::done
  in
    conflict_free_state_pair(s,s',alpha_p,alpha_p',gamma)andalso
    checked_state_pairs(pending union (filter (fn x=> not (x member done'))
    (successor_pairs(s,s',alpha_p,alpha_p'))), done', p, p',
    gamma)
  end
in
  infix conflict_free_wrt
  fun (p:process ref, p':process ref) conflict_free_wrt gamma =
  checked_state_pairs
  ([(#initial_state (!p), #initial_state (!p'))],[],p,p',gamma)
end;

(*-------------------------------------------------------------------|
|  Split network into essential components                           |
|-------------------------------------------------------------------*)

(* Attempt to factorise the deadlock analysis of a network. First the
   network is checked for business and triple_disjointedness. If this
   fails an exception is raised. Otherwise each disconnecting edge of
   the communication graph is tested for conflict-freedom with respect
   to the vocabulary, and if successful is removed from the
   communication graph. The connected components of the remainder of
   the graph is returned. By Brookes and Roscoe's Theorem, proof of
   deadlock-freedom of the network is reduced to a proof of deadlock-
   freedom of each of these components, which are added to the stack *)
exception Decompose_network;
fun decompose_network () =
let
  val (g, gamma) = get_communication_graph_and_vocabulary (!(!user_net))
in
  if (triple_disjoint (!(!user_net)) andalso busy (!(!user_net)))then
    let
      val remainder =
      disjoint_components(g minus_set
      (filter
      (fn (n,n') =>
      let
        val p = extract_process (n,(!(!user_net)))
        and p' = extract_process (n',(!(!user_net)))
      in
        if (p,p') conflict_free_wrt gamma then
          (print((pname p)^" and "^(pname p')^" are conflict-free"^
          " wrt vocab\n");true)
        else
          (print((pname p)^" and "^(pname p')^" are not"^
          " conflict-free wrt vocab\n");false)
      end)
      (disconnecting_edges g)))
      (* sort components back into original order *)
```

```
      val remainder = map tmergesort remainder
    in
      if remainder = [] then
        deadlocks false
      else
        (print("Deadlock analysis reduced to: \n");
        map (fn x => (print("{"^(string_set (map process_name x))^"}\n")))
        remainder;
        make_subnetworks remainder;())
    end
  else
    raise Decompose_network
end;

(*-------------------------------------------------------------------|
|  Check for "acyclic" deadlock-freedom. SDD algorithm                |
|-------------------------------------------------------------------*)

(* For each pair of processes that communicate, (P,Q), find all the
   states of P and Q where P ->* Q. Amalgamate these pairs into
   a directed graph of states of processes in the network. If this
   has no circuits the network is deadlock-free. For a future
   enhancement note that circuits which cross over the same process
   twice should be permitted as it is not possible for a process to
   be in two states simultaneously  *)
fun ungranted_request_refusal_set_pair(x,x',alpha_p,alpha_p',gamma)=
let
(*  val _ = print("x="^event_list(x)^"\nx'="^event_list(x')^"\n")  *)
  val r=alpha_p minus_set x
  val r'=alpha_p' minus_set x'
in
  (not(r disjoint alpha_p')) andalso (r disjoint r') andalso (r subset gamma)
  andalso (r' subset gamma)
end;


(* If any strong conflict is found an exception is raised. Our methods
   cannot cope with networks with strong conflict *)
exception Strong_conflict of string;

fun ungranted_requests_state_pair(n,n',m:int,m':int,s:state,s':state,
alpha_p,alpha_p',gamma,p:process ref, p':process ref)=
let
  val i = ref 0
  and i' = ref 0
  and l = length(#refusals s)
  and l' = length(#refusals s')
  and found = ref []
in
  while (!i) < l do
    (i := (!i)+1;
    i' := 0;
    while (!i') < l' do
      (i':= (!i')+1;
      (let
        val x=nth(!i,#refusals s)
        and x'=nth(!i',#refusals s')
      in
        (if ungranted_request_refusal_set_pair(x,x',alpha_p,
        alpha_p',gamma)then
          (found := (((n,m,!i),(n',m',!i'))::(!found));
          if ungranted_request_refusal_set_pair(x',x,alpha_p',
          alpha_p,gamma)then
            (found := (((n',m',!i'),(n,m,!i))::(!found));
            if ((alpha_p minus_set x) subset alpha_p') orelse
```

```
            ((alpha_p' minus_set x') subset alpha_p) then
              (print ("Acceptance sets: ");
               print ("{"^event_list(alpha_p minus_set x)^"}, ");
               print ("{"^event_list(alpha_p' minus_set x')^"}\n");
               raise Strong_conflict ((pname p)^" and "^(pname p'));())
            else
              ())
           else ())
        else if ungranted_request_refusal_set_pair(x',x,alpha_p',
        alpha_p,gamma)then
          found := (((n',m',!i'),(n,m,!i))::(!found))
        else ())
      end)));
  !found
end;

fun ungranted_requests_process_pair(n,n',gamma)=
let
  val p = extract_process(n,!(!user_net))
  and p' = extract_process(n',!(!user_net))
  val _ = print("Checking "^(pname p)^" with "^(pname p')^"\n")
  val alpha_p = #alphabet(!p)
  and alpha_p' = #alphabet(!p')
  fun checked_state_pairs([],_,found)=found
    | checked_state_pairs((m,m')::pending,done,found)=
  let
    val s = extract_state(m,#states (!p))
    and s'= extract_state(m',#states (!p'))
    and done'=(m,m')::done
  in
    checked_state_pairs(pending union (filter (fn x=> not (x member done'))
    (successor_pairs(s,s',alpha_p,alpha_p'))), done', found@
    ungranted_requests_state_pair(n,n',m,m',s,s',alpha_p,alpha_p',gamma,p,p'))
  end
in
  checked_state_pairs([(#initial_state (!p), #initial_state(!p'))],[],[])
end;

fun state_dependence_digraph () =
let
  val (g,gamma) = get_communication_graph_and_vocabulary (!(!user_net))
  fun make_links (n,n') = ungranted_requests_process_pair(n,n',gamma)
in
  flat(map make_links g)
end;

(* When a potential cycle of ungranted requests is discovered we try
   to give the user some useful information in order to fix it *)
fun blockage_report (c) =
let
  val i = ref 1
  and l = length(c)
in
  while (!i) < l do
    (let
      val (n,m,r) = nth(!i,c)
      and (n',m',r') = nth((!i)+1,c)
      val p = extract_process (n,!(!user_net))
      and p' = extract_process (n',!(!user_net))
      val alpha_p = #alphabet (!p)
      val alpha_p' = #alphabet (!p')
      val s = extract_state (m, #states (!p))
      and s' = extract_state (m', #states (!p'))
      val x = nth(r,#refusals s)
      and x' = nth(r',#refusals s')
```

```
    in
      (print((pname p)^" ready to do ");
      print(implode(map (fn x=> event(x)^" ") ((alpha_p minus_set x)
      intersect alpha_p')));
      print("blocked by "^(pname p')^"\n"))
    end;
    i := (!i)+1)
end;

exception Acyclic_deadlock_free;
fun acyclic_deadlock_free () =
if (triple_disjoint (!(!user_net)) andalso busy (!(!user_net)))then
  (let
    val c = right(find_circuit (state_dependence_digraph ()))
  in
    if (c = []) then
      (deadlocks false;())
    else
      (print("Found possible cycle of ungranted requests:\n");
      blockage_report(c);
      ())
  end) handle Strong_conflict(mess) =>
  print("Strong conflict between "^mess^"\n")
else
  raise Acyclic_deadlock_free;


(*-----------------------------------------------------------------|
|  "Coloured" SDD algorithm                                         |
|-----------------------------------------------------------------*)

(* Find a circuit in a digraph of length > 2 if one exists *)
fun find_three_circuit digraph =
let
  fun three_circuit [] = []
    | three_circuit ((x,y)::remainder) =
  let
    val c = left(get_path(y,x,digraph minus_set [(y,x)]))
  in
    if c = [] then
      three_circuit remainder
    else c@[y]
  end
in
  three_circuit digraph
end;


(* As long as the network is strong conflict-free, a "coloured" digraph
   is legal if it has no blue arc which lies on a circuit of length > 2,
   nor any circuit of length >2 consisting purely of red arcs. *)
val red = 1; val blue = 2; val green = 3;

fun legal_coloured_digraph coloured_digraph =
let
  val digraph = map (fn (x,y,c) => (x,y)) coloured_digraph
  fun bad_blue_arc [] = []
    | bad_blue_arc ((x,y,c)::d) =
  if c = blue then
    (let  (* look for path of length > 1 from y to x *)
      val c = left(get_path(y,x,digraph minus_set [(y,x)]))
    in
      if c = [] then
        bad_blue_arc d
      else
```

```
        c@[y]
    end)
  else
    bad_blue_arc d
  val c = bad_blue_arc coloured_digraph
in
  if c <> [] then
    c
  else
    (let
      val digraph2 = map (fn (x,y,c) => (x,y))
      (filter (fn(x,y,c) => c=red ) coloured_digraph)
    in
      find_three_circuit digraph2
    end)
end;

(* For each pair of processes that communicate (P,P') we build up a
   list of state pairs (S,S') that they can be in simultaneously. At the
   same time we calculate (for each new pair discovered) M(S,S') - the
   difference between the number of times that the initial state of P has
   been crossed minus the number of times that the initial state of P' has
   been crossed. When we reach a state pair that has alreay been visited we
   check to see whether this variant has a consistent value. Once the list
   of state pairs has been constructed, we look for ungranted requests in
   this list. If, in states (S,S'), P->*P' then we add an arc to the
   digraph coloured_state_dependence_digraph. If M(S,S') is defined and >0
   we colour the arc green, if M(S,S') = 0 we colour the arc red, otherwise
   we colour it blue. We do a similar thing if P'->*P. The resulting
   digraph is then checked. If it has no circuit containing a blue arc and
   no circuit entirely consisting of red arcs the network is deadlock-free
*)
local
  fun make_pair(transitions,e,n,count,count',initial,initial') =
  map (fn (e',n')=>(n,n',if n=initial then count+1 else count,
  if n'=initial' then count'+1 else count') )
  (get (fn (e',_) => e'=e ) transitions)
  fun new_pairs([],transitions,count,count',initial,initial')=[]
    | new_pairs((e,n)::pending,transitions,count,count',initial,initial')=
  new_pairs(pending,transitions,count,count',initial,initial')
  @make_pair(transitions,e,n,count,count',initial,initial')
in
  fun successor_pairs_traces(s:state,s':state,count,count',alpha_p,alpha_p',
  initial,initial')=
  let
    val t = #transitions s
    and t' = #transitions s'
    val t1 = filter (fn x => not ((left x) member alpha_p')) t
    and t1' = filter (fn x => not ((left x) member alpha_p)) t'
    val t2 = t minus_set t1
    and t2' = t' minus_set t1'
  in
    setof((map (fn (e,n) => (n,#number s',if n = initial then
    count+1 else count, count')) t1) @
    (map (fn (e',n') => (#number s, n', count, if n' = initial'
    then count'+1 else count')) t1') @
    new_pairs(t2,t2',count,count',initial,initial'))
  end
end;

val undefined = ~9999;

fun state_pairs (p:process ref,p':process ref,alpha_p,alpha_p') =
let
  fun calc_status(m,m',count:int,count') = count - count'
```

```
  fun update_status(m,m',count,count',status) =
  if calc_status(m,m',count,count') = status then
    status
  else
    undefined
  fun visited_before((m,m',count,count'),[],done)=(false,done)
    | visited_before((m,m',count,count'),(m'',m''',status)::pending,done)=
  if m=m'' andalso m'=m''' then
    (true,(m,m',update_status(m,m',count,count',status))::(pending@done))
  else
    visited_before((m,m',count,count'),pending,(m'',m''',status)::done)
  fun form_state_pairs([],found)=found
    | form_state_pairs((m,m',count,count')::pending,found)=
  let
    val (flag,newlist) = visited_before((m,m',count,count'),found,[])
  in
    if flag then
      form_state_pairs(pending,newlist)
    else
      (let
        val s = extract_state (m,#states (!p))
        val s' = extract_state (m',#states (!p'))
        val xs=successor_pairs_traces(s,s',count,count',alpha_p,alpha_p',
        #initial_state (!p), #initial_state (!p'))
      in
        form_state_pairs(xs@pending,(m,m',calc_status(m,m',count,count'))::
        found)
      end)
  end
in
  form_state_pairs([(#initial_state (!p),#initial_state (!p'),1,1)],[])
end;

fun coloured_ungranted_requests (n,n',gamma)=
let
  val p = extract_process(n,!(!user_net))
  and p' = extract_process(n',!(!user_net))
  val _ = print("Checking "^(pname p)^" with "^(pname p')^"\n")
  val alpha_p = #alphabet(!p)
  and alpha_p' = #alphabet(!p')
  val state_pair_list = state_pairs(p,p',alpha_p,alpha_p')
  val consistent = forall (fn (m,m',status) => status <> undefined)
  state_pair_list
  fun get_colour ((n1,m1,x1),(n2,m2,x2),status) =
  if consistent then
    if n1 = n then
      if status > 0 then
         green
      else if status = 0 then
         red
      else
         blue
    else
      if status < 0 then
        green
      else if status = 0 then
         red
      else
         blue
  else
    blue
in
  flat(map ( fn (m,m',status) =>
  (map  (fn (k,k') => (k,k',get_colour(k,k',status)))
  (ungranted_requests_state_pair(n,n',m,m',extract_state(m,#states (!p)),
```

```
  extract_state(m',#states (!p')),alpha_p,alpha_p',gamma,p,p'))))
  state_pair_list)
end;

fun coloured_state_dependence_digraph () =
let
  val (g,gamma) = get_communication_graph_and_vocabulary (!(!user_net))
  fun make_links (n,n') = coloured_ungranted_requests(n,n',gamma)
in
  flat(map make_links g)
end;

exception Acyclic_deadlock_free2;
fun acyclic_deadlock_free2 () =
if (triple_disjoint (!(!user_net)) andalso busy (!(!user_net)))then
  (let
    val d = coloured_state_dependence_digraph ()
    val c = legal_coloured_digraph d
  in
    if (c = []) then
      (deadlocks false;())
    else
      (print("Found possible cycle of ungranted requests:\n");
      blockage_report(c);
      ())
  end) handle Strong_conflict(mess) =>
  print("Strong conflict between "^mess^"\n")
else
  raise Acyclic_deadlock_free2;

(*-------------------------------------------------------------------|
|  "Flashing" SDD algorithm                                          |
|-------------------------------------------------------------------*)

(* A flashing digraph is legal if it contains no circuit of length >2
   containing a non-flashing arc. This assumes that the network is
   strong-conflict-free *)
val flashing = 1; val non_flashing = 2;

fun legal_flashing_digraph flashing_digraph =
let
  val digraph = map (fn (x,y,c) => (x,y)) flashing_digraph
  fun bad_non_flashing_arc [] = []
    | bad_non_flashing_arc ((x,y,c)::d) =
  if c = non_flashing then
    (let  (* look for path of length > 1 from y to x *)
      val c = left(get_path(y,x,digraph minus_set [(y,x)]))
    in
      if c = [] then
        bad_non_flashing_arc d
      else
        c@[y]
    end)
  else
    bad_non_flashing_arc d
in
  bad_non_flashing_arc flashing_digraph
end;

local
  fun make_pair(transitions,e,n,status,status',alpha_p,alpha_p') =
  map (fn (e',n')=>(n,n',true,true))
  (get (fn (e',_) => e'=e ) transitions)
  fun new_pairs([],transitions,status,status',alpha_p,alpha_p')=[]
    | new_pairs((e,n)::pending,transitions,status,status',alpha_p,alpha_p')=
```

```
  new_pairs(pending,transitions,status,status',alpha_p,alpha_p')
  @make_pair(transitions,e,n,status,status',alpha_p,alpha_p')
in
  fun successor_pairs_statuses(s:state,s':state,
  status,status',alpha_p,alpha_p',gamma) =
  let
    val t = #transitions s
    and t' = #transitions s'
    val t1 = filter (fn x => not ((left x) member alpha_p')) t
    and t1' = filter (fn x => not ((left x) member alpha_p)) t'
    val t2 = t minus_set t1
    and t2' = t' minus_set t1'
  in
    setof((map (fn (e,n) => (n,#number s',
    if (e member gamma) then false else status ,status')) t1) @
    (map (fn (e',n') => (#number s, n',
    status, if (e' member gamma) then false else status')) t1') @
    new_pairs(t2,t2',status,status',alpha_p,alpha_p'))
  end
end;

fun state_pairs2 (p:process ref,p':process ref,alpha_p,alpha_p',gamma) =
let
(* If ever a status value of false is found, meaning that the process has
   not most recently communicated with the other one, then that is
   the permanent setting *)
  fun visited_before((m,m',status,status'),[],done)=(false,done)
    | visited_before((m,m',status,status'),
      (m'',m''',status'',status''')::pending,done)=
  if m=m'' andalso m'=m''' then
    (true,(m,m',status andalso status'',status' andalso status''')
    ::(pending@done))
  else
    visited_before((m,m',status,status'),pending,
    (m'',m''',status'',status''')::done)
  fun form_state_pairs([],found)=found
    | form_state_pairs((m,m',status,status')::pending,found)=
  let
    val (flag,newlist) = visited_before((m,m',status,status'),found,[])
  in
    if flag then
      form_state_pairs(pending,newlist)
    else
      (let
        val s = extract_state (m,#states (!p))
        val s' = extract_state (m',#states (!p'))
        val xs=successor_pairs_statuses(s,s',status,status',
        alpha_p,alpha_p',gamma)
      in
        form_state_pairs(xs@pending,(m,m',status,status')::
        found)
      end)
  end
in
  form_state_pairs([(#initial_state (!p),
  #initial_state (!p'),false,false)],[])
end;

(* An ungranted request from p to p' is set to "flashing" if p' must
   have previously communicated with p and more recently than with
   any other process *)
fun flashing_ungranted_requests (n,n',gamma)=
let
  val p = extract_process(n,!(!user_net))
  and p' = extract_process(n',!(!user_net))
```

```
  val _ = print("Checking "^(pname p)^" with "^(pname p')^"\n")
  val alpha_p = #alphabet(!p)
  and alpha_p' = #alphabet(!p')
  val state_pair_list = state_pairs2(p,p',alpha_p,alpha_p',gamma)
  fun set_flashing ((n1,m1,x1),(n2,m2,x2),status1,status2) =
  if n1 = n then
    if status2 then
       flashing
    else
       non_flashing
  else
    if status1 then
      flashing
    else
      non_flashing
in
  flat(map ( fn (m,m',status1, status2) =>
  (map  (fn (k,k') => (k,k',set_flashing(k,k',status1,status2)))
  (ungranted_requests_state_pair(n,n',m,m',extract_state(m,#states (!p)),
  extract_state(m',#states (!p')),alpha_p,alpha_p',gamma,p,p'))))
  state_pair_list)
end;

fun flashing_state_dependence_digraph () =
let
  val (g,gamma) = get_communication_graph_and_vocabulary (!(!user_net))
  fun make_links (n,n') = flashing_ungranted_requests(n,n',gamma)
in
  flat(map make_links g)
end;

exception Acyclic_deadlock_free3;
fun acyclic_deadlock_free3 () =
if (triple_disjoint (!(!user_net)) andalso busy (!(!user_net)))then
  (let
    val d = flashing_state_dependence_digraph ()
    (* val _ = print(makestring(d)) *)
    val c = legal_flashing_digraph d
  in
    if (c = []) then
      (deadlocks false;())
    else
      (print("Found possible cycle of ungranted requests:\n");
      blockage_report(c);
      ())
  end) handle Strong_conflict(mess) =>
  print("Strong conflict between "^mess^"\n")
else
  raise Acyclic_deadlock_free3;

(*------------------------------------------------------------------|
|  Check for divergence freedom in network                          |
|-------------------------------------------------------------------*)

(* We implement the proof rule of Roscoe to check for divergence-
   freedom in a network, after the vocabulary has been hidden. The
   order of processes in the network is significant. If no process can
   communicate indefinitely with its predecessors in the list then the
   network is divergence-free after the vocabulary is hidden *)
fun communicate_indefinitely (p:process ref, gamma) =
not (circuit_free (digraph(p, gamma)));

exception Proven_divergence_free;
fun proven_divergence_free () =
if (busy (!(!user_net))) then
```

```
  if (exists communicate_indefinitely
  (extract_backwards_communications (!(!user_net)))) then
    (print("Unable to establish divergence-freedom for "^
    (!net_name)^"\n");())
  else
    (print("Network "^(!net_name)^" is livelock-free\n");())
else
  raise Proven_divergence_free;

(*-----------------------------------------------------------------|
|  Check for deadlock in network of cyclic processes               |
|-----------------------------------------------------------------*)

(* We check that a process p is cyclic and derive its channel
   dependencies. This is done in two passes. The first pass assumes that
   the process is indeed cyclic. In each state s we look at every
   transition (e,s') that does not take us back to the initial state of
   p. If an event e' is possible in state s' that was refused in state s
   we assume that e'>e. The second pass checks that p really does
   conform to the cyclic ordering that we deduced on the first pass. We
   start at the initial state and check that there is a single maximal
   refusal set equal to all the non-minimal events in the alphabet of p.
   We also check that there is a transition for each minimal event of
   the alphabet of P. We then recursively perform similar checks on each
   of the successor states, making sure that they are exactly as we
   would expect, using a depth-first search *)
(* pass one *)
fun process_dependence (p: process ref) =
let
  fun state_dependence (s:state) =
  let
    fun transition_dependence (e,n') =
    if n'= (#initial_state (!p)) then [] else
    let
      val x = hd (#refusals s)
      and s'= extract_state(n', #states (!p))
      val x' = hd (#refusals s')
      val z = #alphabet (!p) minus_set x'
      val y = z intersect x
      fun construct_dependence (n'') = (n'',e)
    in
      map construct_dependence y
    end
  in
    flat (map transition_dependence (#transitions s))
  end
in
  flat (map state_dependence (#states (!p)))
end;

(* pass two *)
fun cyclic_process (p: process ref) =
let
  val g = process_dependence p
  infix has_been;
  fun x has_been [] = false
    | x has_been (y::l) = (x= left y) orelse (x has_been l);
  fun checked_states ([],p':process ref,visited) = true
    | checked_states ((n,done)::xs, p', visited)=
  if n has_been visited then
  (((n = #initial_state (!p')) andalso (done equals_set #alphabet (!p')))
  orelse exists(fn (n',done') => n'=n andalso done equals_set done')
  visited) andalso checked_states(xs, p', visited)
  else
  let
```

```
    val ready_set = minimal_elements (#alphabet (!p') minus_set done, g)
    val s = extract_state(n, #states (!p'))
    fun successors [] = []
      | successors ((e,n')::xs) = (n',e::done)::(successors xs)
  in
    length (#refusals s) = 1
    andalso (hd(#refusals s) equals_set
    ((#alphabet (!p')) minus_set ready_set))
    andalso (initials (s) equals_set ready_set)
    andalso
    checked_states(successors(#transitions s)@xs, p', (n,done)::visited)
  end
in
  if checked_states([(#initial_state (!p),[])],p,[]) then
  (print("Process "^(pname p)^" is cyclic-po:\n"^event_graph(g));
  (g,true))
  else (print("Process "^(pname p)^" is not cyclic-po\n");
  ([],false))
end;

exception Cyclic_network;
fun cyclic_network () =
if (busy (!(!user_net)) andalso triple_disjoint (!(!user_net))) then
  let
    val xs = (map cyclic_process (!(!user_net)))
  in
    if exists (fn (_,y)=> (y=false)) xs then
      (print("Not a cyclic-po network\n");())
    else
      let
        val (z,w) = find_circuit (flat (map left xs))
      in
        if (w=[]) then
          deadlocks false
        else
          (print("Found closed trail of dependent channels:\n<"^
          event_list(w)^">\n");
          deadlocks true)
      end
  end
else
  raise Cyclic_network;

(*-------------------------------------------------------------------|
|  Check for deadlock in client-server network                      |
|-------------------------------------------------------------------*)

(* Check that the client and server bundles of a process are disjoint,
   each is contained within the alphabet, and has length 1 or 2. *)
fun valid_bundles ((p:process ref), clients, servers) =
let
  val y = clients@servers
  val x = flat y
  fun right_length b =
  let
    val len = length b
  in
    len = 1 orelse len = 2
  end
in
  length x = length (setof x)
  andalso x subset (#alphabet (!p))
  andalso forall right_length y
end;
```

```
(* Check that an individual busy process obeys the four requirements of the
   client-server protocol on its client and server channel bundles *)
fun obeys_protocol ((p:process ref), clients, servers) =
let
  fun ready_or_not es [] = true
    | ready_or_not es (r::rs) =
  ((es subset r) orelse (es disjoint r))
  andalso ready_or_not es rs
  fun pairs []=[]
    | pairs (x::xs) = if length x = 2 then (hd x, hd (tl x))::pairs xs
  else pairs xs
  fun alternate guarantee_ack (req,ack) =
  let
    infix has_been
    fun x has_been [] = false
      | x has_been (y::l) = (x = left y) orelse (x has_been l);
    fun cannot_refuse y [] = true
      | cannot_refuse y (x::xs) = not (ack member x) andalso
    cannot_refuse y xs
    fun successors (n', level') [] = []
      | successors (n', level') ((e,n'')::ts) =
    if e = req then (n'', level'+1)::(successors (n', level') ts)
    else if e = ack then (n'', level'-1)::(successors (n', level') ts)
    else (n'', level')::(successors (n', level') ts)
    fun consistent_states ([],visited) = true
      | consistent_states ((n,level)::pending,visited) =
    (level = 0 orelse level = 1) andalso
    ( if (guarantee_ack andalso level = 1) then
    cannot_refuse ack (#refusals (extract_state (n, #states (!p))))
    else true )
    andalso
    ( if (n has_been visited) then
    (((n,level) member visited)
    andalso consistent_states (pending, visited))
    else
    consistent_states(
    (successors (n,level) (#transitions (extract_state (n, #states (!p)))))
    @pending, (n,level)::visited) )
  in
    consistent_states ([((#initial_state (!p)),0)],[])
  end
  fun alternating_guaranteeing_ack z = alternate true z
  fun alternating_not_guaranteeing_ack z = alternate false z
in  (* process p is assumed to be busy *)
  valid_bundles (p, clients, servers)
  andalso
  ready_or_not (setof (map hd servers))
  (flat (map #refusals (#states (!p))))
  andalso forall alternating_not_guaranteeing_ack (pairs servers)
  andalso forall alternating_guaranteeing_ack (pairs clients)
end;

(* Deduce the server channel bundles from the server channels of process p.
   This will only work on client-server process that doesn't use
   "polling". I.e. we assume that server communication can only take
   place in a stable state of the process, when no internal activity is
   possible. I.e. any server event that can be performed in state sigma
   of p is not a member of every maximal refusal set of sigma. Use a DFS
   to find the first time that the process becomes ready to communicate
   on a server channel. By the ready-or-not rule it must then be ready
   to communicate on all its server request or drip channels. For each
   transition that is a server event we can then construct a server
   bundle by doing a DFS to find the next time that the process becomes
   ready to do a server event. If this is a request or drip then the
   channel we were just looking at is a drip, otherwise the second
```

```
   channel is the acknowledgement of the first. *)
exception Extract_server_bundles of string;
fun extract_server_bundles (p:process ref, []) = []
  | extract_server_bundles (p, server_set) =
let
  fun get_next_server_state([],visited) = []
    | get_next_server_state(n::pending,visited) =
  let
    fun server_state (s':state)= not ((map left (#transitions s'))
    disjoint server_set)
    val s = extract_state (n,#states (!p))
  in
    if server_state s then [(s,(map left (#transitions s))
    intersect server_set)] else
    get_next_server_state((successors s)@pending,n::visited)
  end
  (* There must be some state of p which can perform an event in server_set *)
  val s = left(hd(get_next_server_state([#initial_state (!p)], [])))
  fun choose_refusal (x::xs) =
  if server_set subset x then
    choose_refusal xs
  else
    x
  val rs = ((#alphabet (!p)) minus_set choose_refusal (#refusals s))
  intersect server_set
  fun make_bundle r =
  let
    fun next_state ((r',n')::list) =
    if r=r' then
      n'
    else
      next_state list
    val sl =
    get_next_server_state([next_state (#transitions s)], [])
  in
    if (sl = []) orelse not ((right(hd(sl))) disjoint rs) then [r]
    else [r, hd (right(hd(sl)))]
  end
  val x = map make_bundle rs
in
  (* We need to check that the server bundles formed contain all the
  server channels *)
  if not ((flat x) equals_set server_set) then
    raise Extract_server_bundles (pname p)
  else
     x
end;

(* Form the client bundle list for each process from the server lists *)
fun form_client_bundles (list,[]) = list
  | form_client_bundles (list,(p,servers)::xs) =
let
  fun add_clients_to [] = []
    | add_clients_to ((p:process ref,c,s)::list') =
  let
    fun new_clients [] = []
      | new_clients (s::ss) = if s subset (#alphabet (!p)) then
    (s::new_clients ss) else new_clients ss;
  in
    (p,(new_clients servers)@c,s)::add_clients_to(list')
  end
in
  form_client_bundles((add_clients_to list)@[(p,[],servers)],xs)
end;
```

```
exception Extract_bundles;
fun extract_bundles net =
let
  val net2 = extract_backwards_communications net
  val net2 = combine(net, map extract_server_bundles net2)
  val x = form_client_bundles ([],net2)
in
  (* Check that each server bundle of the network matches a client bundle *)
  if (flat (map (fn (p,c,s) => c) x)) equals_set
  (flat (map (fn (p,c,s) => s) x)) then
    x
  else
    raise Extract_bundles
end;

(* Check a network for adherence to the client-server design rule *)
exception Client_server_network;
fun client_server_network ()=
if (busy (!(!user_net)) andalso triple_disjoint (!(!user_net))) then
  let
    fun client_server_protocol [] = true
      | client_server_protocol ((p,c,s)::xs) =
    (if obeys_protocol (p,c,s) then
      (print("Process "^(pname p)^" obeys client-server protocol\n");
      print("clients("^(pname p)^") =\n{"^(event_list_list c)^"}\n");
      print("servers("^(pname p)^") =\n{"^(event_list_list s)^"}\n");
      true)
    else
      (print("Process "^(pname p)^" disobeys client_server protocol\n");
      print("clients("^(pname p)^") =\n{"^(event_list_list c)^"}\n");
      print("servers("^(pname p)^") =\n{"^(event_list_list s)^"}\n");false))
    andalso client_server_protocol xs
  in
    if (client_server_protocol (extract_bundles (!(!user_net))))then
      deadlocks false
    else
      print("Deadlock-freedom has not been established\n")
  end
else
  raise Client_server_network;

(*-------------------------------------------------------------------|
|  Extended Resource Allocation Protocol                             |
|-------------------------------------------------------------------*)

(* Starting at the far end of a network <P1, P2,.. , PN>, we check
   each process to see whether it acts as a resource to its
   predecessors in the list. When we find one that does not act as a
   resource we assume that it is a user process and all its
   predecessors as well. There could be a problem with this approach
   if one of the user processes has been written so that it behaves
   like a resource - but this is extremely unlikely. Having
   partitioned the network into users and resources we turn our
   attention to the users. For each one we do the following check.
   Starting at the initial state we maintain a list of which resources
   are currently held in each state. If any are held then the process
   is only allowed to make claims to processes of lower priority than
   any of those held, and may not communicate with other user processes
   at all. We do a DFS, and on arrival at a state that has already been
   visited we must check that the current inventory of resources is
   consistent. If any of these checks fail the Extended Resource Allocation
   Protocol is not adhered to and we can go no further. Otherwise the
   deadlock analysis is reduced to consideration of the subnetwork of
   user processes. Note that this implementation allows a slight extension
   of the protocol in that a single process may have more than one pair
```

```
   of claim-release channel connection pairs to a single resource *)
fun claim_release_pairs (p:process ref) =
let
  val s0 = extract_state(#initial_state(!p),#states(!p))
  val t0 = initials(s0)
  fun make_pairs [] pairs = pairs
    | make_pairs ((e,n)::ts) pairs =
  let
    val s1 = extract_state(n,#states(!p))
  in
    if (length (#transitions s1)) = 1 andalso (length (#refusals s1)) = 1
    andalso (right(hd (#transitions s1))) = (#initial_state (!p))
    andalso ((#alphabet (!p)) minus_set (hd(#refusals s1))) equals_set
    initials(s1) then
      make_pairs ts ((e,left(hd(#transitions s1)))::pairs)
    else
      []
  end
in
  if (length(#refusals s0)) = 1
  andalso ((#alphabet (!p)) minus_set (hd(#refusals s0))) equals_set t0 then
    make_pairs (#transitions s0) []
  else
    []
end;

fun resource (ps:process ref list) (p:process ref) =
let
  (* val _ = print(pname p) *)
  val pairs = claim_release_pairs p
  (* val _ = print(makestring(map (fn (e,f) =>
  event(e)^":"^event(f)) pairs)) *)
  fun match_pairs [] matched_pairs = (true, matched_pairs)
    | match_pairs ((c,r)::pairs) matched_pairs =
  let
    val l = get (fn (p:process ref) => c member (#alphabet (!p))
    andalso r member (#alphabet (!p))) ps
  in
    if length(l) = 0 then  (* cannot match pair with any user process *)
      (false, [])
    else
      match_pairs pairs (( (#number (!(hd(l)))) , (#number (!p)) ,c,r)::
      matched_pairs)
    end
in
  if pairs = [] then
    (false, [])
  else (* match claim-release pairs with user processes *)
    match_pairs pairs []
end;

fun resource_allocation_protocol gamma cr_list (p:process ref) =
let
  val cr_list = filter (fn (x,_,_,_) => x = (#number (!p))) cr_list
  fun checked_states ([],visited) =
    true
    | checked_states ((ns,holding)::pending,visited) =
  let
    val invisited = get (fn (ns',h') => ns'=ns) visited
    val lowest = min holding handle Min => ~999 (* for empty list *)
    fun valid_transition (e,n) =
    if e member gamma then
      if lowest <> ~999 then
        (print("User process "^pname(p)^" communicates on "^event(e)^
        " still holding a resource\n");(false,[]))
```

```
      else
        (true,[(n,holding)])
    else
      let
        val l = get (fn (n1,n2,c,r) => e=c) cr_list
      in
        if l<>[] then  (* a claim event *)
          if lowest = ~999 then  (* no resources held *)
            (true,[(n,[#2 (hd(l))])])
          else if (#2 (hd(l))) >= lowest then (* breaks protocol *)
            (print("User process "^pname(p)^" claims resource "^
            pname(extract_process ((#2 (hd(l))), (!(!user_net))))^
            " still holding "^pname(extract_process (lowest, (!(!user_net))))^
            "\n");(false,[]))
          else
            (true,[(n,(#2 (hd(l)))::holding)])
        else
          let
            val l = get (fn (n1,n2,c,r) => e=r) cr_list
          in
            if l<>[] then (* a release event *)
               if (#2 (hd(l))) member holding then
                 (true,[(n,holding minus_set [#2 (hd(l))])])
               else (* tries to release resource it does not hold *)
                 (print("User process "^pname(p)^" tries to release resource "^
                 pname(extract_process ( (#2 (hd(l))), (!(!user_net))))^
                 " which it does not hold\n");(false,[]))
            else
              (true,[(n,holding)])
          end
      end
    fun valid_state () =
    let
      val l = (map valid_transition
      (#transitions (extract_state (ns, #states (!p)))))
    in
      if forall (fn (x,s) => x) l then
        (true,flat(map right l))
      else
        (false,[])
    end
  in
    if invisited = [] then
      let
        val (ok,successors) = valid_state ()
      in
        if ok then (* form successor states and add to pending *)
          checked_states (successors@pending,(ns,holding)::visited)
        else
          false
      end
    else if holding=right(hd(invisited)) then  (* consistent inventory *)
      checked_states (pending,visited)
    else
      (print("User process "^pname(p)^" has inconsistent resource inventory\n"
      );false)
  end
in
  if cr_list = [] then
    (print("Process "^pname(p)^" uses no resources\n");true)
  else if checked_states ([(#initial_state (!p),[])],[])
  then
    (print("User process "^pname(p)^" obeys resource allocation protocol\n");
    true)
  else
```

```
    false
end;

fun get_users ([], cr_list) = ([],[])
  | get_users ((p:process ref)::ps, cr_list) =
let
  val (is_resource, cr_list') = resource ps p
in
  if is_resource then
    (print("Process "^pname(p)^" acts as a resource\n");
    get_users (ps, cr_list'@cr_list))
  else
    (print("Process "^pname(p)^" is not a resource\n");
    ((p::ps), cr_list))
end;

exception User_resource_network;
fun user_resource_network ()=
if (busy (!(!user_net)) andalso triple_disjoint (!(!user_net))) then
  let
    val (users,cr_list) = get_users( rev (!(!user_net)), [])
    val gamma = right(get_communication_graph_and_vocabulary users)
  in
    if forall (fn x=> x)
    (map (resource_allocation_protocol gamma cr_list) users) then
      if gamma <> [] then
        (make_derived_net (rev users);
        print("Deadlock analysis reduces to: \n");
        print("{"^(string_set (map pname (rev users)))^"}\n");())
      else
        deadlocks false
    else
      ()
  end
else
  raise User_resource_network;

(*-------------------------------------------------------------------|
|  teletype interface program                                        |
|-------------------------------------------------------------------*)

(* This is the function which controls an interactive session with
   the user *)
fun teletype () =
let
  val _ = (log := (open_out "check.log"))
  val command = ref "b"
in
  while !command <> "q" do
    (case !command of
      "b" => print("Welcome to Deadlock Checker\n")
    | "h" => (print(" h             - help: display this menu\n");
             print(" l <file>      - load network file\n");
             print(" n             - display list of networks in memory\n");
             print(" s <name>      - select network\n");
             print(" c             - display currently selected network\n");
             print(" p             - display list of processes in current "^
             "network\n");
             print(" d             - decompose network deadlock analysis\n");
             print(" v             - check for acyclic deadlock-freedom "^
             "(SDD algorithm)\n");
             print(" x             - check for acyclic deadlock-freedom "^
             "(CSDD algorithm)\n");
             print(" f             - check for acyclic deadlock-freedom "^
             "(FSDD algorithm)\n");
```

```
            print(" o             - check for deadlock in cyclic-po "^
            "network\n");
            print(" w             - check for deadlock in client-server "^
            "network\n");
            print(" a             - check for resource allocation "^
            "protocol\n");
            print(" r             - restrict network to its vocabulary\n");
            print(" t             - test for livelock-freedom "^
            "(Roscoe's rule)\n"))
     | "n" => list_networks ()
     | "c" => current_network ()
     | "p" => list_processes ()
     | "v" => ((acyclic_deadlock_free () handle Acyclic_deadlock_free =>
        print("Network unsuitable for this analysis\n"));
        update_network_status())
     | "x" => ((acyclic_deadlock_free2 () handle Acyclic_deadlock_free2  =>
        print("Network unsuitable for this analysis\n"));
        update_network_status())
     | "f" => ((acyclic_deadlock_free3 () handle Acyclic_deadlock_free3  =>
        print("Network unsuitable for this analysis\n"));
        update_network_status())
     | "d" => ((decompose_network () handle Decompose_network =>
        print("Network unsuitable for this analysis\n"));
        update_network_status())
     | "o" => ((cyclic_network () handle Cyclic_network =>
        print("Network unsuitable for this analysis\n"));
        update_network_status ())
     | "w" => ((client_server_network () handle Client_server_network =>
        print("Network unsuitable for this analysis\n")
        | Extract_bundles =>
        print("Cannot match client bundles to server bundles within network\n")
        | Extract_server_bundles (x) =>
        print ("Cannot partition server channels of process "^x^"\n"));
        update_network_status ())
     | "a" => ((user_resource_network () handle User_resource_network =>
        print("Network unsuitable for this analysis\n"));
        update_network_status ())
     | "r" => restrict_to_vocabulary ()
     | "t" => (proven_divergence_free () handle Proven_divergence_free =>
        (print("Network unsuitable for this analysis\n");()))
     | "" => ()
     | other =>
       (case (hd(explode other)) of
         "l" => (load_network (implode(tl(filter (fn x => x <> " ")
                (explode other))))
         handle Io(mess) => (print (mess^"\n");()) | Hd =>
         (print("Parse failed\n");()))
       | "s" => (if length(explode other) > 1 then
           select_network (implode(filter (fn x => x <> " ")
           (tl(explode other))))
         else
           print("invalid command\n");())
       | _ => (print("invalid command\n");()));
    print ("Command (h for help, q to quit):");
    command := (input_line std_in);
    output(!log, (!command)^"\n"));
  close_out (!log);
  ()
end;

(* When the software is loaded we go straight into interative mode *)
teletype ();
```

C Sample Network Definitions

Dining Philosophers - phils.csp

```
PHILNAMES= {0,1,2,3,4}
FORKNAMES = {0,1,2,3,4}

pragma channel eats:PHILNAMES
pragma channel takes,drops:PHILNAMES.FORKNAMES

PHIL(i) = takes.i.i -> takes.i.((i-1)%5) -> eats.i ->
          drops.i.((i-1)%5) -> drops.i.i -> PHIL(i)

FORK(i) = takes.i.i -> drops.i.i -> FORK(i) []
          takes.((i+1)%5).i -> drops.((i+1)%5).i -> FORK(i)

--+ PHIL(0),PHIL(1),PHIL(2),PHIL(3),PHIL(4)
--+ FORK(0),FORK(1),FORK(2),FORK(3),FORK(4)
```

Torus with Control Process - mtorus.csp

```
n = 4

indices = {0,1,2,3}

pragma channel e : indices . indices . {left,up,right,down}
pragma channel to_control, from_control : indices.indices

CELL(i,j) = if ((i+j)%2==0) then LEFT2(i,j) else RIGHT(i,j)

LEFT2(i,j) = to_control.i.j -> from_control.i.j -> LEFT(i,j) |~| LEFT(i,j)

LEFT(i,j) = e.i.j.left -> e.((i-1)%n).j.right -> UP(i,j) []
            e.((i-1)%n).j.right -> e.i.j.left -> UP(i,j)

UP(i,j) = e.i.j.up -> e.i.((j-1)%n).down -> RIGHT(i,j) []
          e.i.((j-1)%n).down -> e.i.j.up -> RIGHT(i,j)

RIGHT(i,j) = e.i.j.right -> e.((i+1)%n).j.left -> DOWN(i,j) []
             e.((i+1)%n).j.left -> e.i.j.right -> DOWN(i,j)

DOWN(i,j) = e.i.j.down -> e.i.((j+1)%n).up -> LEFT2(i,j) []
            e.i.((j+1)%n).up -> e.i.j.down -> LEFT2(i,j)

CONTROL = [] i:indices @ ([] j:indices @ (to_control.i.j -> from_control.i.j ->
          CONTROL))

--+ CELL(0,0),CELL(1,0),CELL(2,0),CELL(3,0)
--+ CELL(0,1),CELL(1,1),CELL(2,1),CELL(3,1)
--+ CELL(0,2),CELL(1,2),CELL(2,2),CELL(3,2)
--+ CELL(0,3),CELL(1,3),CELL(2,3),CELL(3,3)
--+ CONTROL
```

Bunjee Jump Simulation - bunjee.csp

```
indices = {0,1,2,3,4,5}
```

```
pragma channel left, right, dump, control:indices
pragma channel from_mouse, from_keyboard, to_screen

ROPE(i) = ROPEA(i);ROPEB(i);ROPEC(i);ROPE(i)
ROPEA(i) = (control.i -> SKIP [] SKIP) |~| SKIP
ROPEB(i) = left.i -> right.i -> SKIP [] right.i -> left.i -> SKIP
ROPEC(i) = dump.i -> SKIP |~| SKIP

USER_INTERFACE = USER_INTERFACE2 ||| END_ROPE_LEFT ||| END_ROPE_RIGHT
USER_INTERFACE2 = ((|~| i:indices @ control.i -> USER_INTERFACE2) |~|
                  to_screen -> USER_INTERFACE2) []
                  from_mouse -> USER_INTERFACE2 []
                  from_keyboard -> USER_INTERFACE2
END_ROPE_LEFT = left.0 -> END_ROPE_LEFT
END_ROPE_RIGHT = right.5 -> END_ROPE_RIGHT

MOUSE = from_mouse -> MOUSE

SCREEN = to_screen -> SCREEN

KEYBOARD = from_keyboard -> KEYBOARD

GRAPHICS = [] i:indices @ dump.i -> GRAPHICS

--+ MOUSE,KEYBOARD,SCREEN,USER_INTERFACE,GRAPHICS
--+ ROPE(0),ROPE(1),ROPE(2),ROPE(3),ROPE(4),ROPE(5)
```

Switching Network - `switching.csp`

```
pragma channel vlink:{0,1,2,3,4,5,6,7}.{0,1,2,3}
pragma channel xlink:{0,1,2,3,4,5,6,7}.{0,1,2}

SWITCH(in,out,swin,swout) =  in -> (out -> SWITCH(in,out,swin,swout) |~|
                                    SWITCH'(in,out,swin,swout)) []
                             swin -> out -> SWITCH(in,out,swin,swout)
SWITCH'(in,out,swin,swout)   =  swout -> SWITCH(in,out,swin,swout) []
                             swin -> out -> SWITCH'(in,out,swin,swout)

--+ SWITCH(vlink.0.0,vlink.0.1,xlink.0.0,xlink.1.0)
--+ SWITCH(vlink.1.0,vlink.1.1,xlink.1.0,xlink.0.0)
--+ SWITCH(vlink.2.0,vlink.2.1,xlink.2.0,xlink.3.0)
--+ SWITCH(vlink.3.0,vlink.3.1,xlink.3.0,xlink.2.0)
--+ SWITCH(vlink.4.0,vlink.4.1,xlink.4.0,xlink.5.0)
--+ SWITCH(vlink.5.0,vlink.5.1,xlink.5.0,xlink.4.0)
--+ SWITCH(vlink.6.0,vlink.6.1,xlink.6.0,xlink.7.0)
--+ SWITCH(vlink.7.0,vlink.7.1,xlink.7.0,xlink.6.0)
--+ SWITCH(vlink.0.1,vlink.0.2,xlink.0.1,xlink.2.1)
--+ SWITCH(vlink.1.1,vlink.1.2,xlink.1.1,xlink.3.1)
--+ SWITCH(vlink.2.1,vlink.2.2,xlink.2.1,xlink.0.1)
--+ SWITCH(vlink.3.1,vlink.3.2,xlink.3.1,xlink.1.1)
--+ SWITCH(vlink.4.1,vlink.4.2,xlink.4.1,xlink.6.1)
--+ SWITCH(vlink.5.1,vlink.5.2,xlink.5.1,xlink.7.1)
--+ SWITCH(vlink.6.1,vlink.6.2,xlink.6.1,xlink.4.1)
--+ SWITCH(vlink.7.1,vlink.7.2,xlink.7.1,xlink.5.1)
--+ SWITCH(vlink.0.2,vlink.0.3,xlink.0.2,xlink.4.2)
```

```
--+ SWITCH(vlink.1.2,vlink.1.3,xlink.1.2,xlink.5.2)
--+ SWITCH(vlink.2.2,vlink.2.3,xlink.2.2,xlink.6.2)
--+ SWITCH(vlink.3.2,vlink.3.3,xlink.3.2,xlink.7.2)
--+ SWITCH(vlink.4.2,vlink.4.3,xlink.4.2,xlink.0.2)
--+ SWITCH(vlink.5.2,vlink.5.3,xlink.5.2,xlink.1.2)
--+ SWITCH(vlink.6.2,vlink.6.3,xlink.6.2,xlink.2.2)
--+ SWITCH(vlink.7.2,vlink.7.3,xlink.7.2,xlink.3.2)
```