FDR Manual

Release 4.2.7

University of Oxford

May 11, 2020

CONTENTS

CHAPTER ONE

INTRODUCTION

FDR is a tool for analysing programs written in Hoare's CSP notation, in particular machine-readable CSP namely *CSPM*, which combines the operators of CSP with a functional programming language. The original FDR was written in 1991 by Formal Systems (Europe) Ltd, and a completely revised version FDR2 was released in the mid-1990s by the same organisation. The current version of the tool is FDR4, first released in October 2016 following FDR3 which was first released in 2013. Both of these versions were released by the University of Oxford, which also released FDR2 versions 2.90 and above in the period 2008-12.

FDR4.0 has extremely similar functionality to FDR2.94, but is completely re-written. The main differences are:

1. The *user interface* has been completely revised.
2. The *debugger* has been completely revised and gives simultaneous information about all components of a system, rather than one at a time.
3. There is an *integrated type checker* for CSP_M.
4. It now uses multi-core parallelism to speed up its operation.
5. A version of the *ProBE CSP animator* has been integrated.
6. There is *a utility* for drawing graphical representations of the labelled transition systems that represent processes within FDR.

The only significant functionality of FDR2.94 that FDR4.0 lacks is support for the revivals and refusal testing models of CSP and their divergence-strict versions (i.e. `[V=`, `[VD=`, `[R=` and `[RD=`). Note that the batch mode of FDR2.94 has been replaced by a new *machine-readable interface* based on standard formats (JSON, XML and YAML are supported).

FDR uses many algorithms and data structures. The ones used in FDR4 are in some cases the same, in some cases mildly modified, and in other cases completely new. Papers about FDR4 and its development can be found in *References*. Many books and papers have been written about CSP and earlier versions of FDR.

1.1 Citing FDR

When citing FDR, please refer to the following paper:

```
@inproceedings{fdr,
   title={{FDR3 --- A Modern Refinement Checker for CSP}},
   author={Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, A.W. Roscoe}
↪,
   booktitle={Tools and Algorithms for the Construction and Analysis of Systems},
   year = {2014},
   pages = {187-201},
   volume={8413},
```

(continues on next page)

(continued from previous page)

```
    series={Lecture Notes in Computer Science},
    editor={Ábrahám, Erika and Havelund, Klaus},
}
```

The manual may be cited as:

```
@manual{fdrmanual,
    title={{Failures Divergences Refinement (FDR) Version 3}},
    author={Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, A.W. Roscoe}
↪,
    year={2013},
    url={https://www.cs.ox.ac.uk/projects/fdr/},
}
```

CHAPTER

TWO

THE FDR USER INTERFACE

To launch FDR on Mac OS X simply open the FDR application that you have downloaded (normally this will be inside Downloads in your home folder). To launch FDR under Linux simply type `fdr4` from a command prompt (providing the installation instructions have been followed). Alternatively, a particular file can be loaded by typing `fdr4 file.csp` into a command prompt.

However FDR is launched, the main window that is presented is known as the *session window*, and is documented further in *Session Window*.

2.1 Getting Started

In this section we give a brief overview of the basics of operating FDR4. Firstly, we give recommended *installation instructions* before giving a short *tutorial introduction* to FDR4. If FDR4 is already installed, simply skip ahead to *A Short Tutorial Introduction*.

Warning: It is strongly recommended that when using FDR you have at least a basic knowledge of CSP, or are acquiring this by studying it. Roscoe's books *The Theory and Practice of Concurrency* and *Understanding Concurrent Systems* each contains an introduction to CSP that covers the use of FDR and, in particular, covers CSP_M.

2.1.1 Installation

To install FDR4 simply follow the installation instructions below for your platform.

Linux

The recommended method of installing FDR is to add the FDR repository using the software manager for your Linux distribution. This makes it extremely easy to update to new FDR releases, whilst also ensuring that FDR is correctly installed and accessible.

If your distribution uses `yum` (e.g. RHEL, CentOS or Fedora) as its package manager, the following commands can be used to install FDR:

```
sudo sh -c 'echo -e "[fdr]\nname=FDR Repository\nbaseurl=http://www.cs.ox.ac.uk/
↪projects/fdr/downloads/yum/\nenabled=1\ngpgcheck=1\ngpgkey=http://www.cs.ox.ac.uk/
↪projects/fdr/downloads/linux_deploy.key" > /etc/yum.repos.d/fdr.repo'
sudo yum install fdr
```

The first of the above commands adds the FDR software repository to `yum`, whilst the second command installs `fdr`. If your distribution uses `apt-get` (e.g. Debian or Ubuntu), then the following commands can be used to install FDR:

```
sudo sh -c 'echo "deb http://www.cs.ox.ac.uk/projects/fdr/downloads/debian/ fdr␣
↪release\n" > /etc/apt/sources.list.d/fdr.list'
wget -qO - http://www.cs.ox.ac.uk/projects/fdr/downloads/linux_deploy.key | sudo apt-
↪key add -
sudo apt-get update
sudo apt-get install fdr
```

The first of these adds the FDR software repository to `apt-get`, the second installs the GPG key that is used to sign FDR releases, the third fetches new software from all repositories, whilst the last command actually installs FDR.

Alternatively, if your system does not use `apt-get` or `yum`, FDR can also be installed simply by downloading the `tar.gz` package. To install FDR from such a package, firstly extract it. For example, if you downloaded FDR4 to `~/Downloads/fdr-linux-x86_64.tar.gz`, then it can be extracted by running the following commands in a terminal:

```
cd ~/Downloads
tar xzvf fdr-linux-x86_64.tar.gz
```

This will create a folder `~/Downloads/fdr4`, that contains FDR4.

Next, pick an installation location and copy the FDR4 files to the location. For example, you may wish to install FDR4 in `/usr/local` and can do so as follows:

```
mv ~/Downloads/fdr4 /usr/local/fdr4
```

At this point FDR4 can be run be executing `/usr/local/fdr4/bin/fdr4`. In order to make it accessible from the command line simply as `fdr4`, a symbolic link needs to be created from a location on `$PATH` to `/usr/local/fdr4/bin/fdr4`. For example, on most distributions `/usr/local/bin` is on `$PATH` and therefore running:

```
ln -s /usr/local/fdr4/bin/fdr4 /usr/local/bin/fdr4
```

The above command may have to be run using `sudo`, i.e. `sudo ln -s /usr/local/fdr4/bin/fdr4 /usr/local/bin/fdr4`. At this point you should be able to run FDR4 by simply typing `fdr4` into the command prompt.

Mac OS X

To install FDR4 on Mac OS X, simply open the downloaded application, which is named `FDR4`. On the first run, FDR4 will offer to move itself to the `Applications` folder. FDR4 can now be opened like any other program, by double clicking on FDR4 within `Applications`.

> **Warning:** When running Mac OS X 10.8 or later with Gatekeeper enabled, in order to open FDR4 you need to right-click on FDR4, and select 'Open'.

2.1.2 A Short Tutorial Introduction

It is strongly recommended that when using FDR you have at least a basic knowledge of CSP, or are acquiring this by studying it. Roscoe's books *Understanding Concurrent Systems* and *Theory and Practice of Concurrency* each contains an introduction to CSP that covers the use of FDR and particular covers CSP_M. This introduction therefore does not attempt to give a detailed introduction to CSP.

As a quick introduction to FDR, including many of the new features in FDR4, we recommend downloading and completing the simple exercises in the following file.

Download intro.csp

```
-- Introducing FDR4.0
-- Bill Roscoe, November 2013

-- A file to illustrate the functionality of FDR4.0.

-- Note that this file is necessarily basic and does not stretch the
-- capabilities of the tool.

-- To run FDR4 with this file just type "fdr4 intro.csp" in the directory
-- containing intro.csp, assuming that fdr4 is in your $PATH or has been aliased
-- to run the tool.

-- Alternatively run FDR4 and enter the command ":load intro.csp".

-- You will see that all the assertions included in this file appear on the RHS
-- of the window as prompts. This allows you to run them.

-- This file contains some examples based on playing a game of tennis between A
-- and B.

channel pointA, pointB, gameA, gameB

Scorepairs = {(x,y) | x  <- {0,15,30,40}, y <- {0,15,30,40}, (x,y) != (40,40)}

datatype scores = NUM.Scorepairs | Deuce | AdvantageA | AdvantageB

Game(p) = pointA -> IncA(p)
         [] pointB -> IncB(p)

IncA(AdvantageA) = gameA -> Game(NUM.(0,0))
IncA(NUM.(40,_)) = gameA -> Game(NUM.(0,0))
IncA(AdvantageB) = Game(Deuce)
IncA(Deuce) = Game(AdvantageA)
IncA(NUM.(30,40)) = Game(Deuce)
IncA(NUM.(x,y)) =  Game(NUM.(next(x),y))
IncB(AdvantageB) = gameB -> Game(NUM.(0,0))
IncB(NUM.(_,40)) = gameB -> Game(NUM.(0,0))
IncB(AdvantageA) = Game(Deuce)
IncB(Deuce) = Game(AdvantageB)
IncB(NUM.(40,30)) = Game(Deuce)
IncB(NUM.(x,y)) = Game(NUM.(x,next(y)))
-- If you uncomment the following line it will introduce a type error to
-- illustrate the typechecker.
-- IncB((x,y)) = Game(NUM.(next(x),y))

next(0) = 15
next(15) = 30
next(30) = 40

-- Note that you can check on non-process functions you have written. Try typing
-- next(15) at the command prompt of FDR4.

-- Game(NUM.(0,0)) thus represents a game which records when A and B win
```

(continues on next page)

(continued from previous page)

```
-- successive games, we can abbreviate it as

Scorer = Game(NUM.(0,0))

-- Type ":probe Scorer" to animate this process.
-- Type ":graph Scorer" to show the transition system of this process

-- We can compare this process with some others:

assert Scorer [T= STOP
assert Scorer [F= Scorer
assert STOP [T= Scorer

-- The results of all these are all obvious.

-- Also, compare the states of this process

assert Scorer [T= Game(NUM.(15,0))
assert Game(NUM.(30,30)) [FD=  Game(Deuce)

-- The second of these gives a result you might not expect: can you explain why?
-- (Answer below....)

-- For the checks that fail, you can run the debugger, which illustrates why the
-- given implementation (right-hand side) of the check can behave in a way that
-- the specification (LHS) cannot.  Because the examples so far are all
-- sequential processes, you cannot subdivide the implementation behaviours into
-- sub-behaviours within the debugger.

-- One way of imagining the above process is as a scorer (hence the name) that
-- keeps track of the results of the points that A and B score.  We could put a
-- choice mechanism in parallel: the most obvious picks the winner of each point
-- nondeterministically:

ND = pointA -> ND |~| pointB -> ND

-- We can imagine one where B gets at least one point every time A gets one:

Bgood = pointA -> pointB -> Bgood |~| pointB -> Bgood

-- and one where B gets two points for every two that A get, so allowing A to
-- get two consecutive points:

Bg = pointA -> Bg1 |~| pointB -> Bg

Bg1 = pointA -> pointB ->  Bg1 |~| pointB -> Bg

assert Bg [FD= Bgood
assert Bgood [FD= Bg

-- We might ask what effect these choice mechanisms have on our game of tennis:
-- do you think that B can win a game in these two cases?

BgoodS = Bgood [|{pointA,pointB}|] Scorer
BgS = Bg [|{pointA,pointB}|] Scorer

assert STOP [T= BgoodS \diff(Events,{gameA})
```

(continues on next page)

(continued from previous page)

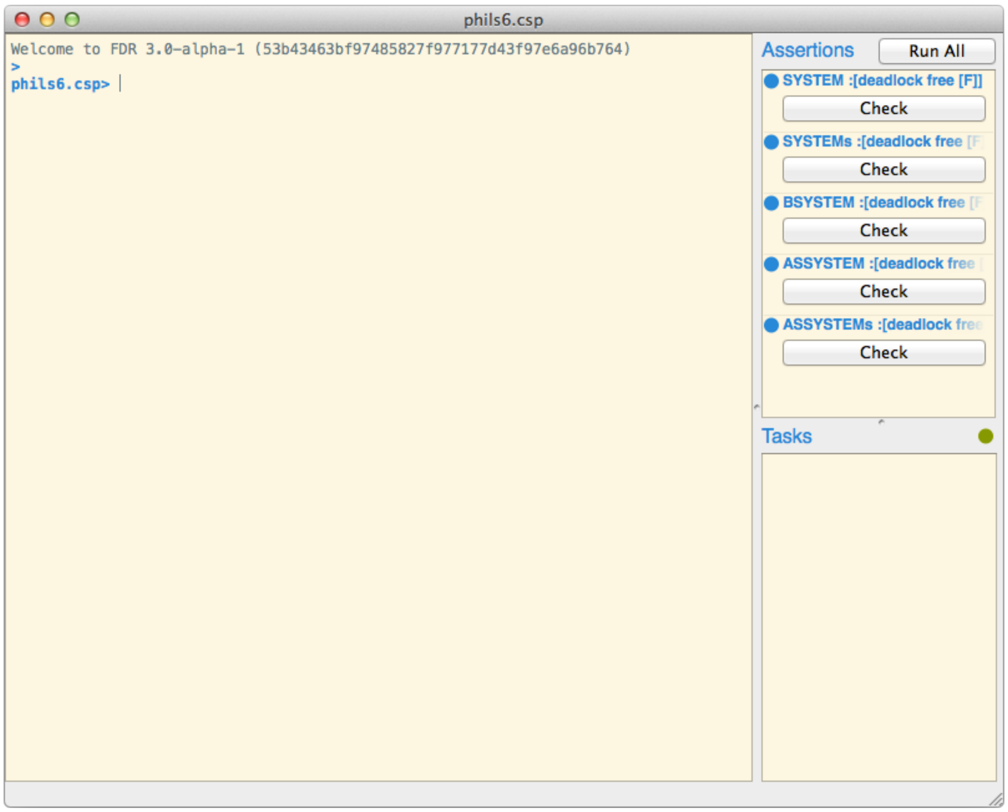
```
assert STOP [T= BgS \diff(Events,{gameA})

-- You will find that A can in the second case, and in fact can win the very
-- first game.  You can now see how the debugger explains the behaviours inside
-- hiding and of different parallel components.

-- Do you think that in this case A can ever get two games ahead?  In order to
-- avoid an infinite-state specification, the following one actually says that A
-- can't get two games ahead when it has never been as many as 6 games behind:

Level = gameA -> Awinning(1)
        [] gameB -> Bwinning(1)

Awinning(1) = gameB -> Level -- A not permitted to win here

Bwinning(6) = gameA -> Bwinning(6) [] gameB -> Bwinning(6)
Bwinning(1) = gameA -> Level [] gameB -> Bwinning(2)
Bwinning(n) = gameA -> Bwinning(n-1) [] gameB -> Bwinning(n+1)

assert Level [T= BgS \{pointA,pointB}

-- Exercise for the interested: see how this result is affected by changing Bg
-- to become yet more liberal. Try Bgn(n) as n copies of Bgood in ||| parallel.

-- Games of tennis can of course go on for ever, as is illustrated by the check

assert BgS\{pointA,pointB} :[divergence-free]

-- Notice that here, for the infinite behaviour that is a divergence, the
-- debugger shows you a loop.

-- Finally, the answer to the question above about the similarity of
-- Game(NUM.(30,30)) and Game(Deuce).

-- Intuitively these processes represent different states in the game: notice
-- that 4 points have occurred in the first and at least 6 in the second. But
-- actually the meaning (semantics) of a state only depend on behaviour going
-- forward, and both 30-all and deuce are scores from which A or B win just when
-- they get two points ahead.  So these states are, in our formulation,
-- equivalent processes.

-- FDR has compression functions that try to cut the number of states of
-- processes: read the books for why this is a good idea. Perhaps the simplest
-- compression is strong bisimulation, and you can see the effect of this by
-- comparing the graphs of Scorer and

transparent sbisim, wbisim, diamond

BScorer = sbisim(Scorer)

-- Note that FDR automatically applies bisimulation in various places.

-- To see how effective compressions can sometimes be, but that
-- sometimes one compression is better than another compare

NDS = (ND [|{pointA,pointB}|] Scorer)\{pointA,pointB}

```

(continues on next page)

(continued from previous page)

```
wbNDS = wbisim(NDS)
sbNDS = sbisim(NDS)
nNDS = sbisim(diamond(NDS))
```

2.2 Session Window



The main window of the GUI is the *session window*, and is illustrated above. The session window provides the main interface to CSP files, and allows them to be loaded (see `load`), expressions to be evaluated (see *Available Statements*) and assertions run (see *The Assertion List*). Below, we give an overview of the three main components of the session window; the *Interactive Prompt*, the *Assertion List* and the *Task List*.

2.2.1 The Interactive Prompt

The GUI is structured around an interactive prompt, in which expressions may be evaluated, new definitions given, and assertions specified. For example, if FDR was started with *Dining Philosophers* loaded (i.e. by typing `fdr4 phils6.csp` from a command prompt), then the following session is possible:

```
phils6.csp> head(<1..>)
1
phils6.csp> let f(x) = 24
phils6.csp> f(1)
24
phils6.csp> assert not PHIL(1) [F= PHIL(2)
Assertion 5 created (run using :run 5).
```

The command prompt also exposes a number of *commands* which are prefixed with `:`. For example, the type of an expression can be pretty printed using `:type`:

```
phils6.csp> :type head
head :: (<a>) -> a
```

In addition, the command prompt has intelligent (at least in some sense) tab completion. For example:

```
phils6.csp> c<tab>
card     concat
phils6.csp> :<tab>
assertions  debug       graph       help        load        options
processes   quit        reload      run         type        version
```

The interactive prompt will also indicate when a file has been modified on disk, but has not yet been reloaded, by suffixing the file name at the prompt with a `*`.

Available Statements

Expressions can be evaluated by simply typing them in at the prompt. For example, typing `1+1` would print `2`. In order to create a new definition, `let` can be used as follows:

```
phils6.csp> let f(x) = x + 1
phils6.csp> let (z, y) = (1, 2)
```

As with interactive prompts for other languages, each let statement overrides any previous definitions of the same variables, but does not change the version that previous definitions refer to. For example, consider the following:

```
phils6.csp> let f = 1
phils6.csp> let g = f
phils6.csp> let f(x) = g + x
phils6.csp> f(1)
2
```

In the above, even though `f` has been re-bound to a function, `g` still refers to the previous version.

Transparent and external functions can be imported by typing `transparent x, y` at the prompt:

```
phils6.csp> normal(STOP)
<interactive>:1:1-7:
    normal is not in scope
    Did you mean: normal (import using 'transparent normal')
phils6.csp> transparent normal
phils6.csp> normal(STOP)
...
```

New assertions can be created exactly as they would be in a CSP file, by typing `assert X [T= Y`, or `assert STOP :[deadlock free [F]]`. For example:

```
phils6.csp> assert not PHIL(1) [F= PHIL(2)
Assertion 5 created (run using :run 5).
```

Available Commands

There are a number commands available at the command prompt that expose various pieces of functionality. Note that all commands below may be abbreviated, providing the abbreviation is unambiguous. For example, `:assertions` may be abbreviated to `:a`, but `:reload` cannot be abbreviated to `:r` as this could refer to `:run`.

command :assertions
Lists all of the currently defined assertions. For example, assuming that *Dining Philosophers* is loaded:

```
phils6.csp> :assertions
0: SYSTEM :[deadlock free [F]]
1: SYSTEMs :[deadlock free [F]]
2: BSYSTEM :[deadlock free [F]]
3: ASSYSTEM :[deadlock free [F]]
4: ASSYSTEMs :[deadlock free [F]]
```

The index displayed on the left is the index that should be used for other commands that act on assertions (such as *debug*).

command :communication_graph <expression>
Given a CSP expression that evaluates to a process, displays the communication graph of the process, as per *Communication Graph Viewer*.

command :counterexample <assertion index>
Assuming that the given assertion has been checked and fails, pretty prints a textual represenation of the counterexamples to the specified assertion.

command :cd <directory name>
Changes the current directory that files are loaded from. This will affect subsequent calls to *load*.

command :debug <assertion index>
Assuming that the given assertion has been checked and fails, opens the *Debug Viewer* on the counterexample to the specified assertion.

command :graph <model> <expression>
Given a CSP expression that evaluates to a process, displays a graph of the process in the *Process Graph Viewer*. By default, the process will be compiled in the *failures-divergences model* but a specific model can be specified, for example, by typing `:graph [Model] P`, where the model is specified as per *assertions*. For example, `:graph [F] P` will cause the failures model to be used.

command :help
Displays the list of available commands and gives a short description for each.

command :help <command name>
Displays more verbose help about the given command, which should be given without a `:`. For example `:help type` displays the help about the `type` command.

command :load <file name>
Loads the specified file, discarding any definitions or assertions that were given at the prompt.

command :options
See *options list*.

command :options get <option>
Displays the current value for the specified option.

command :options help <option>
Displays a brief description about the specified option, along with details on the range of permitted values.

command :options list
Lists all available program options and prints a brief description and the current value for each option. See *Options* for details on the available options.

command :options reset <option>
Resets the specified option to the default value.

command :options set <option> <value>
Sets the specified option to the given value, displaying an error if the value is not permitted.

command :probe <expression>
Given a CSP expression that evaluates to a process, explores the transitions of the process using *Probe*.

command :processes
Lists all currently defined processes, including functions that evaluate to processes.

command :processes false
Lists all currently defined processes but, in contrast to `processes`, does not include functions that evaluate to processes.

command :profiling_data
If `cspm.profiling.active` is set to `On` and the current file has been loaded/reloaded since this option was activated, this will display profiling data about every function that has been executed since the file was loaded. For example, suppose the following sequence of commands had been executed at the prompt:

```
phils6.csp> :option set cspm.evaulator.profiling On
phils6.csp> :run 0
phils6.csp> :profiling_data
```

Then any profiling data that was generated by executing the first assertion (which would include any function calls made by any function executed as part of the first assertion) in the file would be displayed.

For an explanation of the output format see *Profiling*.

command :quit
Closes FDR.

command :reload
If no file is currently loaded then this resets the current session to a blank session, discarding any definitions or assertions that were given at the prompt. Otherwise, if a file is loaded, then this loads a fresh copy of the file, again discarding any definitions or assertions that had been given at the prompt.

command :run <assertion index>
Runs the given assertion. This consists of two phases; in the first phase the specification and implementation of the given assertion are *compiled* into state machines whilst in the second phase the assertion itself is checked. Presently, no further commands may be evaluated until the first phase has completed.

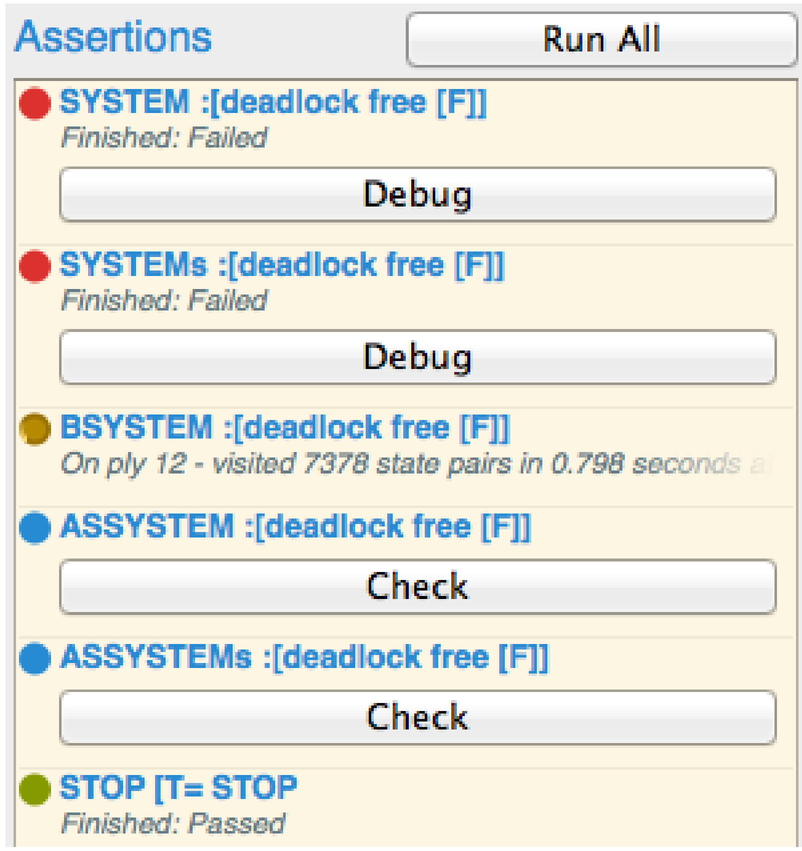
command :statistics <assertion index>
Assuming the given assertion is either running or has already completed, this displays various statistics relating to the assertion including: the number of states visited, the number of transitions visited, the amount of time required to complete the check, and the amount of memory used.

command :structure <expression>
Given a CSP expression that evaluates to a process, displays the compiled structure of the process, as per *Machine Structure Viewer*.

command :type <expression>
Prints the type of the given CSP expression. For example, `:type STOP` prints `STOP :: Proc`.

command :version
Displays the full version number of the currently running FDR.

2.2.2 The Assertion List



The top-right of the *session window*, illustrated to the right, displays the list of assertions that have been defined. This includes all assertions defined in the source file, together with all of those defined *in the session*, in the order of definition. An assertion may be run by clicking the appropriate *Check* button. Alternatively, the *Run All* button may be clicked, which will run all of the un-checked assertions, in parallel. If an assertion has been run and fails, the *Debug* button may be pressed, which will launch the *Debug Viewer*.

Whilst an assertion is being checked, the progress of the check is displayed inline, just below the title of the assertion. The status of the assertion is indicated by the small circle next to the title of each assertion. The colours indicate the following:

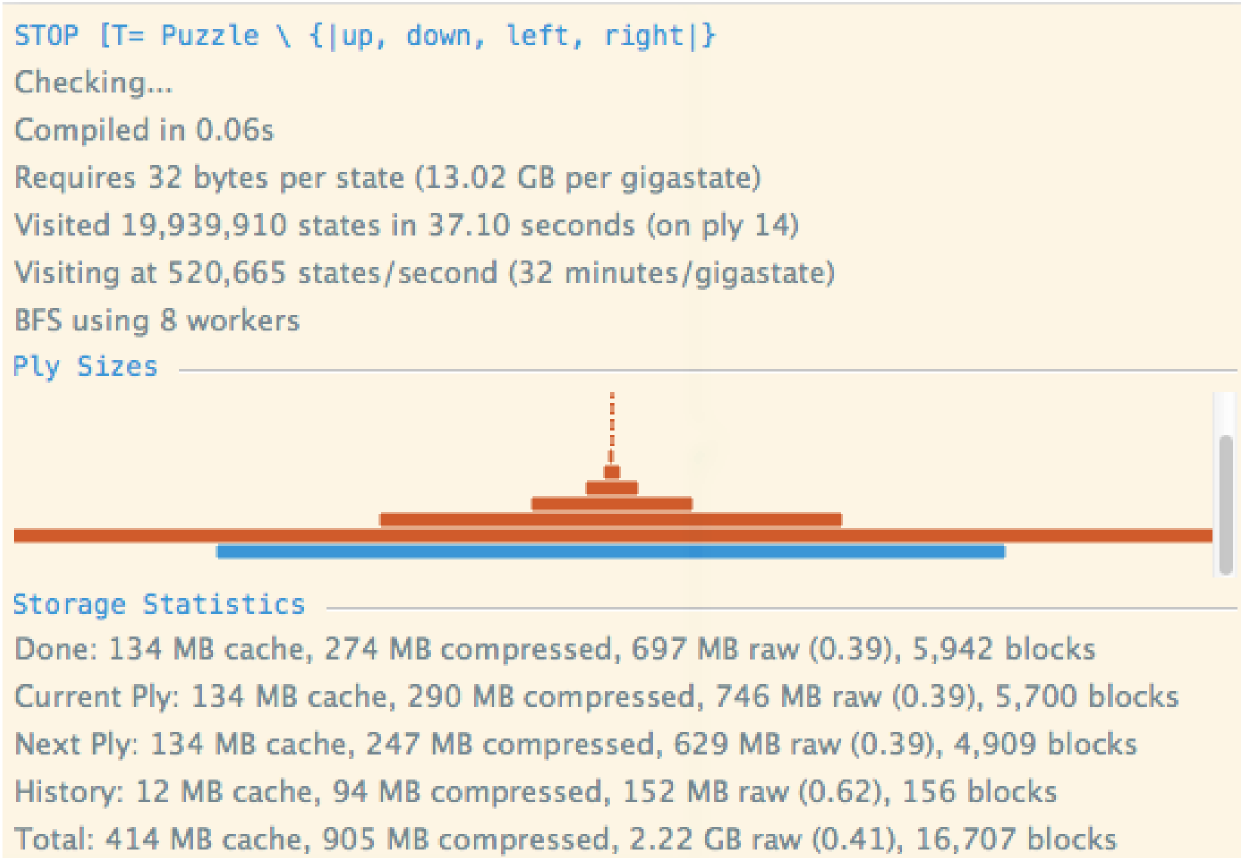
Blue The assertion has not yet been run.

Yellow The assertion is currently being checked.

Red The assertion has been checked and has failed.

Green The assertion has been checked and passed.

Clicking the info button (with a question mark) on an individual assertion displays a window that with various performance statistics about the refinement check. For example:



This window displays, in descending order of display:

Compilation Time The time taken to *compile* the assertion.

Storage Requirements The number of uncompressed bytes to store each node pair encountered during the check. The total amount of storage required is therefore proportional to the number of visited states times this figure, multiplied by the average achieved compression ratio (see below).

Visited States The number of states visited, and the current ply of the breadth-first search.

Number of Workers The number of workers used in the parallel breadth-first search (this can be configured using `refinement.bfs.workers`).

Ply Sizes The relative size of each of the plys in the breadth-first search is indicated by the width of the bars. The absolute size of a ply can be viewed by hovering over a particular ply in the search. The current ply of the search is drawn in blue.

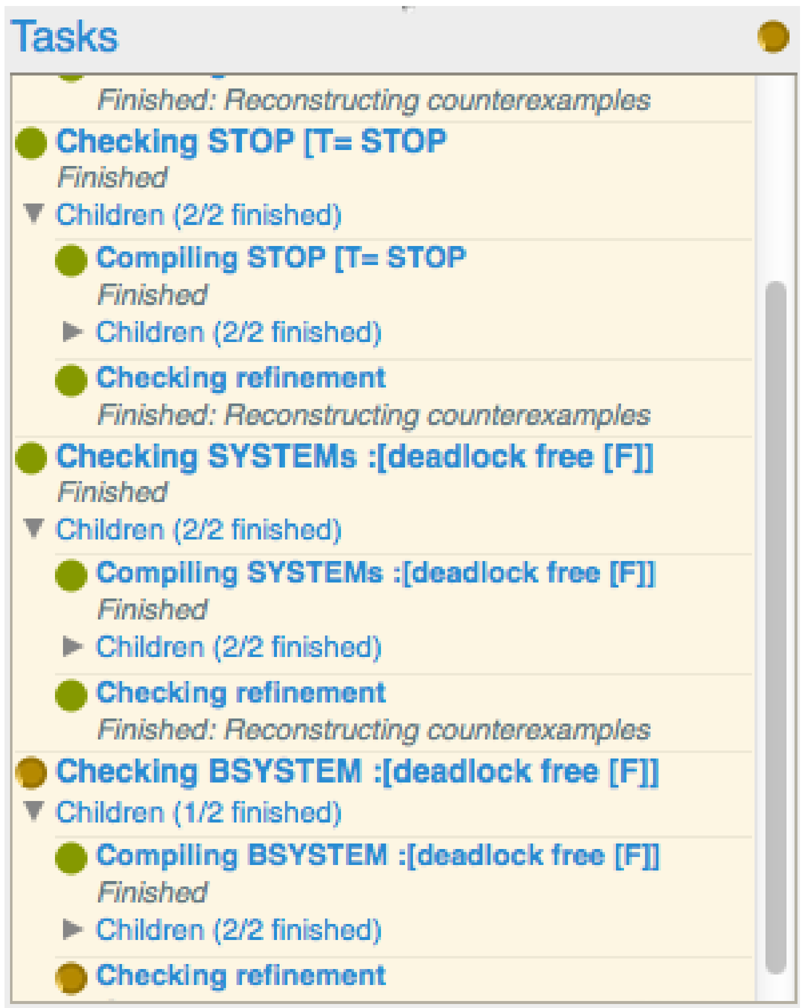
Storage Statistics The total amount of memory, allocated for each type of block-level storage in the search. The cache figure is the amount of memory used to store uncompressed blocks in memory. Compressed refers to the amount of memory allocated for storing compressed blocks whilst raw indicates the amount of memory that would have been required to store the blocks uncompressed. Finally, the achieved compression ratio is given.

Storage statistics are provided for each of the main block stores. Done is the store used to store blocks used by the set of visited states. Current level is the store used to store blocks used by the set of states to visit on the current ply of the search. Next level is the store used to store block that will be visited on the next ply whilst history is the store used to store blocks that indicate how node pairs were reached (for purposes of counterexample reconstruction). Blocks is the number of memory blocks allocated to store data (note that they are of a fixed size). Total is simply the sum of the above figures.

Thus, the total amount of block-level storage required is equal to the total cache size plus the total size of the compressed store (although this will not include the cost to store state machines and other miscellaneous data structures).

For more information on what these statistics represent see *Compilation* and *Refinement Checking*.

2.2.3 The Task List

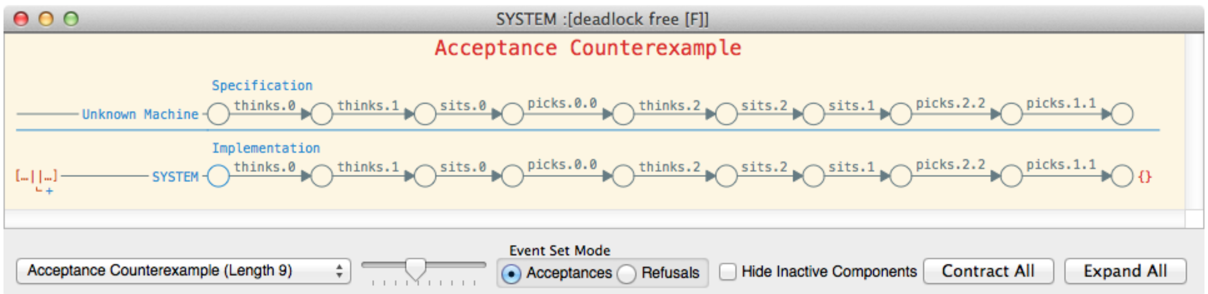


The bottom-right of the *session window*, illustrated to the right, displays the list of *tasks* that are currently running. A *task* is any long-running job performed by the GUI, and includes items like checking refinements, graphing processes or evaluating expressions. The list of tasks is hierarchical, allowing the dependencies between tasks to be tracked. For example, if a task is a *Checking Refinement* task, then the task will have two children; a *compiling* task and a *checking refinement* task (see *Compilation* for more details about tasks during compilation).

The status of each task, if available, is displayed inline, below the task title. Any child tasks of the parent are displayed below the status, in a section that can be expanded or contracted by clicking the triangle. The circular indicator next to the task title indicates the task status; if it is yellow then the task is running, green indicates that the task completed successfully whilst red indicates that the task fails. Hovering over the circular indicator will display the runtime of the task.

2.3 Debug Viewer

The debug viewer, as shown below, allows a counterexample to a refinement assertion to be viewed. In particular, it attempts to explain how the implementation evolved into a state where it could perform a behaviour that was prohibited by the specification. Conceptually, the debug viewer is a table where the rows represent behaviours of particular machines whilst the columns represent events that are synchronised.

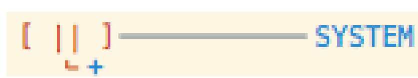


The above counterexample, which we use as an example throughout this section, is to the first assertion of an edited version of *Dining Philosophers* where `N = 3` (for ease of exposition).

> **Warning:** Note that the counterexample that is displayed will be picked somewhat non-deterministically. See *Counterexamples* for further details.

2.3.1 Behaviours

The individual rows within the counterexample represent the *behaviour* of a machine. A behaviour of a particular machine is a sequence of states that the machine transitioned through, along with the events that it performed. In the default view of a counterexample, the first row represents the behaviour of the specification machine, whilst the second row represents the behaviour of the implementation.

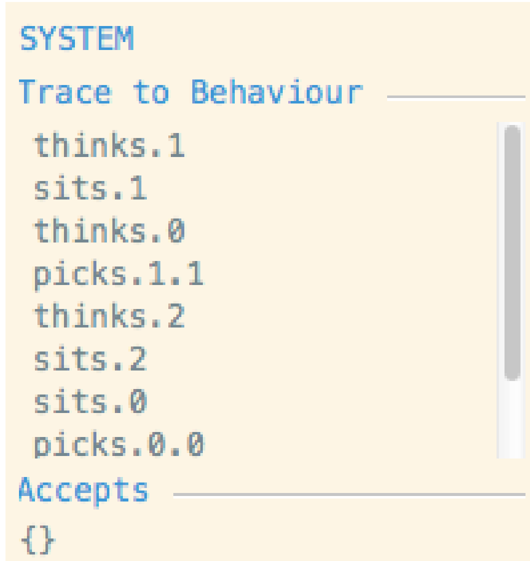


On the left of a behaviour row, two items of information are displayed, as shown to the right. The first is the *operator* that the machine represents, whilst the second is the name of the machine, if one was given to it in the script. Note that the operator is only displayed if the machine is a *high-level* or a *compressed* machine. Hovering over the operator will reveal the operator type and its arguments. For example, hovering over the operator in the image to the right displays the following.

```
Alphabetised parallel with process alphabets:
    1: {thinks.0, sits.0, eats.0, getsup.0, picks.0.0, picks.0.1,
        picks.2.0, putsdown.0.0, putsdown.0.1, putsdown.2.0}
    2: {thinks.1, sits.1, eats.1, getsup.1, picks.0.1, picks.1.1,
        picks.1.2, putsdown.0.1, putsdown.1.1, putsdown.1.2}
    3: {thinks.2, sits.2, eats.2, getsup.2, picks.1.2, picks.2.0,
        picks.2.2, putsdown.1.2, putsdown.2.0, putsdown.2.2}
```

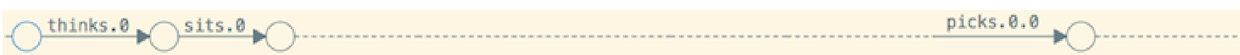
which indicates that the operator in question is an alphabetised parallel operator with 3 processes, each of which has the corresponding alphabet.

Hovering over the machine name will display extra information in the right pane that includes the full name (if it had to be abbreviated), a scrollable view of the trace (which is particularly useful if the trace is long), and a textual representation of the behaviour, if the row represents an error. For example, hovering over the `SYSTEM` name in the above displays the following:



This indicates the trace that `SYSTEM` performed and the fact it accepted `{}` at the end of trace, violating the specification.

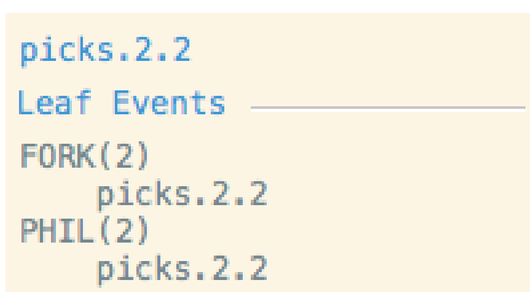
The second component, which is the main section, shows the trace that this machine performed:



Each of the circles represents a state that the machine reaches. The edges between the states are labelled by the events that the machine performs. Named states are drawn in blue and, when hovering over a state, identical states of the machine are highlighted in red. A dashed edge indicates that the machine performed no event. Green edges indicate that the machine was restarted by the event (e.g. consider `P = X ; P`).

Clicking on a state will reveal several items of information about it including: the state name (if any); the available events of the state; the minimal acceptances of the state. It also allows *Probe* to be launched to inspect the state's transitions and, providing the machine does not have too many states, allows the *Process Graph Viewer* to be launched on the state. If the graph view is launched, then in addition to displaying the usual graph of the machine, the behaviour in the selected row will be highlighted (see *Behaviours* for more information).

Clicking on an edge will display several extra pieces of information in the right pane. In particular, if the event is a tau that resulting from a hiding it will reveal what the inner event is. Further, it will detail what events the leaf processes perform in order to perform the overall event. For example, hovering over the last event in the above counterexample (i.e. picks.1.1) in the row labelled `Implementation` will display the following in the right pane:

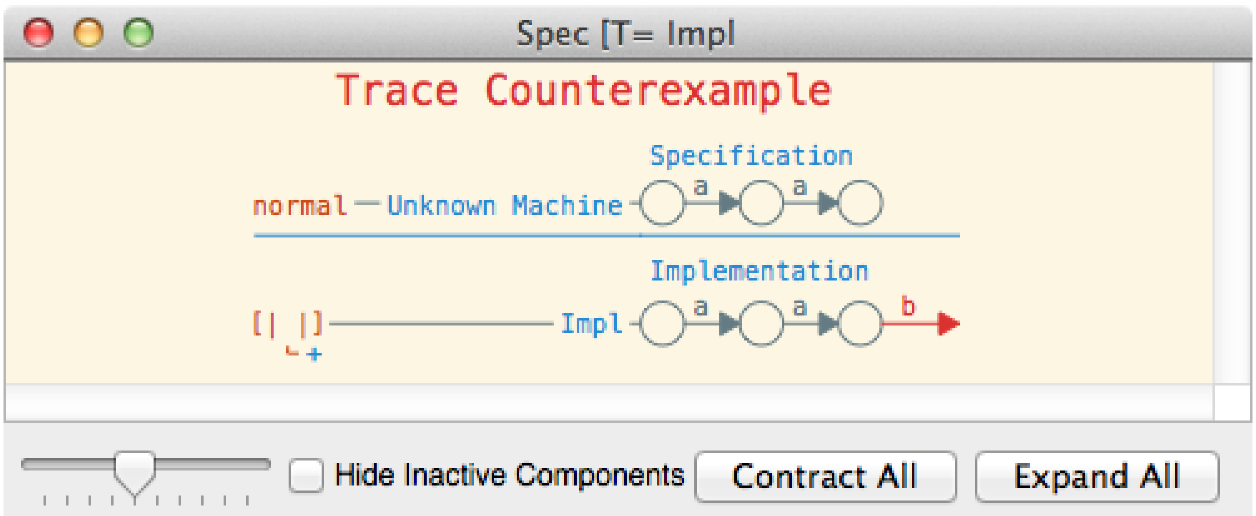


This indicates that `FORK(2)` and `PHIL(2)` both performed `picks.2.2` in order to produce the overall `picks.2.2` event.

The third and rightmost component of a behaviour indicates how this machine contributes to the prohibited behaviour of the specification. For example, consider the following script:

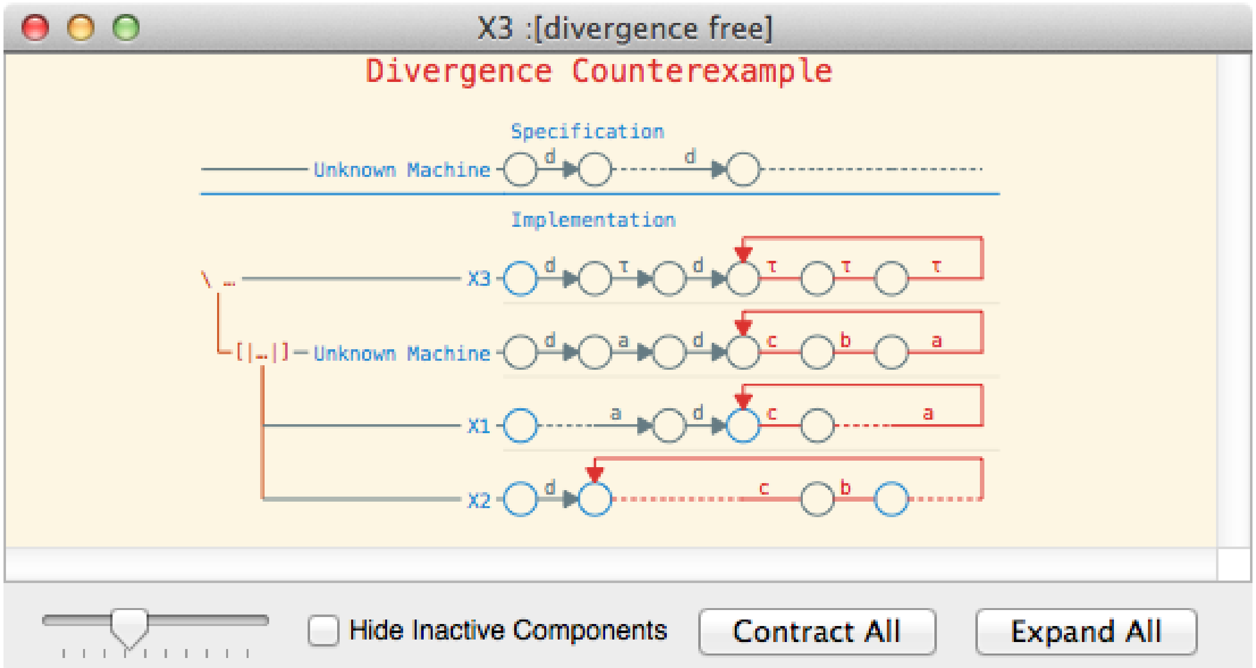
```
channel a, b, c

Left = a -> c -> STOP
Right = a -> b -> STOP

Impl = Left[[c <- b]] [| {b} |] Right
Spec = a -> Spec

assert Spec [T= Impl
```

The above assertion will fail as `Impl` can perform the trace `a, a, b`, which is not a trace of the specification. This results in the following counterexample:



The rightmost component of the implementation behaviour indicates that the process attempted to perform the event `b`, which was disallowed by the specification (hence it is an error and is thus drawn in red).

Acceptance errors are indicated by giving the erroneous acceptance. Hence, in the *above case*, as shown to the right, the empty acceptance is shown as the machine deadlocks after the given trace. It is also possible to view maximal refusals rather than acceptances by selecting *Refusals* in the *Event Set Mode* in the bottom left hand corner of the debug viewer (as shown in the *above case*). The maximal refusals are relative to the set of events the machine in question can perform.



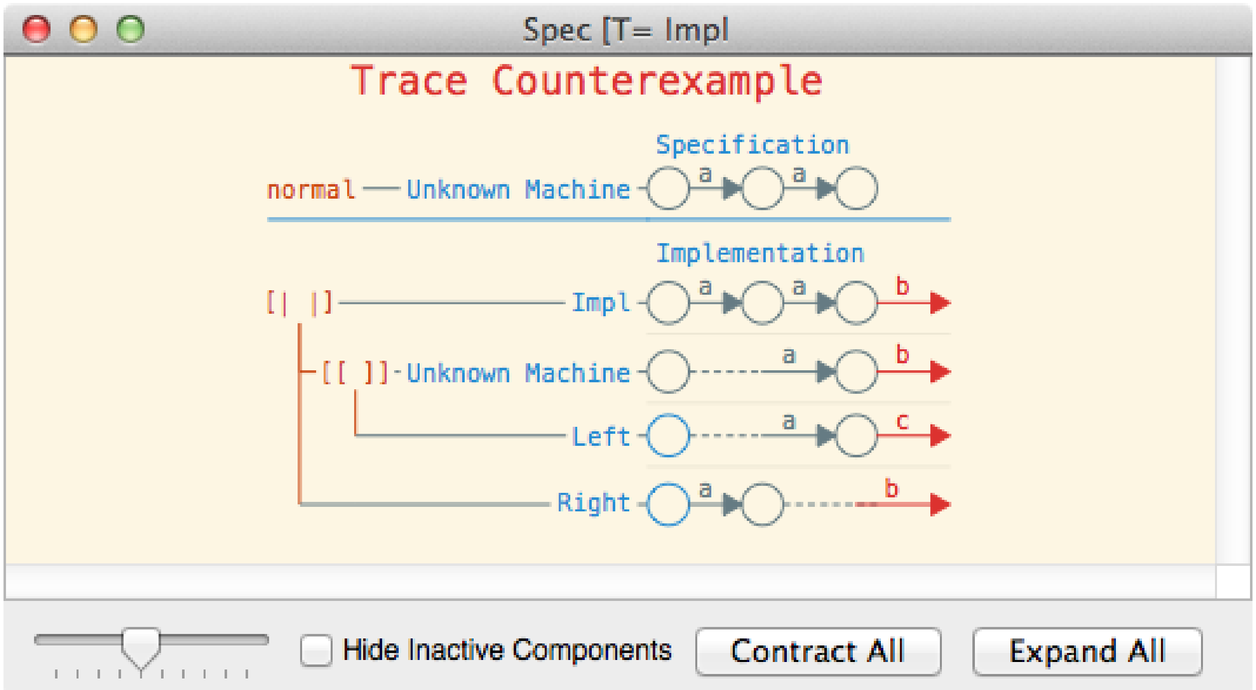
Divergence errors, or loop errors, are indicated by drawing the trace that repeats in red. For example, in the above screenshot, the process `X3` can diverge by repeating a sequence of three taus, which actually is caused by a machine repeating the trace `c, b, a`, which are then all hidden.

Divergence errors may also be indicated by simply stating that a given state diverges, as indicated to the right. This occurs, for example, when checking `normal(div)` (for a suitable definition of `div`) for livelock freedom. For more information see *Generalised Low-Level Machine*.

2.3.2 Dividing Behaviours

If a behaviour is a behaviour of a *high-level* machine or a *compressed* machine then it may be *divided* into behaviours of its components. This can help significantly with working out either how a machine performed a particular event, or why the machine can perform the behaviour that was prohibited by the specification. A behaviour can be divided either by double clicking the operator of the machine, double clicking the plus button under the operator or double clicking any blank space in the middle section of the behaviour (i.e. the section with the states and events).

For example, consider the simple example script shown *above*. Clicking *Expand All* reveals the following:



The left portion of the diagram shows the structure of the machines. In particular, we can deduce that the implementation process (i.e. `Impl`) is a parallel composition of `Right` and a renaming of `Left` (the operator arguments can be viewed by *hovering* over the operators themselves).

We can also deduce how the events synchronise together, as events in the same column indicate that they are synchronised. Thus, in the above we can deduce that the first `a` event occurred because `Right` performed an `a`, whilst the second occurred because `Left` performed an `a`. In both cases we can see that the event did not synchronise with any other event, due to the dashed edges. The diagram also allows us to deduce why the `b` was performed. In particular, the `b` must have occurred due to `Right` and the renamed copy of `Left` synchronising on a `b`. Further, this was possible because `Left` performed a `c` that was renamed to a `b`.

2.3.3 Navigating The Debug Viewer

The debug viewer can be panned (or moved) by clicking and dragging whilst scrolling will zoom in or out. Further, double clicking will zoom in by a constant amount. Alternatively, if gestures are supported, zooming can be accomplished by pinching (as on a touch screen device) and panning is instead done by scrolling. In either case, the zoom level can be modified by moving the slider in the bottom left.

Hovering over the title of the counterexample (i.e. Acceptance Counterexample) in the above screenshot displays a textual description of the counterexample. For example, in the above case the description is:

```
After performing the trace:
   thinks.0, sits.0, thinks.2, picks.0.0, thinks.1, sits.2, sits.1,
   picks.1.1, picks.2.2
the implementation offers the set of events:
   {}
which is not a superset of one of the specification acceptances:
   {thinks.0}, {sits.0}, {eats.0}, {getsup.0}, {picks.0.0},
   {picks.0.1}, {picks.2.0}, {putsdown.0.0}, {putsdown.0.1},
   {putsdown.2.0}, {thinks.1}, {sits.1}, {eats.1}, {getsup.1},
   {picks.1.1}, {picks.1.2}, {putsdown.1.1}, {putsdown.1.2},
```

(continues on next page)

(continued from previous page)

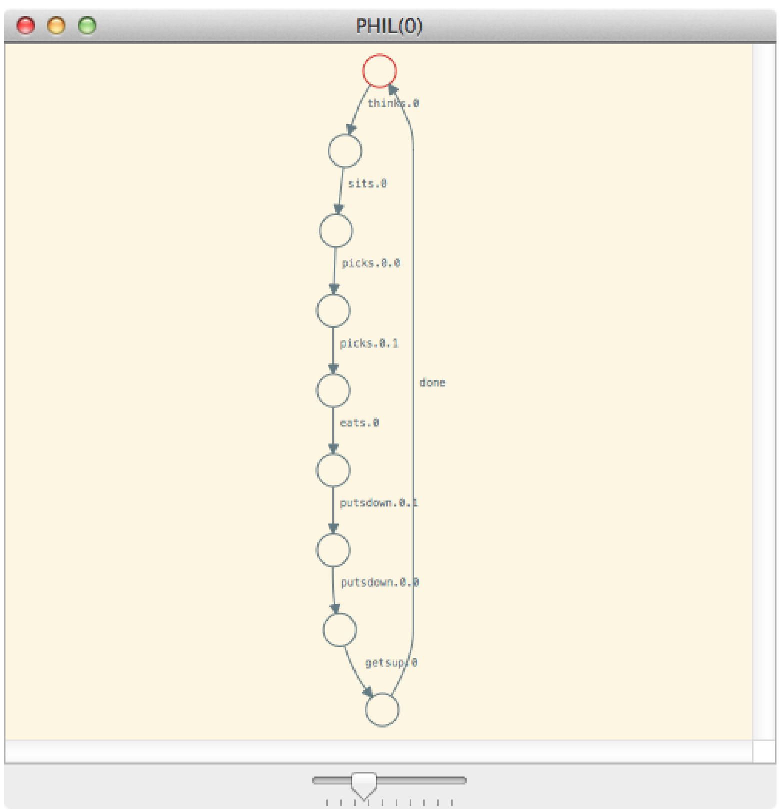
```
{thinks.2}, {sits.2}, {eats.2}, {getsup.2}, {picks.2.2},
{putsdown.2.2}.
```

Checking *Hide Inactive Components* will elide any rows in which the machine does not contribute to the overall behaviour (i.e. it is inactive). Unchecking *View Taus* will elide any column in which only taus are visible. Clicking *Expand All* or *Contract All* will result in all behaviours being *divided* recursively, or contracted recursively, respectively. If `refinement.desired_counterexample_count` is set to a value greater than 1 and FDR is able to find multiple counterexamples, then the displayed counterexample can be selected using the drop-down on the bottom left corner of the debug viewer, as shown *above*. Clicking on a cell in the debug viewer will higlight the row and column corresponding to that cell.

2.4 Process Graph Viewer

FDR is also capable of displaying graphs of processes, rendered using GraphViz. This can be very useful for visualising small processes, but once the graph grows beyond a few hundred states or transitions, it quickly becomes unmanageable (and can cause GraphViz to consume vast amounts of memory).

For example, if *Dining Philosophers* is loaded, then typing `:graph PHIL(0)` into the `prompt` will display the following window:

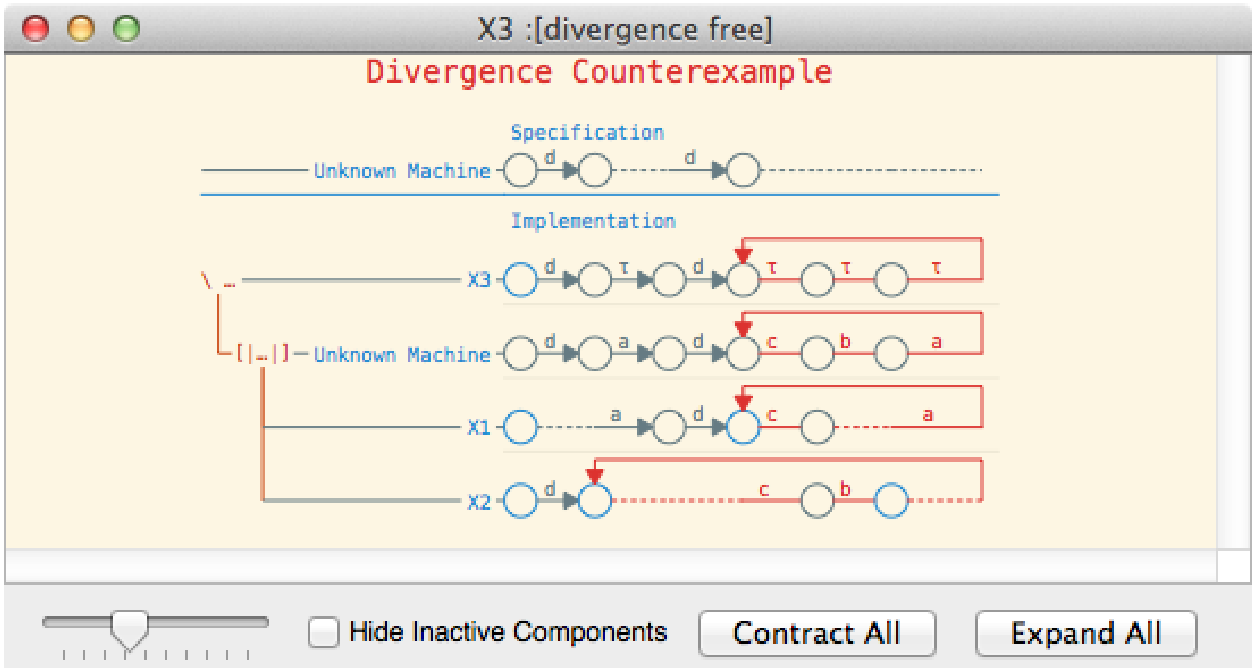


The graph is rendered in the obvious way. Each state of a process is rendered as a circular node, whilst the transitions are drawn as labelled edges. The initial node of the machine is drawn in red, whilst named nodes are drawn in blue (i.e. if a node corresponds to a named process). Clicking on a node will, as with the *Debug Viewer*, display some information about the node in the right-hand pane, including the available events, its minimal acceptances, and the node name, if one is available. It also allows *Probe* to be launched from the selected state.

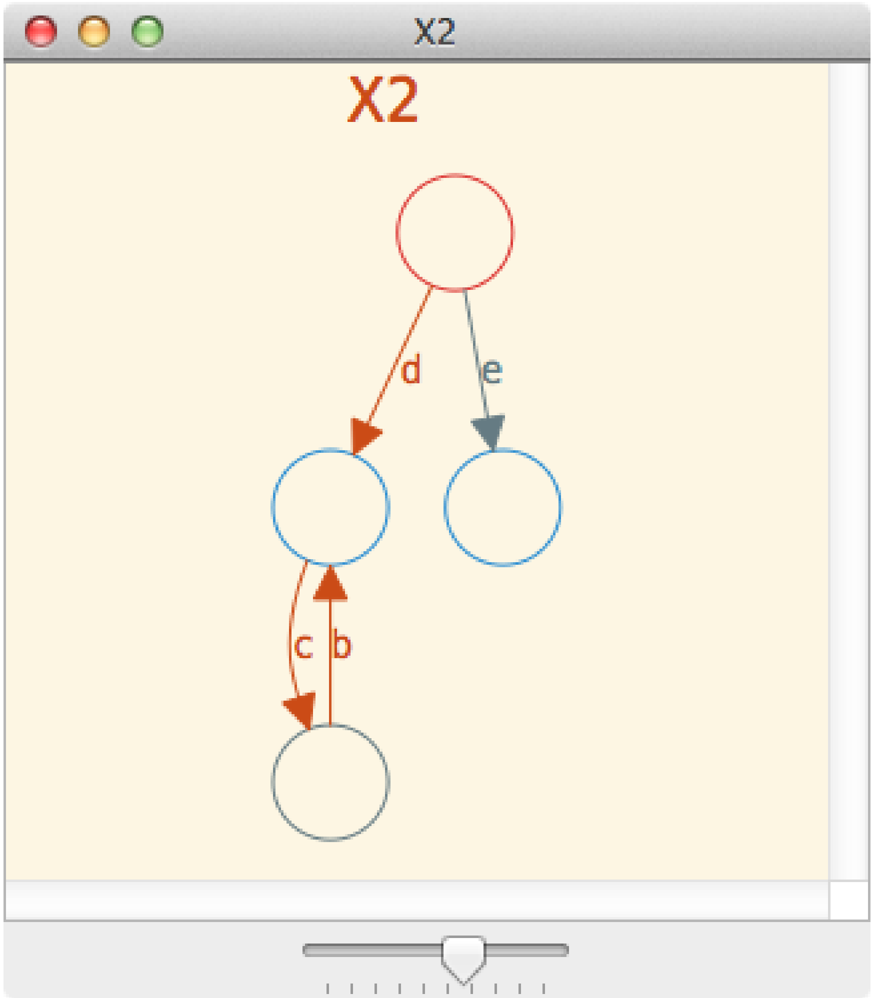
The graph can be navigated exactly like the *Debug Viewer*. Thus, it can be panned (or moved) by clicking and dragging whilst scrolling will zoom in or out. Double clicking will zoom in by a constant amount. Alternatively, if gestures are supported, zooming can be accomplished by pinching (as on a touch screen device) and panning is instead done by scrolling. In either case, the zoom level can be modified by moving the slider at the bottom.

2.4.1 Behaviours

If the process graph viewer is launched on a machine or a node from within the *debug viewer*, then the selected behaviour is highlighted on the graph. For example, suppose the following counterexample is being viewed in the debug viewer:



Clicking on *View Graph* in the right pane will display the following graph:



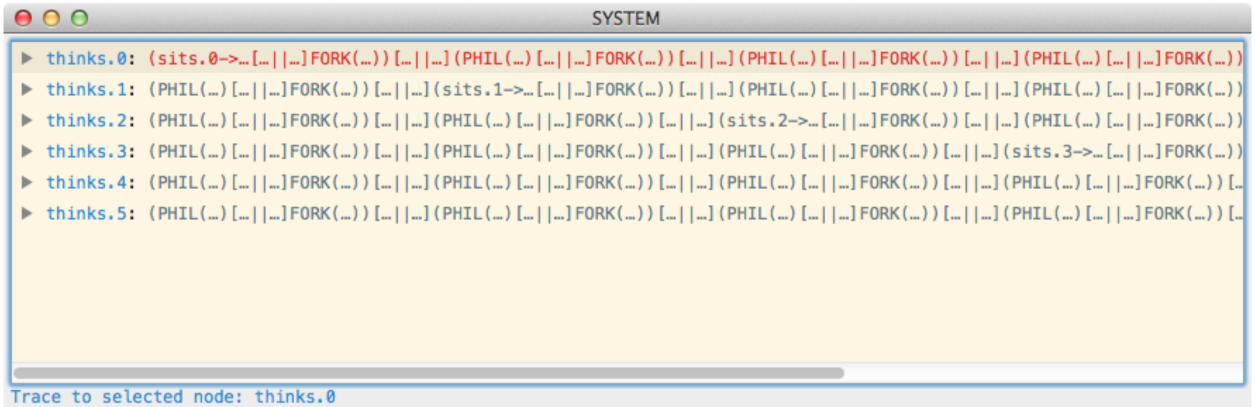
In the above, each transition that formed part of the behaviour of `X2` is highlighted. Hence, the `d` transition to the `c`, `b` loop is highlighted, since this is the behaviour that is being displayed in the debug viewer. In addition, hovering over the machine name will display a textual description of the behaviour. For example, in the above window hovering over `X2` will display the following text:

```
Highlighting the behaviour:
    X2 (Repeat Behaviour):
        Trace: <d>
        Repeats the trace: <c, b>
```

2.5 Probe

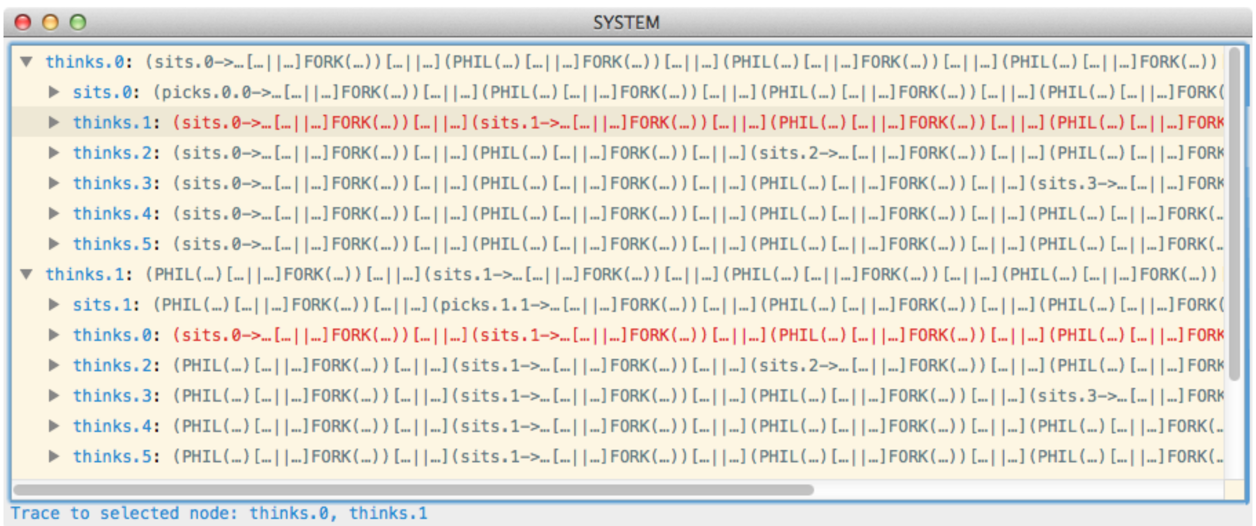
Probe can be used to manually explore the transitions of a process as a tree, which can be very helpful when debugging a process definition. Probe can be launched on any process by typing `:probe P` into the *prompt*. Alternatively, Probe may be launched on any state via a node (such as those in the *Debug Viewer* or *Process Graph Viewer*) by clicking the Probe button.

As an example of how to use Probe, consider loading *Dining Philosophers* and typing `:probe SYSTEM` into the prompt; this will result in the following window being displayed:



In the above, the transitions of the initial state (i.e. `SYSTEM`) are shown in sorted order of event. A line of the form `ev: P` means that the process can perform the event `ev` and evolve into the process `P`. Further, each of these lines can be expanded to reveal the transitions of the resulting process, yielding a tree of transitions. To expand the transitions of a process either double click on the row, click on the triangle to the left of the row, or use the right arrow keyboard button. To contract a row, either double click on the row, click on the triangle or use the left arrow keyboard button.

For example, if we expand the first and the second rows, we see the following view:

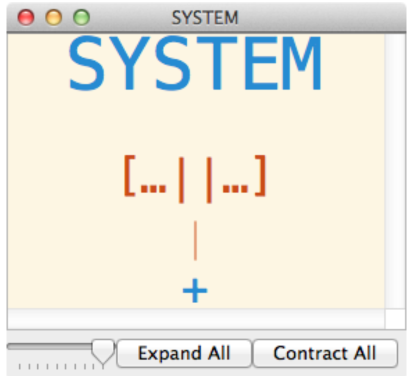


Probe also highlights syntactically equivalent nodes in red. For example, in the above screenshot there are two rows highlighted as the state reached by `SYSTEM` performing `thinks.0` and then `thinks.1` is identical to the state reached by performing the events in the opposite order.

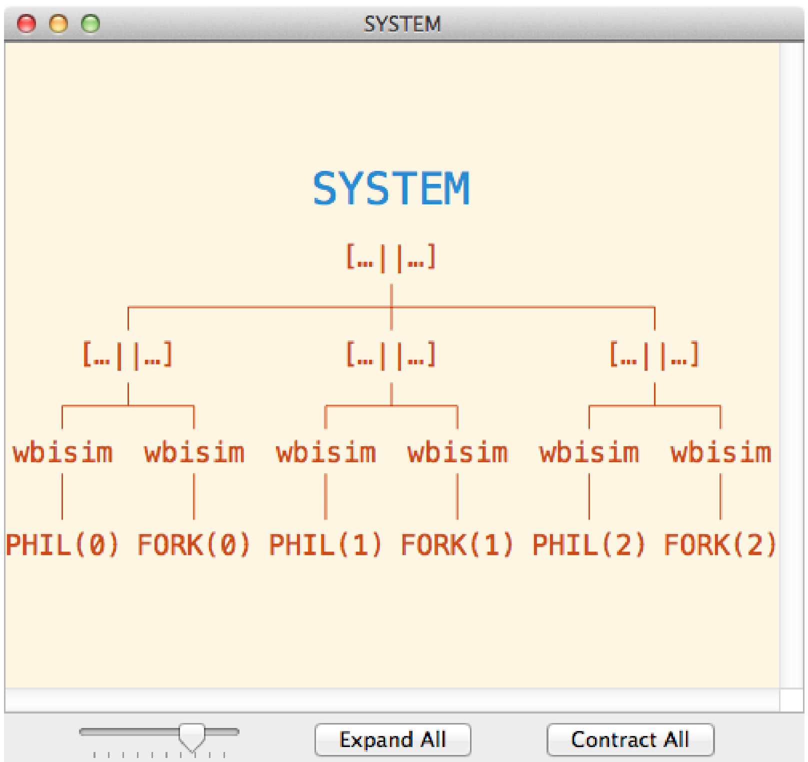
The trace required to reach the currently selected node is displayed at the bottom of the window.

2.6 Node Inspector

When debugging a counterexample or when using probe, FDR allows an individual node (i.e. state) to be inspected in order to allow the actual structure of the node to be understood. The node inspector can be launched on any state in *Probe* by selecting a row, right-clicking and then selecting *Inspect Node* in the resulting menu. To launch the node inspector from the *Debug Viewer*, hover over the desired state and select *Inspect Node* in the window that appears. For example, suppose the *counterexample* shown as part of the *Debug Viewer* is being viewed and that the very first node of the implementation row is hovered over (i.e. the blue node on the `SYSTEM` line). Clicking *Node Inspector* in the resulting window will result in the following being displayed:



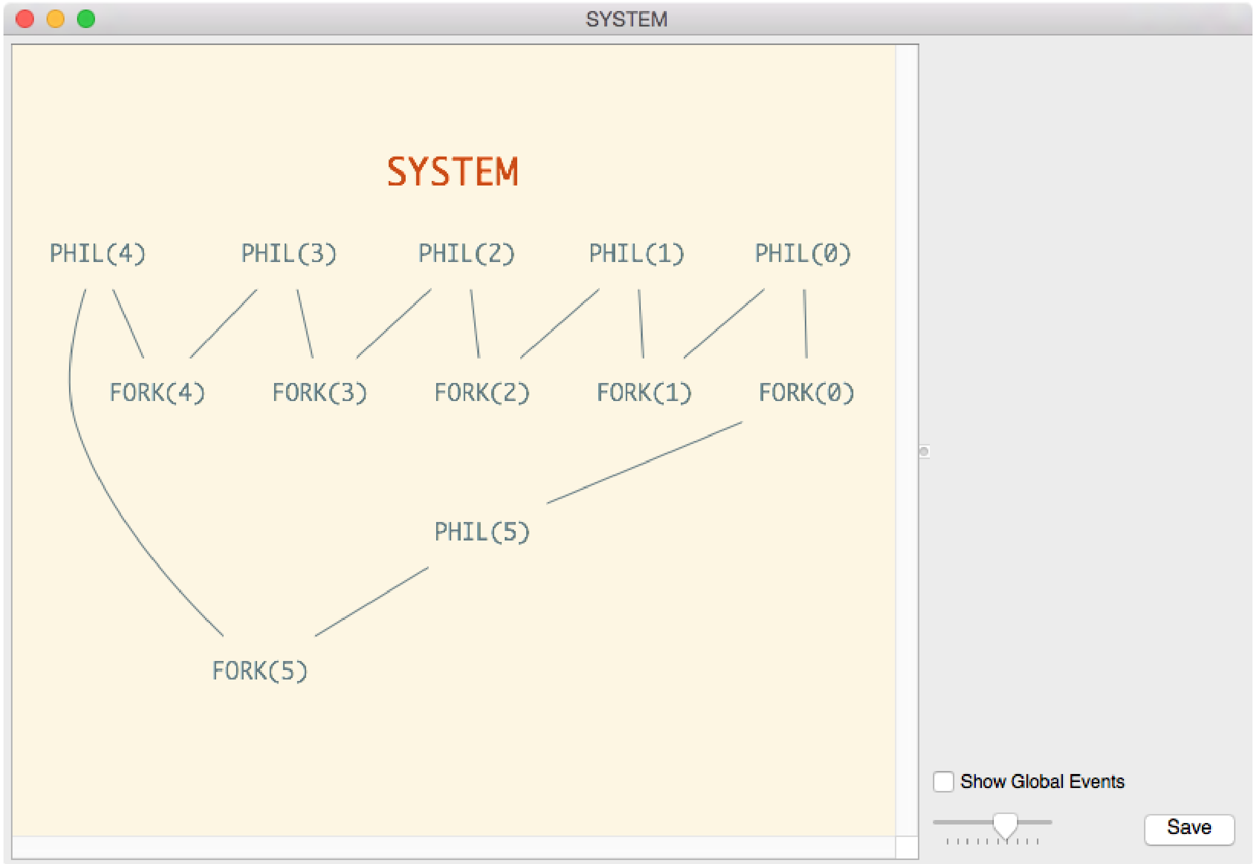
Clicking on *Expand All* will result in the full node structure of the selected node being displayed, as shown below:



The above indicates that the selected node is an alphabetised parallel of three alphabetised parallels, the first of which is a parallel of the weak bisimulation of `PHIL(0)` and the weak bisimulation of `FORK(0)`. Hovering over the nodes in the window will display information various information, such as the synchronisation sets in use. Unknown nodes are indicated using a `?`. Note that hovering over the node will reveal some information about it, such as which state machine the node is a member of (if known).

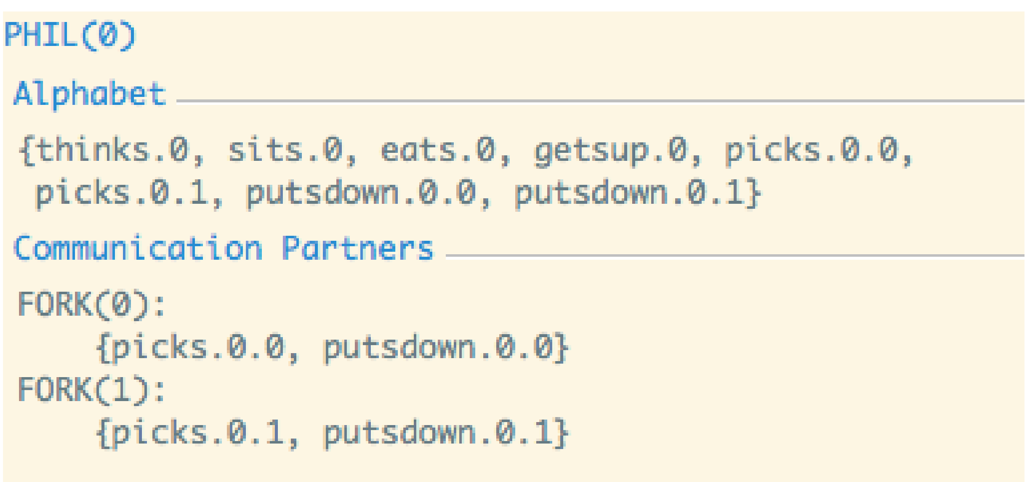
2.7 Communication Graph Viewer

The *communication graph* of a process is a graph that indicates which processes communicate with which other processes. Thus, it is a graph in which the vertices are subprocesses of the top-level process, and in which there is an edge between any two processes if they can both perform some event. For example, if *Dining Philosophers* is loaded, then typing `:communication_graph SYSTEM` into the *prompt* would show the following window:



In the graph, there is a vertex for each `PHIL` process and a vertex for each `FORK` process. Further, there is a edge between each `PHIL` and its neighbouring `FORK` processes, since they can both perform `pickup` and `putsdown` events.

Clicking on a node in the graph will change the contents of the right-hand pane. For example, selecting `PHIL(0)` will change the contents of the right-hand pane to the following:

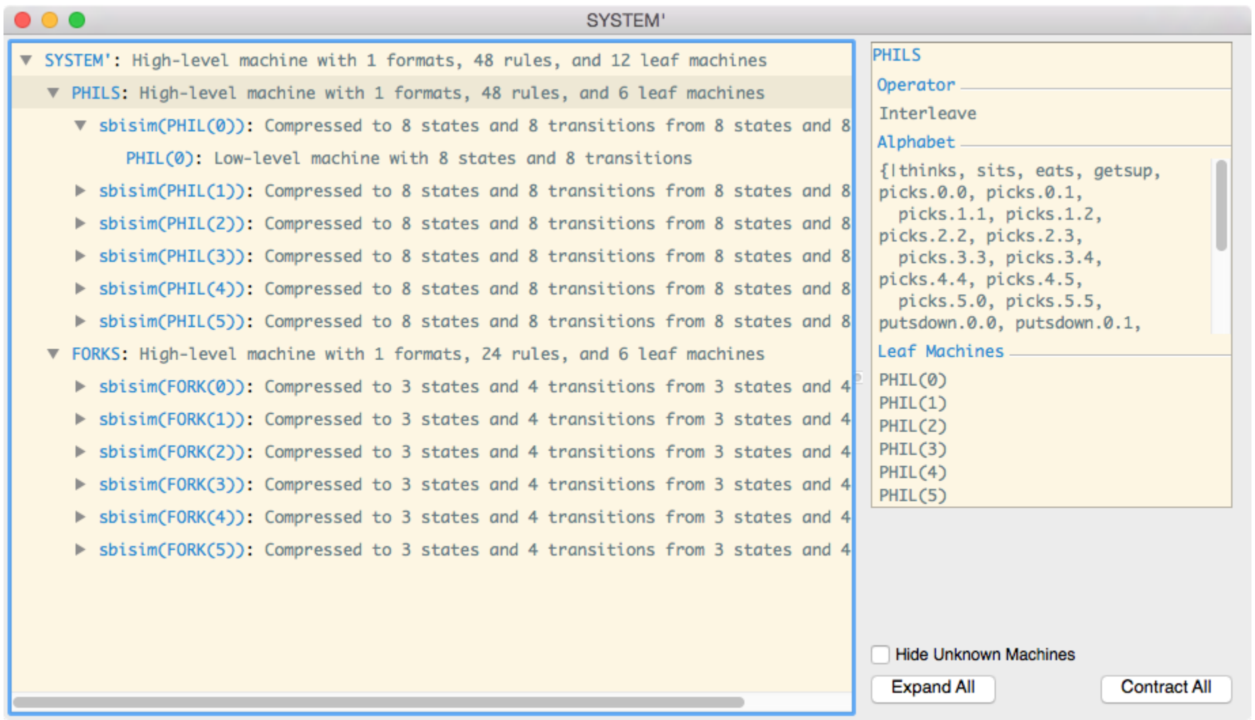


This indicates the name of the process that is selected, along with the processes' alphabet (i.e. the set of events it can actually perform). Further, the pane also displays *communication partners*, which are the other processes with which the process communicates, along with the set of events upon which they synchronise.

The *Show Global Events* option on the bottom right can be used to toggle whether or not a *global event* is considered in the communication graph. A global event is defined as any event that more than 90% of the processes participate in, and common examples are events such as `tock`. If this box is left unchecked, then such events will be elided, which will often result in a reduction in the number of edges in the graph.

2.8 Machine Structure Viewer

The machine structure viewer provides a way of viewing FDR's internal representation of a CSP process. This takes the form of a tree of processes, where the leaves are simple processes, and the internal vertices (i.e. non-leaf nodes) correspond to processes composed of others. For example, if *Dining Philosophers* is loaded, then typing `:structure SYSTEM'` into the `prompt` would show the following window:

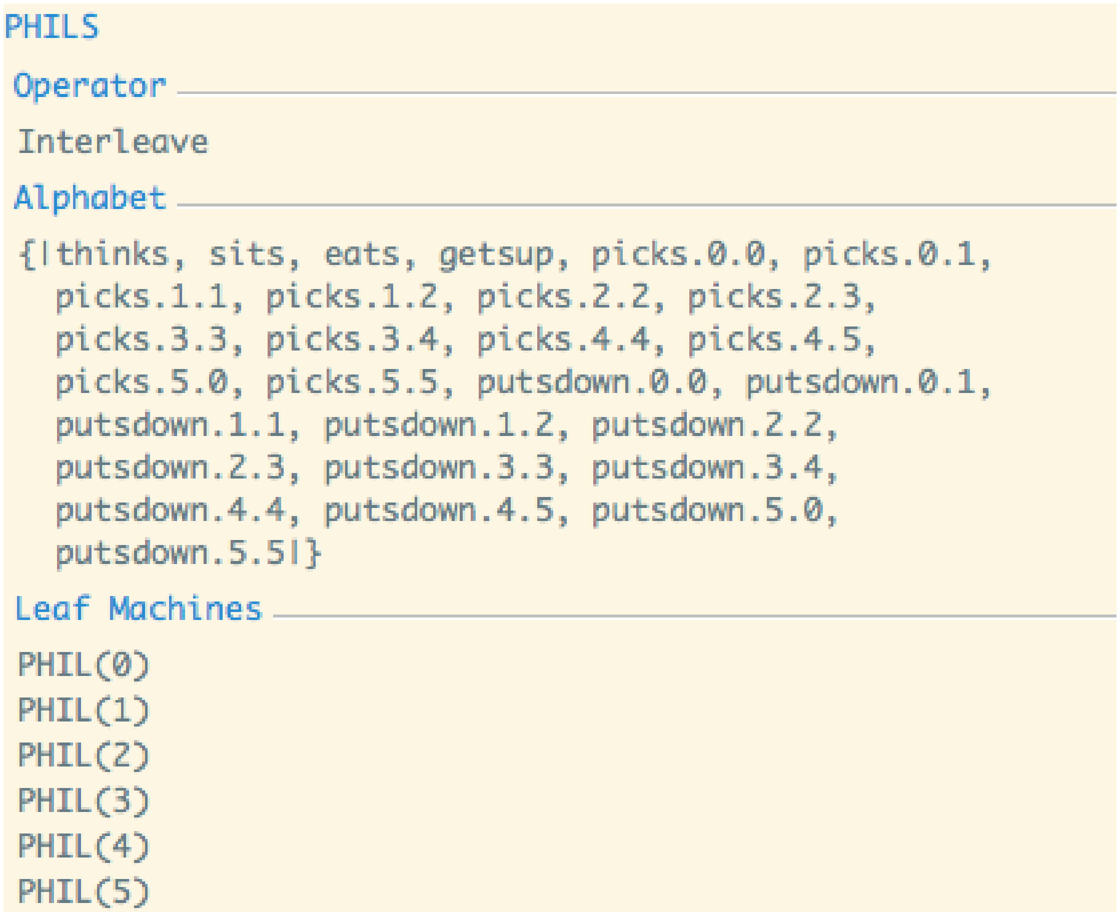


This indicates that the process `SYSTEM'` is composed of two subprocesses, `PHILS` and `FORKS`. Further, `PHILS` is composed of 6 subprocesses, each of the form `sbisim(PHIL(i))` (i.e. `sbisim` applied to `PHIL(i)`). This viewer can be a useful tool for checking that a process has been defined correctly, and for investigating how FDR has represented the process in order to diagnose any performance problems. It is also a useful tool for investigating the effectiveness of *compression*, since it is capable of displaying the number of states both before and after compression.

In general, there are three different types of nodes that will appears in the tree. Exactly which node will appear depends on how FDR internally decides to represent a process.

- High-level machine nodes: these correspond to individual CSP operators, such as `Alphabetised Parallel` and `Hide`, that are applied to any number of subprocesses (depending on the operator). In this case, the node will display the number of formats and rules, which can be a useful indicator of the complexity of a machine (the more formats and rules, the more complex the machine).
- Low-level machine nodes: these will generally correspond to simple recursive CSP processes which FDR decides to compile into a single labelled-transition system, and thus these nodes are not divisible. The number of states and transitions is given.
- Compressed machines: these will appear wherever a compression function, such as `normal` or `sbisim`, is applied. This will indicate how many states and transitions the compression managed to save. This data is used to *estimate* the total number of states and transitions that compression saved.

The right-hand pane displays information about the selected process. For example, if `PHILS` is selected, the information pane displays the following:



This shows information about what CSP operator the process represents (if the node is a non-leaf node), the alphabet of the process (i.e. the set of events it can perform), and the list of processes that are leaf processes underneath this vertex (if the node is not a leaf itself).

2.9 Options

The options are catagorised according to what they affected below.

2.9.1 Compiler

option compiler.check_for_immediate_recursions

Default On

If true, FDR will check to see if any processes of the form `P = P [] STOP` are defined. If this option is not set then such a process will cause FDR to fail at runtime, generally with a segmentation fault. If you know that, by construction, no such processes can be contained within your file then this option can be disabled to reduce the time required to compile processes. Note that this will only have an impact on huge processes, that contain millions of names. On anything smaller the different will be negligible.

option compiler.leaf_compression

Default `sbisim`

The compression that is used for compressing arguments of high-level machines. This may either be `none`, indicating that no compression should be used, or *`sbisim`* or *`wbisim`*, to indicate that the strong or weak bisimulation compression function should be used, respectively.

option compiler.recursive_high_level

Default On

If true FDR will compile processes of the form `P = Q ; P` in an optimised format. It is recommended that this is not disabled, unless it is producing poor results on such a process.

option compiler.reuse_machines

Default On

If true FDR will, at the cost of increased memory usage, re-use named state machines in different assertions. If the same state machine is not being used in more than one assertion then it may make sense to disable this as it will decrease memory use.

2.9.2 Functional Language

option cspm.evaluator_heap_size

Default A default value chosen by FDR.

This option allows for more memory to be allocated for the evaluator up front and can be used to improve performance in some cases. If the value is suffixed by `K`, `M`, or `G`, it is a size in kilobytes, megabytes, or gigabytes, respectively. Otherwise, it is a size in bytes.

Warning: This option will not take effect until FDR is restarted.

option cspm.profiling.active

Default Off

If set to On, then simple profiling statistics are collected that detail how many times each function has been called, and by which other functions. For further details on profiling see *Profiling*. Note that turning on this option will reduce the performance of the evaluator and therefore, this option should only be kept activated for as long as necessary.

Warning: This option will not take effect until *`load`* or *`reload`* is executed.

option cspm.profiling.flatten_recursion

Default On

This option affects how profiling statistics are reported (and is therefore only relevant assuming *`cspm.profiling.active`* is On). If this option is `Off`, then recursive functions, such as:

```
f(0) = 0
f(x) = f(x-1)
```

will result in a hierarchy that is as deep as the longest possible chain of recursive calls, causing profiling to be slower and making the data more accurate, but potentially more difficult to interpret. See *Profiling* for further details.

> **Warning:** This option will not take effect until *load* or *reload* is executed.

option cspm.runtime_range_checks

Default On

If Off, then FDR will not check to see if the values that are dotted with channel or datatype constructors are in the defined sets. For example, consider the following script:

```
channel c : {0..1}
datatype X = Y.{0..1}
```

If this option is On, then FDR would throw an error if `c.2` or `Y.2` were evaluated. Turning this option off will suppress these errors.

This option should only be used for performance reasons in circumstances in which you are confident that it would be impossible for this error to be produced. In particular, whilst turning this option off will never cause FDR to emit further errors, it is possible to construct processes that are incorrect. For example, in theory, hiding `Events` should result in a process that can perform no visible events, but `(c.2 -> STOP) \ Events` is equivalent to `c.2 -> STOP`, since `c.2` is not a valid event and is therefore not in `Events`.

> **Warning:** This option will not take effect until *load* or *reload* is executed.

2.9.3 Graphical User Interface

option gui.close_windows_on_load

Deafult Off

If set to On, then whenever a *load* or *reload* command is executed, all windows relating to the current session will be closed.

option gui.console.history_length

Default 100

The number of history entries to keep in the *command prompt*. Any integer strictly greater than 0 is permitted for this option.

2.9.4 Refinement Checking

option refinement.bfs.workers

Default The number of available cores.

The number of workers to use for a refinement check that is using breadth first search. This may be set to any value strictly greater than 0. If 1 core is selected then the algorithm used is identical to the algorithm of FDR2 (at least conceptually). If more than 1 worker is required then a parallelised version of breadth first search is used, as explained in *Refinement Checking*.

option refinement.cluster.homogeneous

Default false

If set to true, FDR will assume that the nodes in the cluster are homogeneous (i.e. each node has the same number of cores of the same speed), rather than trying to calculate the speed of each machine.

If you are operating on a homogeneous cluster, it is strongly recommended that this flag should be set, as otherwise the cluster may end up being imbalanced.

option refinement.desired_counterexample_count

Default 1

The number of counterexamples that FDR should attempt to find during a refinement check. Note that FDR may not be able to find this many counterexamples, but will continue to search until it has either found the desired number, or until every state pair has been explored.

See *Uniqueness* for further details on what FDR considers to be a *unique* counterexample. See *Debug Viewer* for a description on how to view the different counterexamples.

option refinement.explorer.storage.type

Default A default value chosen by FDR.

The data structure used for storing states to visit on the next ply of the breadth first search. The permitted values are:

Default Indicates that FDR should choose a storage structure. Presently, the default value is always a BTree, but at some point in the future it may be auto-selected depending on the problem.

BTree Use a *BTree* to store the states. This uses memory proportional to the number of node pairs stored, but with an overhead of at least 8 bytes per node pair stored.

LSMTree Use a *Log-Space Merge Tree*. This tree should perform only slightly worse than the BTree in the worst case, but may be much faster on some examples, particularly those that incorporate machines that have relatively random transition systems. This has essentially no storage overhead, but can end up storing multiple copies of the same node pair. However, the number of duplicate copies is bounded by log of the number of stored node pairs.

option refinement.storage.compressor

Default A default value chosen by FDR.

The type of compressor to use for the block-based storage used during *refinement checking*.

Default Lets FDR chose a compressor. Currently, this always defaults to LZ4.

DefaultHigh Lets FDR chose a compressor that will result in a high compaction ratio. Currently, this defaults to Zip. This can be useful on checks that exceed the amount of RAM available.

LZ4 Compresses the data using the LZ4 compression algorithm. This generally reduces the storage requirements to 0.3 of the original, whilst having no noticeable impact on checking time.

LZ4HC Compresses the data using the LZ4HC compression algorithm. This generally reduces the storage requirements to around 0.25 the original requirement, but halves the number of states that can be checked per second compared to LZ4.

Zip Compresses the data using the Zip compression algorithm. This generally reduces the storage requirements to around 0.2 of the original requirement, but will reduce the number of states that can be checked per second to around three quarters of the number that can be checked per second using LZ4.

option refinement.storage.file.path

Default None

If set to a comma-separated list of writeable directory paths FDR will use on-disk storage to store data during refinement checks, rather than relying on the system's swap implementation. In particular, FDR will still make use of RAM to cache data, but when the cache is filled, FDR will evict data to files stored in the directory as set above. The size of the in-memory cache can be set using *refinement.storage.file.cache_size*. For example, setting this to `/scratch/disk1,/scratch/disk2` will cause FDR to write data to both folders.

This option is only useful if the amount of memory required to complete a check exceeds the available RAM. In such cases, setting this option to a valid directory path typically increases performance. Further, it allows FDR to allocate more memory than is available to the system (which can only use RAM and swap).

For maximum performance this directory should point to a location on a solid-state drive (SSD). Use of traditional disk drives is not recommended since FDR is not optimised for such cases.

When this option is selected it may be necessary to increase the maximum number of files that are allowed to be simultaneously open. On large checks that consume several hundred gigabytes of storage, FDR will easily require several thousand files to be simultaneously open. The required number of files can be estimated by dividing the number of megabytes of storage required for the check by 64 (which is the file size FDR uses by default). To adjust the maximum number consult the documentation for your Linux distribution (searching for "set maximum number of open files" generates useful results). FDR checks at runtime if this value is sufficient for operation.

See also:

refinement.storage.file.cache_size Allows the amount of memory that FDR uses for the in-memory cache to be set.

refinement.storage.file.checksum If set, FDR will verify data that is written to the disk.

refinement.storage.file.compressor Allows extra compression to be applied to blocks written to disk, thus minimising the amount of on-disk storage required.

option refinement.storage.file.cache_size

Default 90%

If file based storage is being used (see *refinement.storage.file.path*), this specified the amount of memory to use before evicting data to disk. This can be specified either as a percentage of the available system RAM at the point the refinement check starts, or as an absolute number of bytes. For example, specifying `50%` will cause FDR to use 50% of the remaining system RAM at the point the check starts, whilst `100GB` would cause FDR to use 100GB of RAM for the cache. `KB`, `MB`, `GB` and `TB` may be used as suffixes to specify the amount of RAM to use for the cache.

For maximum performance this value should be set as high as possible, but must not be set so high that FDR would start to use swap for its in-memory cache.

option refinement.storage.file.checksum

Default Off

If file based storage is being used (see *refinement.storage.file.path*) and this option is set to `On`, this will cause FDR to verify data that is read from disk. This can be useful as a guard against disk corruption. If FDR detects a corrupted block it will abort the check (there is no way to simply revisit the affected states, unfortunately). This causes a small increase in runtime, generally around 5%.

option refinement.storage.file.compressor

Default None

If file based storage is being used (see *refinement.storage.file.path*) and this is set to a value other than `None`, this specifies an additional type of compression to apply to blocks that evicted to disk. This can be useful to minimise the amount of disk storage that is being used, but does result in the time to check a property increasing by anywhere between 5 and 50% (depending on the problem).

This may be set to any of the values that *refinement.storage.compressor* can be set to, in addition to the value `None`.

option refinement.track_history

Default On

This option controls whether FDR will record information that enables it to reconstruct traces if a counterexample is found. If this option is disabled, FDR will require less memory (up to 50% less), but will not be able to report any counterexamples. This option should only be used if the check passes, since FDR will not provide any information about why the check fails if the assertion does not pass.

CHAPTER

THREE

THE FDR COMMAND-LINE INTERFACE

In addition to the graphical interface, FDR also exposes a command-line interface. Whilst this is not particularly useful as a standalone tool, primarily due to the difficulty in navigating counterexamples, it can be useful for quickly checking if assertions pass. More importantly, the command-line tool can also produce *machine-readable output* (in either JSON, XML or YAML), providing an easy way of integrating FDR into other tools. The command-line version can also be executed on *clusters*, enabling massive problems to be tackled.

On Linux the command-line tool can be invoked simply as `refines` (providing the standard installation instructions have been followed). Under Mac OS X, the command line tool can be invoked by launching `/Applications/FDR4.app/Contents/MacOS/refines`, assuming that FDR has been installed to `/Applications/`.

3.1 Command-Line Flags

`refines` takes as input a list of files and will check all assertions in all files. For example:

```
$ refines a.csp b.csp
```

will cause `refines` to load `a.csp`, check all assertions in it, then load `b.csp` and then check all assertions. If the filename is set to `-`, then `refines` will then read from stdin. For instance:

```
$ refines - < a.csp
```

will cause `refines` to check assertions in `a.csp`, since it is piped into stdin.

`refines` takes various option flags, as follows

--archive `<output_file>`

If this option is specified then FDR will read in the specified CSP file, calculate all files that it *includes*, and then save all of these into a single compressed file named output_file. `output_file` may subsequently be loaded by refines in the normal way. For example:

```
$ refines --archive phils.arch phils6.csp
Saved phils8.csp (and all depdendent files) to phils.arch
$ refines phils.arch
SYSTEM :[deadlock free [F]]:
  Log:
      Found 50 processes including 7 names
      ...
```

This is intended to help running refines on a remote server since only the single combined archive needs to be transferred, rather than all files that the root file includes. For example:

```
$ refines --archive phils.arch phils6.csp
$ scp phils.arch server:phils.arch
$ ssh server "refines phils.arch"
```

would execute refines on the computer named `server`.

--brief, -b
If this option is included then only the result of each assertion is printed, rather than a description of the counterexample.

See also:

refines --divide, *refines --reveal-taus*

--divide, -d
If selected, any counterexample that is generated will be split and the behaviours of component machines will also be output (as per the *Debug Viewer*).

See also:

refines --brief, *refines --reveal-taus*

--format `<format>`**, -f** `<format>`
Specifies the output format, which must be one of:

colour This is the default mode. This pretty-prints texts in a human-readable format and uses some colours to highlight text printed to the terminal.

plain As per `colour`, but no colours are used.

json Outputs machine-readable JSON, as described below in *Machine-Readable Formats*.

framed_json This is as per the `json` format, but instead outputs one complete JSON object after each assertion or print statement. See *Machine-Readable Formats* for further details.

xml Outputs machine-readable XML, as described below in *Machine-Readable Formats*.

yaml Outputs machine-readable YAML, as described below in *Machine-Readable Formats*.

--help, -h
Prints the list of command-line flags that are available.

--quiet, -q
This suppresses all the progress logging that FDR normally generates.

--remote `<ssh_host>`
This causes `refines` to check the assertions on the specified remote host, using SSH. `refines` will connect to the specified server, upload the script (including any scripts it includes), and then invokes `refines` on the remote server. For example:

```
$ refines --remote myserver phils6.csp
```

would cause `refines` to use SSH to connect to `myserver` and then invoke `refines on the remote server in order to check all assertions in ``phils6.csp`.

The `--remote` option can be used to specify an arbitrary sequence of options to `ssh`. For instance:

```
$ refines --remote '-o "PubkeyAuthentication=yes" -p 1000 user@remote' phils6.csp
```

would mean `refines` invoked `ssh` as `ssh -o "PubkeyAuthentication=yes" -p 1000 user@remote>` (along with some other options to start `refines` on the remote server).

In order to use this comand, `ssh` (or `ssh.exe` on Windows) must be available on your `$PATH`, and `refines` must be available on your `$PATH` *on the remote server*. The version of `refines` does not need to exactly match the version on the remote server, but should be similar.

See also:

refines --remote-path in order to change the path to `refines` on the remote server.

--remote-path <path>

This can be used to specify the path to `refines` on the remote server when using *refines --remote*. For example:

```
$ refines --remote myserver --remote-path /var/bin/refines phils6.csp
```

would cause `/var/bin/refines` to be invoked on the remote server, rather than `refines`.

See also:

refines --remote for further details on how to invoke `refines` with the `--remote` option.

--reveal-taus

If selected, will print the top-level trace of the system with all taus revealed. That is, any tau in the counterexample that is a tau that resulted from hiding will be *revealed* and the event that was hidden given instead.

For example, consider the following CSP script:

```
channel a, b

P = a -> b -> STOP

assert STOP [T= P \ {a}
```

Running refines on this file using the above option would print:

```
Trace: <τ>
Error Event: b
Unhidden trace: <a>
```

which indicates that the first tau was an *a*.

See also:

refines --brief, *refines --divide*

--typecheck, -t

Typecheck the file arguments and exit.

--version, -v

Prints the version of the current version of FDR.

Further, any of the options that are available in the GUI, listed in *Options* can also be specified from the command line by replacing each `.` or `_` in the option name with a `-`. For example, *refinement.storage.compressor* can be set to `Zip` by adding the argument `--refinement-storage-compressor Zip`.

3.2 Examples

The following causes FDR to check all assertions in the file, outputting as much information as possible.

```
$ refines phils6.csp
SYSTEM :[deadlock free [F]]:
    Log:
        Found 50 processes including 7 names
        Visited 43 processes and discovered 6 recursive names
        Constructed 0 of 1 machines
        Constructed 0 of 2 machines
        Constructed 0 of 3 machines
...
```

The next example will cause FDR to check all assertions in the file, but suppresses all logging (due to *refines --quiet*) and causes only a summary of the results to be produced (due to *refines --brief*).

```
$ refines --brief --quiet phils6.csp
SYSTEM :[deadlock free [F]]: Failed
SYSTEMs :[deadlock free [F]]: Failed
BSYSTEM :[deadlock free [F]]: Passed
ASSYSTEM :[deadlock free [F]]: Passed
ASSYSTEMs :[deadlock free [F]]: Passed
```

This example supresses all logging, but will cause FDR to pretty-print counterexamples to any assertions that fail. FDR will also divide the counterexamples to give behaviours for each subprocess of the main process (thus allowing each component's contribution to the error to be deduced).

```
$ refines --divide --quiet phils6.csp
SYSTEM :[deadlock free [F]]:
    Log:
    Result: Failed
        Visited States: 181
        Visited Transitions: 493
        Visited Plys: 9
        Counterexample (Deadlock Counterexample)
        Implementation Debug:
            SYSTEM (Failure Behaviour):
                Trace: <thinks.1, sits.1, thinks.0, thinks.2, sits.2, sits.0,
                picks.1.1, picks.0.0, picks.2.2>
                Min Acceptance: {}
                Component Behaviours:
...
```

3.3 Using a Cluster

`refines` can also be executed on clusters using *MPI <http://www.mpi-forum.org/>* in the standard way. For example:

```
$ mpiexec refines phils6.csp
```

will execute refines on whatever cluster `mpiexec` is configured to use. Note that all other options for refines, including those that control machine- readable output, function as normal.

For optimal performance, `refines` should be executed on a cluster with a high-performance interconnect and consisting of homogeneous compute nodes (i.e. with the same number and speed of cores). Further, exactly one copy of `refines` should be executed on each physical server. `refines` will still use all of the cores, but will use a more efficient communication algorithm for communication with other threads on the same physical node.

For example, to execute `refine` on two homogeneous nodes `node001` and `node002`, the following could be used:

```
$ mpiexec -hosts node001,node002 refines --refinement-cluster-homogeneous true phils6.
↪csp
```

Note that *refinement.cluster.homogeneous* has been set to `true`. This option should always be specified when using a homogeneous cluster, since it makes FDR assume the cluster is homogeneous. If it is not specified there is a small chance FDR may fail to detect the homogeneity of the cluster, leading to suboptimal performance.

The required interconnect speed depends on the problem, but in our experience, if each node in the cluster has 8 cores, the interconnect needs to be able to support 750 Mb/s (e.g. gigabit Ethernet), whilst if each node has 16 cores, the interconnect needs to support 1500 Mb/s (e.g. 10-gigabit Ethernet, or InfiniBand).

See also:

refines --archive If your CSP_M scripts make use of *Include Files*, then `refines --archive` can help when transferring files over to a cluster since it packages all files, including all files that are included, into a single compressed archive. For example:

```
$ refines --archive phils.arch phils6.csp
$ scp phils.arch cluster-master:phils.arch
$ ssh cluster-master "mpiexec -hosts node001,node002 refines --refinement-cluster-
↪homogeneous true phils.arch"
```

would execute FDR on the cluster consisting of `node001` and `node002`.

Warning: At the present time, only checks in the traces and failures models will take full advantage of the cluster mode as parallelising divergence checking has not be parallelised to take advantage of multiple machines in an optimal way.

3.3.1 Supported MPI Versions

Currently the cluster version of `refines` only supports Linux running one of the following MPI distributions:

- MPICH 1.4.1;
- MPICH 3.1.2;
- MVAPICH 1.8.1.

The usage of `MPICH 3.1` (or any other MPI3 compliant distribution) is strongly recommended, particularly on larger clusters.

Further, FDR4 requires that the MPI distribution has support for multi-threaded applications (in particular `MPI_THREAD_FUNNELED`). When running under `mvapich`, this requires `-env IPATH_NO_CPUAFFINITY 1 -env MV2_ENABLE_AFFINITY 0` to be passed to `mpiexec`.

Please contact us if you would like us to add support for an alternative MPI distribution, or if `refines` fails to auto-detect your MPI distribution.

3.3.2 Using Cloud Computing Services

Amazon's EC2 is ideally suited to being used for FDR. In particular, Amazon's support for Enhanced Networking (i.e. 10-gigabit Ethernet) and the availability of Placement Groups (which guarantee optimal network performance between compute nodes) means that they are ideal hosts for FDR. In particular, the large `r3.4xlarge` (8 cores) and `r3.8xlarge` (16 cores) provide an excellent balance of memory and compute power for `refines`.

Creating and optimally configuring a cluster on Amazon is complex. We recommend the use of StarCluster, which can automatically setup and configure a cluster on EC2. You may use the following starcluster configuration as a starting point for a FDR-compatible cluster. Note the presence of `SUBNET_ID` and `VPC_ID`: in order for Enhanced Networking to function correctly, the nodes in the cluster must be part of VPC on Amazon. The StarCluster user guide can provide more help on this point.

```
[cluster fdr]
CLUSTER_SIZE = ...
NODE_INSTANCE_TYPE = r3.8xlarge
DNS_PREFIX = True
NODE_IMAGE_ID = ami-52a0c53b
SUBNET_ID = ...
VPC_ID = ...
PLUGINS = mpich2, mpich3, pkginstaller, sge
DISABLE_QUEUE = True

[plugin mpich2]
SETUP_CLASS = starcluster.plugins.mpich2.MPICH2Setup

[plugin sge]
# Don't use the default SGE setup because we only want one slot per node
SETUP_CLASS = starcluster.plugins.sge.SGEPlugin
SLOTS_PER_HOST = 1

[plugin pkginstaller]
SETUP_CLASS = starcluster.plugins.pkginstaller.PackageInstaller
PACKAGES = language-pack-en

[plugin mpich3]
SETUP_CLASS = mpich3.MPICH3Setup
```

This also requires the following plugin, which builds mpich3 from source, to be installed to `~/.starcluster/plugins/mpich3.py`:

```
from starcluster.clustersetup import ClusterSetup
from starcluster.logger import log

class MPICH3Setup(ClusterSetup):
    def _configure_node(self, node):
        env_file = node.ssh.remote_file('/etc/profile.d/mpich3.sh', 'w')
        env_file.write("PATH=/home/sgeadmin/mpich3/bin:$PATH\n")
        env_file.close()

    def run(self, nodes, master, user, user_shell, volumes):
        log.info("Building mpich 3.1.1 from source")
        master.ssh.execute("""
            cd /home/sgeadmin
            wget http://www.mpich.org/static/downloads/3.1.1/mpich-3.1.1.tar.gz
            tar xzvf mpich-3.1.1.tar.gz
            cd mpich-3.1.1
            ./configure --prefix=/home/sgeadmin/mpich3
            make -j16
            make install
            """)
        log.info("Setting environment on all nodes...")
        for node in nodes:
            self._configure_node(node)
```

(continues on next page)

(continued from previous page)

```
    def on_add_node(self, node, nodes, master, user, user_shell, volumes):
        self._configure_node(node)
```

```
from starcluster.clustersetup import ClusterSetup
from starcluster.logger import log
from starcluster import threadpool

class FDR4Setup(ClusterSetup):
    def _configure_node(self, node):
        node.ssh.execute("echo \"deb http://www.cs.ox.ac.uk/projects/fdr/downloads/
↪debian/ fdr release\n\" > /etc/apt/sources.list.d/fdr.list")
        node.ssh.execute("wget -qO - http://www.cs.ox.ac.uk/projects/fdr/downloads/
↪linux_deploy.key | apt-key add -")
        node.ssh.execute("apt-get update")
        node.ssh.execute("apt-get install fdr")

    def run(self, nodes, master, user, user_shell, volumes):
        log.info("Installing FDR4 on each node")
        pool = threadpool.get_thread_pool(20, False)
        for node in nodes:
            pool.simple_job(self._configure_node, (node), jobid=node.alias)
        pool.wait(numtasks=len(nodes))

    def on_add_node(self, node, nodes, master, user, user_shell, volumes):
        self._configure_node(node)
```

Other cloud computing services are only suitable if they can guarantee extremely high performance connections between the compute nodes in the cluster (multi gigabit sustained throughput for each node, and very low latency).

3.4 Machine-Readable Formats

> **Warning:** Before deciding to use the machine-readable interface to FDR, please firstly consider using the *FDR API* instead, which allows for more thorough integration.

When a machine-readable format is selected (i.e. when *`refines --format`* is set to `framed_json`, `json`, `xml` or `yaml`, e.g. `refines --format json file.csp`), the behaviour of `refines` is slightly modified, as follows. Firstly, errors and warnings that are generated as a result of the input script are no-longer sent to `stderr`, but instead are included as part of the machine-readable output. Errors that result from incorrect command-line flags being passed are still sent to stderr. If *`refines --quiet`* is not specified, then log messages are now sent to stderr. The machine-readable output is sent to stdout. The exit code for `refines` will be 0 precisely when valid machine-readable output has been generated, and a value other than 0 indicates some sort of catastrophic error, either as a result of incorrect flags being passed or a runtime crash (check stderr for more information). In particular, an error in the input CSP script, or a failing assertion, will result in an exit code of 0 and the errors will be included as part of the machine-readable output.

The machine-readable output consists essentially of a number of key-value pairs, specified in either JSON, XML or YAML. The format of the various element types is specified below. The top-level element of the output (the element type is `file` in XML) and is documented in *Files*.

If *`refines --format`* is set to `framed_json`, then the behaviour of `refines` is further altered. Normally, if `refines` were to be killed, or crashed, when the JSON format is specified, then only a partially-formed JSON object will be output. This means that any assertion results that were already finished are essentially lost. The `framed_json` format is designed to avoid this problem. Instead of outputing a JSON document only when all assertions and print statements are checked, it outputs a fully formed JSON object after each assertion or print statement. Thus, the results for the ith assertion are contained in the ith file object.

The delimiter between the different files is a single LF character (i.e. `\\n`). This means the output can be parsed as follows:

```
for document in fdr_output.split('\n'):
   file_object = json.loads(document)
```

Each JSON document is a fully formed File object, as documented in *Files*. However, each File object will contain the results of precisely one assertion or print statement.

3.4.1 Files

The top-level element in the output conceptually represents an input file, and has the following properties.

Element	Meaning
`errors`	A list of errors that were generated in response to loading the file.
`event_map`	For various reasons, FDR internally represents events as integers. Thus, in the counterexamples below all events are actually integers, rather than strings. This element contains a map from each integer event to the corresponding string representation. **Warning:** In the case of JSON the keys to this map are actually strings, rather than integers since JSON requires all keys to be strings. In the case of XML, the map is represented by a series of elements of the form <i_40>done</i_40>, which indicates that the event `40` maps to the string `done`. This is to compensate for the fact that XML does not allow elements to start with numbers. **Warning:** No guarantees are provided about the order of the keys in the map. The only guarantee that is provided is that each key will appear in the map at most once.
`file_name`	The name of the file that was loaded.
`results`	A list of assertion results, the format of which is described below. This will be empty if errors was non-empty. The results will appear in the same order as the assertions in the original file.
`print_statements`	A list of results from evaulating print statements, the format of which is described below. As with `results`, this will be empty if errors was non-empty, and the results will appear in the same order as the print statements in the original file.
`warnings`	A list of warnings that were generated when loading the file.

For example, executing `refines --quiet --format=json phils6.csp` will result in the following JSON to be produced:

```
{
    "errors": [],
    "event_map": {
        "13": "thinks.1",
        "14": "sits.1",
        ...
```

(continues on next page)

(continued from previous page)

```
    },
    "file_name": "phils6.csp",
    "results": [
        ...
    ],
    "print_statement_results": [],
    "warnings": []
}
```

Examples of the result section are given below.

3.4.2 Assertion Results

Each item in the `results` list of *Files* will contain the following fields.

Element	Meaning
`assertion_string`	A string representation of the assertion.
`counterexamples`	A list of counterexamples to the assertion. If the assertion failed (i.e. `result` is `0`) or the assertion passed but `is_negated` is `1`, this will be non-empty. Otherwise it will be empty. The keys that this contains are specified below.
`errors`	A list of errors that were encountered whilst compiling the assertion. If this list is non-empty then none of the following elements will be present.
`is_negated`	This will be `1` if the assertion is of the form `assert not` and `0` otherwise.
`result`	This will be `1` if the assertion passes and `0` otherwise. Note that if the assertion is negated and the inner assertion fails, then this will be `1`.
`visited_plys`	An integer giving the number of plys that were visited in the breadth-first search.
`visited_states`	An integer giving the number of states visited during the search.
`visited_transitions`	An integer giving the number of transitions that were visited.

For example, executing `refines --quiet --format=json phils6.csp` and extracting the first element of the results array will give the following JSON:

```
{
    "assertion_string": "SYSTEM :[deadlock free [F]]",
    "counterexamples": [
        ...
    ],
    "errors": [],
    "is_negated": 0,
    "result": 0,
    "visited_plys": 9,
    "visited_states": 181,
    "visited_transitions": 493
},
```

The contents of an element of the `counterexamples` list are given below.

3.4.3 Counterexample

A counterexample to an assertion conceptually consists of a behaviour of the implementation that violates the assertion in some way. A `counterexample` element contains the following fields.

Element	Meaning
implementation_behaviour	A *Behaviour* of the implementation that the specification cannot perform. The format of this is specified below.
specification_behaviour	A *Behaviour* of the specification. The format of this is specified below. This item is only present when the assertion is a refinement assertion, or a determinism assertion; for all other assertions (such as deadlock and divergence free assertions), this will not be present.
type	A string representing the type of the counterexample. This will either be: **deadlock** This is generated as a counterexample to a `:[deadlock free]` assertion and indicates that the implementation can deadlock after a certain trace (or, if the failures-divergences model is being used, that the implementation can diverge). In this case, `specification_behaviour` is not present since its behaviour is not relevant. **determinism** This is generated in response to a `:[determinism]` assertion, and indicates that the implementation can diverge after a certain trace. In this case, `specification_behaviour` and `implementation_behaviour` are not really behaviours of the specification and the implementation, but instead are two behaviours that demonstrate non-determinism in the implementation process. Either: one behaviour will be a trace behaviour (indicating it can perform a certain visible event) and once behaviour will be an acceptance behaviour, indicating that the process can refuse the event; or one behaviour will be a divergence behaviour and the other will be an irrelevant trace behaviour. **divergence** This is generated in response to a `:[divergence free]` assertion, and indicates that the implementation can diverge after a certain trace. In this case, `specification_behaviour` is not present since its behaviour is not relevant. **refinement_divergence** This is generated in response to a violation of a failures-divergences refinement assertion, and indicates that the implementation can diverge after a certain trace, but the specification cannot. **failure** This indicates that this counterexample is from a failing failures check. Hence, the implementation can refuse to perform events that the specification does not allow to be refused. **trace** This indicates that the implementation can perform a trace that the specification cannot.

For example, executing `refines --quiet --format=json phils6.csp` and extracting the first element of the counterexamples array from the JSON example given in *Assertion Results* will give the following JSON:

```
{
    "implementation_behaviour": ...,
    "type": "deadlock"
}
```

The contents of the `implementation_behaviour` value are given below.

3.4.4 Behaviour

Conceptually, a behaviour of a machine (i.e. a process) is a trace that it can perform, along with some action that it can perform after performing the trace that is of interest. For example, a behaviour may indicate that the machine can diverge after a certain trace, or that it accept only a certain set of events, etc. A `behaviour` element has the following fields:

<table>
<tr><th>Element</th><th>Meaning</th></tr>
<tr><td>child_behaviours</td><td>If refines --divide is specified and this machine is divisible, this will consist of an array of behaviours of the subprocesses of this machine. For example, if this behaviour is a behaviour of `P ||| Q`, then this would have two elements, one representing `P` and the other representing `Q`.</td></tr>
<tr><td>machine_name</td><td>If this behaviour is a behaviour of a machine whose name is known (i.e. this is a behaviour of a process that is defined as `P = X ||| Y` in the input file), this this field contains a string representation of the process name.</td></tr>
<tr><td>trace</td><td>This is an array of integers that represent the events that lead up to the actual behaviour. Note that unlike traces from CSP theory, this may include taus.
Note that FDR aligns all traces (as seen in the Debug Viewer), meaning that if you have two behaviours then their trace lengths are guaranteed to be the same. Further, if two behaviours both perform an event at some index `i` and neither is the child of the other, then it must be the case that the two events are synchronised somehow. In order to ensure that all traces are identical, FDR inserts the event `0` in traces to indicate that this machine does not contribute to this event and does not change state. For example, consider:
<pre>
channel a, b

P = a -> a -> STOP
Q = a -> b -> a STOP

assert a -> b -> STOP [T= P [| {a} |] Q
</pre>
A counterexample to this will have `implementation_behaviour` being a trace behaviour with trace `<a, b>` and error event `a`. Its two children will have traces `<a, 0>` and `<a, b>` respectively. This indicates that both components synchronise on the `a`, but the second component performs the `b` independently of the first.</td></tr>
<tr><td>revealed_trace</td><td>This field is only present if refines --reveal-taus is specified and this is the top-level implementation behaviour. If so, then this will be as per `trace`, but any taus that resulted from hiding will be revealed and the original visible event printed instead.</td></tr>
<tr><td>type</td><td>This indicates the type of the behaviour, and will be one of the following strings:
divergence This indicates that the process can diverge having performed the trace. There are no extra properties of the behaviour in this case.
loop This indicates that the process can repeat some suffix of the trace infinitely often. In this case the field `loop_start` contains an integer that specifies the index at which the repeat starts. For example, if `loop_start` was `0` then this indicates the machine can repeat the entire trace, whilst a value of `1` means that it can repeat all but the first event. This often appears when dividing divergence counterexamples.
min_acceptance This indicates that the process will only accept a certain set of events having performed the trace. The set of events that it accepts is given by the `acceptance` field, which is a list of integers corresponding to the events that it accepts. This will appear either in a deadlock counterexample where the `acceptance` field will be the empty array for the top-most implementation behaviour (indicating that it accepts no events, i.e. it deadlocks). It also appears during `failure` counterexamples.
segmented This behaviour type appears when counterexamples involving processes such as `P = normal(Q) ; P` are discovered. In this case FDR will have, most likely, compiled these to [recursive high-level machines](@ref compiler_machine_types). It is possible for a behaviour of `P` to require several loops around `Q`. Such behaviours are represented by segmented behaviours. Conceptually, these consist of a series of sections, which are themselves behaviours, that the machine performs in sequence. Thus, segmented behaviours have two extra fields:
last_section A `behaviour` that is the last behaviour the machine performs. This is the behaviour that will exhibit the counterexample to the assertion (i.e. this behaviour may be a divergence, loop etc.).
prior_sections A list of `behaviour` elements, all of which are guaranteed to have type `trace`. The machine performs these in order. The `error_event` in each of the behaviours is what causes the machine to change to the next segment.
For example, consider:
<pre>
channel a
transparent normal
Q = a -> SKIP
P = normal(Q) ; P
</pre></td></tr>
</table>

For example, executing `refines --quiet --format=json phils6.csp` and extracting the `implementation_behaviour` of the `counterexample` element in the JSON example given in *Counterexample* will give the following JSON:

```
{
    "acceptance": [],
    "child_behaviours": [
       ...
    ]
    "machine_name": "SYSTEM",
    "trace": [
        13,
        14,
        ...
    ],
    "type": "min_acceptance"
}
```

3.4.5 Print Statements

Each item in the `print_statement_results` list of *Files* will contain the following fields.

Element	Meaning
`print_statement`	The print statement being evaluated.
`location`	The location in the source code of the print statement.
`errors`	A list of errors that were encountered whilst evaluating the print statement. If this list is non-empty then none of the following elements will be present.
`result`	This element is only present if errors is empty, and gives the result of evaluating the print statement.

For example, suppose `test.csp` contains the line `print 1+1`. Then, executing `refines --quiet --format=json test.csp`, and inspecting the first item in the `print_statement` array will give:

```
{
  "print_statement": "1+1",
  "location": "test.csp:1:1-10",
  "errors": [],
  "result": "2"
}
```

3.4.6 Code Examples

The following Python program invokes `refines` and will check all of the assertions in the given file, printing out limited debug information. This can be used as a skeleton for calling `refines`.

```
# Used for parsing the output of refines
import json
# Used to invoke refines
import subprocess
# For accessing command line arguments
import sys

path_to_refines = "refines"
```

(continues on next page)

(continued from previous page)

```
def run_fdr(file_to_check):
    print "Running FDR on", file_to_check
    # Documented at
    # http://docs.python.org/2/library/subprocess.html#subprocess.Popen
    fdr_process = subprocess.Popen([path_to_refines, "--format=json", file_to_check],
        stdout = subprocess.PIPE, stderr = subprocess.PIPE)

    # We launch FDR inside a try block to catch a user pressing CTRL+C and
    # then terminate FDR. If we did not do this FDR would continue running
    # in the background
    try:
        # Documented at
        # http://docs.python.org/2/library/subprocess.html#subprocess.Popen.
↪communicate
        (stdout, stderr) = fdr_process.communicate()
    except KeyboardInterrupt:
        fdr_process.terminate()
        # Re-throw the exception
        raise

    print "Finished"

    if fdr_process.returncode == 0:
        print "Log data was:"
        print stderr

        # Parse the machine-readable data
        parsed_data = json.loads(stdout)

        for error in parsed_data["errors"]:
            print "Error:", error
        for warning in parsed_data["warnings"]:
            print "Warning:", error

        # If we generated results (if there were errors above this would not
        # be present)
        if "results" in parsed_data:
            for assertion in parsed_data["results"]:
                print "Assertion:", assertion["assertion_string"]
                # If the assertion has errors then we emit those.
                if "errors" in assertion:
                    print "    Errors during assertion"
                    for error in assertion["errors"]:
                        print "Error:", error
                else:
                    print "    Visited States:", assertion["visited_states"]
                    print "    Passed:", assertion["result"] == 1
    else:
        # Since FDR exited with a non-zero exit code we cannot parse the
        # data on stdout, since it may well be malformed.
        print "Failed - exit code", fdr_process.returncode

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print "Please specify exactly one file"
    else:
        run_fdr(sys.argv[1])
```

Note: We would welcome contributions of similar examples in other languages.

CHAPTER

FOUR

CSP_M

CSP_M is a lazy functional language with built-in support for defining CSP processes. It also allows *assertions* to be made about the resulting CSP processes.

4.1 Definitions

CSP_M files consist of a number of *definitions*, which are described below. Some of these definitions can also be given at the FDR *The Interactive Prompt* and within `Let` expressions. In this Section we give an overview of the type of declarations allowed in CSP_M. A *declaration list* is simply a list of the declarations in this section, separated by newlines.

4.1.1 Constants

The simplest type of definition in CSP_M binds the value of an *expression* to a *pattern*. In particular, if `p` is a pattern and `e` is an expression, both of some type `a`, then `p = e` matches the value of `e` against `p` and, if `p` does match then binds all the variables of `p`, otherwise throws an error. For example, `x = 5` binds `x` to `5`, `(x, y) = (5, f(5))` binds `x` to `5` and `y` to `f(5)`. Alternatively, given `<x> = <>` the following error is thrown upon evaluating x:

```
Pattern match failure: Value <> does not match the pattern <x>
```

4.1.2 Functions

CSP_M provides a rich syntax for defining functions that is highly expressive. The simplest example of a function is the identify function, which simply returns its argument unaltered. This can be written in CSP_M as `id(x) = x` and has type `(a) -> a`. CSP_M also allows function definitions to use *pattern matching* to define functions. For example, given the definition of `f` as:

```
f(0) = 1
f(1) = 2
f(_) = error("Disallowed argument")
```

Then `f(0)` evaluates to `1`, `f(1)` evaluates to `2`, whilst any other argument evaluates to an error. Functions can also take multiple arguments, separated by commas. Thus, `f(x,y) = x*y` defines the multiplication function.

CSP_M also provides support for curried function definitions. For example, `f(x)(y) = x+y` means that `f` is of type `(Int) -> (Int) -> Int` (noting that `->` is right associative). Evaluating `f(4)` yields a function to which the second argument may be passed. Thus, `f(4)(2) = 6`.

4.1.3 Type Annotations

Functions and *constants* may have their type specified by providing a type annotation on a separate line to the main definition. For example, the following specifies that the function `f` has the type of the identity function:

```
f :: (a) -> a
```

It is also possible to specify that a type variable has certain type constraints. For example, the following requires that `g` is a function that takes two arguments of the same type that must satisfy the *Eq type constraint*:

```
g :: Eq a => (a, a) -> Bool
```

The type annotations, in addition to being a useful way of documenting programs, are used by the type-checker to guide it to deducing the correct type, particularly for functions that use *Dot* in various complex ways. Further, the use of type annotations will result in more useful errors being emitted.

In general, a type annotation is a line of the form:

```
n1, n2, ..., nM :: Type

n1, n2, ..., nM :: TypeConstraint a => Type

n1, n2, ..., nM :: (TypeConstraint a, ..., TypeConstraint a) => Type
```

where the `ni` are names whose definitions are given at the same lexical level (i.e. if the type of `f` is declared inside a let expression, then `f` must also be declared inside exactly the same let expression); `TypeConstraint` is a *type constraint*; `Type` is a *Type System* Type expression.

Type variables within type annotations are scoped in the same way as variables. For example, consider the following definition, which contains a let expression:

```
f :: Set a => (a) -> {a}
f(x) =
   let
      g :: (a) -> {a}
      g(x) = {x}
   within g(x)
```

In the above, the type variable `a` in the type annotation for `g` is precisely the same type variable as in the type annotation for `f`. Hence, the type annotation for `g` does not require the `Set` type constraint since this has already been specified in the type annotation for `f` (in general, type constraints may only be specified in the type annotation where a type variable is created).

Warning: Specifying type annotations for non-existent names will result in an error, as will specifying multiple type annotations for the same name. Further, type constraints may only be specified in the type annotation that creates the type variable.

See also:

Type System This gives the full syntax for types in $\mathrm{CSP_M}$.

Type Constraints This includes a complete list of supported type constraints.

New in version 3.0.

4.1.4 Datatypes

CSP_M also allows structured datatypes to be declared, which are similar to Haskell's `data` declarations. The simplest kind of datatype declaration simply declares constants:

```
datatype NamedColour = Red | Green | Blue
```

This declares `Red`, `Green` and `Blue` as symbols of type `NamedColour`, and binds `NamedColour` to the set `{Red, Green, Blue}`. Datatypes can also have parameters. For example, we could add a RGB colour data constructor as follows:

```
datatype ComplexColour = Named.NamedColour | RGB.{0..255}.{0..255}.{0..255}
```

This declares `Named` to be a *data-constructor*, of type `NamedColour => ComplexColour`, `RGB` to be a data-constructor of type `Int => Int => Int => RGB` and `ComplexColour` to be the set:

```
union({Named.c | c <- NamedColour},
      {RGB.r.g.b | r <- {0..255}, g <- {0..255}, b <- {0..255}})
```

Thus, in general, if a datatype `T` is declared, then `T` is bound to the set of all possible datatype values that may be constructed. Note that, as mentioned in *Dot*, if an invalid data-value is constructed then an error is thrown. Thus, constructing `RGB.1000.0.0` would throw an error, as `1000` is not in the permitted range of values. This is the primary difference between datatypes in other languages and CSP_M: CSP_M requires the actual set of values allowed at runtime to be declared, whilst other languages merely require the type. This is done to allow *Prefix* to be used more conveniently.

One important consideration is that the datatype must be *rectangular*, in that in must be decomposable into a Cartesian product. For example, consider the following:

```
datatype X = A.{x.y | x <- {0..2}, y <- {0..2}, x+y < 2}
```

This datatype is not rectangular as the datatype declaration cannot be rewritten as the Cartesian product of a number of sets, since `A.0.1` is valid, whilst `A.1.1` is not. This will result in the following error being thrown:

```
./test.csp:1:14-57:
 The set: {0.0, 0.1, 1.0}
 cannot be decomposed into a cartesian product (i.e. it is not rectangular).
 The cartesian product is equal to: {0.0, 0.1, 1.0, 1.1}
 and thus the following values are missing: {1.1}
```

Datatypes can also be recursive. For example, the type of binary trees storing integers can be declared as follows:

```
datatype Tree = Leaf.Int | Node.Int.Tree.Tree
```

For example, the following function *flattens* a binary tree to a list of its contents:

```
flatten(Leaf.x) = <x>
flatten(Node.x.l.r) = flatten(l)^<x>^flatten(r)
```

This function is of type `(Tree) -> <Int>`.

In general a CSP_M datatype declaration takes the following form:

```
datatype N = C1.te1 | C2.te2 | ...
```

where N is the datatype name, the `tei` are optional and are *Type Expressions* (which differ from regular expressions) and the `Ci` are the clause names. As a result of this, each `Ci` is bound to a datatype constructor that when dotted with appropriate values (according to `tei`), yield a datatype of type `N`. In particular, if `tei` is of type `{t1.(...).tN}`, then `Ci` is of type `t1 => ... => tN => N`. Further, `N` is bound to the set of all valid datatypes values of type `N`.

Note: CSP_M datatypes cannot be parametric, so it is not possible, for instance, to declare a binary tree that stores any type at its node.

See also:

Variable Pattern for a warning on channel names to avoid.

Type Expressions

The expressions in datatype and channel declarations are interpreted differently to regular expressions. *Type expressions* can either be any *Expressions* that evaluates to a set or a dot separated list of type expressions, or a tuple of type expressions. Thus a type expression `te` is of the form:

```
e | (te1, ..., teN) | te1.(...).teN
```

where `e` denotes an :ref‘expression <csp_expression>‘ of type `{a}`. A type expression of the form `(te1, ..., teN)` constructs the set of all tuples where the i^{th} element is drawn from `tei`. For example, `({0..1}, {2..3})` evaluates to `{(0, 2), (0, 3), (1, 2), (1, 3)}`. A type expression of the form `te1.(...).teN` evaluates like a tuple type expression, but instead dots together the value. Thus, `{0..1}.{2..3}` evaluates to `{0.2, 0.3, 1.2, 1.3}`. For example, consider the following datatype declarations:

```
datatype X = A.(Bool, Bool)
datatype Y = B.Bool.Bool
```

This means that `X` is equal to the set `{A.(false, false), A.(false, true), A.(true, false), A.(true, true)}`, whilst `Y` is equal to the set `{B.false.false, B.false.true, B.true.false, B.true.true}`.

4.1.5 Channels

CSP_M channels are used to create events and are declared in a very similar way to datatypes. For example:

```
channel done
channel x, y : {0..1}.Bool
```

declares three channels, one that takes no parameters (and hence `done` is of type *Event*), and two that have two components: a value from `{0.1}` and a boolean. Thus, the set of events defined by the above is `{done, x.0.false, x.1.false, x.0.true, x.1.true, y.0.false, y.1.false, y.0.true, y.1.true}`. These events can then by used by *Prefix* to declare processes such as `P = x?a?b -> STOP`.

In general, channels are declared using the following syntax:

```
channel n1, ..., nM : te
```

where `te` is a *Type Expressions* and the `ni` are unqualified *names*. As a result of this, `n1`, `nM` are bound to event constructors that when dotted with appropriate values (according to `te`), yield an event‘‘. In particular, if `te` is of type `{t1.(...).tN}`, then each `ni` is of type `t1 => ... => tN => Event`.

See also:

Variable Pattern for a warning on channel names to avoid.

4.1.6 Subtypes

CSP$_M$ allows *subtypes* to be declared, which bind a set to a subset of datatype values. For example, consider the following:

```
datatype Type = Y.Bool | Z.Bool
subtype SubType = Y.Bool
```

This creates a *datatype*, as above, but additionally binds `SubType` to the set `{Y.false, Y.true}`. In general a subtype takes the following form:

```
subtype N = C1.te1 | C2.te2 | ...
```

where `N` is the name of the subtype, the `Ci` are the names of existing data constructors and the `tei` are *Type Expressions*. Note that the `Ci.tei` must all be of some common type `T`, which must be the type of a datatype (e.g., in the above example, `Y.Bool` is of type `Type`).

4.1.7 Named Types

Named types simply associate a name with a set of values, defined using type expressions. For example:

```
nametype X = {0..1}.{0..1}
```

binds `X` to the set `{0.0, 0.1, 1.0, 1.1}`. This is no more powerful than a *`Set Comprehension`*, but as it uses *Type Expressions*, it can be significantly more convenient. The general form of a nametype is:

```
nametype X = te
```

where `te` is a *Type Expressions*.

4.1.8 Assertions

CSP$_M$ also allows various *assertions* to be defined in CSP$_M$ files. These are then added to the *list of assertions* in FDR in order to allow convenient execution of the assertions. FDR permits several different forms of assertions, as described below. Further, options such as partial order reduction may be specified to. See *Assertion Options* for further details.

Refinement Assertions The simplest assertion in CSP$_M$ are *refinement assertions*, which are lines of the form:

```
assert P [T= Q
```

The above will cause FDR to check whether $P \sqsubseteq_T Q$. The semantic model can be any of the following:

- The *traces model*, written as `[T=`;
- The *failures model*, written as `[F=`;
- The *failures-divergences* model, written as `[FD=`.

Deadlock Freedom It is possible to assert that a process is deadlock free, in a particular semantic model. In all cases, FDR internally converts this into a refinement assertion of the form $DF(A) \sqsubseteq P$, where `A` is the alphabet of events that `P` performs and `DF(A)` is the most non-deterministic process over `A`, i.e. $DF(A) = \sqcap_{x \in A} x \rightarrow DF(A)$. Intuitively, in the failures model, this means that the process can never get into a state where no event is offered. This assertion can be written as:

```
assert P :[deadlock free]
```

which defaults to checking the assertion in the failures-divergences model, or a particular semantic model can be specified using:

```
assert P :[deadlock free [F]]
```

which insists that the failures model will be used to check the assertion. Note that only the failures and failures-divergences models are supported for deadlock freedom assertions.

Hint: If the process `P` is known to be divergence free, then checking the deadlock freedom assertion in the failures model, instead of the failures divergences model will result in increased performance. If `P` is not known to be divergence free, then separately checking that `P` is livelock free and that `P` satisfies an appropriate (livelock free) traces or failures specification will, again, result in increased performance.

Divergence Freedom In the failures-divergences model it is also possible to check if a process can diverge, i.e. perform an infinite amount of internal work (this is also known as a livelock freedom check). FDR converts all such checks into a refinement assertion of the form $CHAOS(A) \sqsubseteq P$, where `A` is the alphabet of the process `P`. This essentially says that the process can have arbitrary behaviour, but may never diverge. This can be written as:

```
assert P :[divergence free]
```

which defaults to checking in the failures-divergences model, or a particular semantic model can be specified as follows:

```
assert P :[divergence free [FD]]
```

which insists that the failures-divergences model will be used to check the assertion. Note that only the failures-divergences model is supported for divergence freedom assertions.

Determinism FDR can be used to check if a given process is deterministic by asserting:

```
assert P :[deterministic]
```

The above assertion will check if `P` is deterministic in the failures- divergences model. As with other property checks, a specific model can be specified as follows:

```
assert P :[deterministic [FD]]
```

Note that FDR considers a process P to be deterministic providing no witness to non-determinism exists, where a witness consists of a trace tr and an event ev such that P can both accept and refuse a after tr. Formally, in the failures model, P is non-deterministic iff $\exists tr, a \cdot tr\text{^}\langle a\rangle \in traces(P) \wedge (tr, \{a\}) \in failures(P)$. In the failures-divergences model, $failures$ is replaced by $failures_{\perp}$.

Internally, for the failures-divergences model, FDR actually constructs a deterministic version of the process `P` (using *deter*) and then checks if `deter(P) [FD= P`. Thus, whilst the compilation phase can appear to take a long time to finish, the checking phase is typically faster. For the failures model, FDR uses Lazic's determinism check which runs two copies of the process in parallel. If the process contains a lot of taus, then this can cause the number of states that need to be checked to increase greatly.

Has Trace Assertions FDR has support for asserting that a process has a certain trace. For instance:

```
assert P :[has trace]: <a, b, c>
```

asserts that `P` can perform the trace `<a, b, c>` in the failures-divergences model. This assertion supports the following semantic models:

- The *traces model*, written as `:[has trace [T]]:`. This means that `P` must be able to perform this trace.
- The *failures model*, written as `:[has trace [F]]:`. This means that `P` must not be able to refuse to perform this trace. For instance:

  ```
  assert STOP |~| a -> STOP :[has trace [F]]: <a>
  ```

  would fail in the failures model, since the process may refuse to perform the `a`.
- The *failures-divergences* model, written as `:[has trace [FD]]:`. This means that `P` must not be able to refuse to perform this trace, and may not diverge whilst performing it.

The intention is that this assertion is used for simple unit tests of processes, rather than as a specification in itself.

New in version 3.3.0.

Negated Assertions FDR also allows any assertion to be negated by writing, for example, `assert not P [T= Q`. Note that if a negated assertion fails, it is not possible to debug it since this merely implies that the underlying assertion passed. Conversely, a passing negated assertion can be debugged.

Assertion Options

It is also possible to specify options to the assertions. Not all options are supported by all assertions (and there are no general rules for this in many cases), so FDR will throw an error when it is unable to support a particular assertion option.

Warning: Partial order reduction is still experimental, and improvements are still required, particularly to performance.

Partial Order Reduction FDR can use *partial order reduction* to attempt to automatically reduce the size of the state space of a system in a safe manner. For example, the assertion:

```
assert P [T= Q :[partial order reduce]
```

is precisely the same as the standard trace assertion, but FDR will attempt to automatically reduce the state space. Partial order reduction also has three difference modes `precise` (default), `hybrid` and `fast`. This achieve progressively smaller reductions in the state space, but should run faster. The mode can be selected as follows:

```
assert P [T= Q :[partial order reduce [hybrid]]
```

Partial order reduction will only work on some examples. Generally, it is simplest to try it and to see what state space reduction and time difference it causes. Further, if partial order reduction works well on one instance of a problem, it will also work on larger instances.

In general, partial order reduction will be effective on examples where there is a parallel composition where each process, or group of processes, has a number of events that are independent of the events that other processes perform. For example, *dinning philosophers* achieves excellent partial order reduction because each philosopher has a number of events that do not conflict with other philosophers (e.g. `thinks.i`, `eats.i` etc). Partial order reduction will remove the redundant interleavings.

The degree of reduction that partial order reduction can achieve is also affected by the specification in question. It is most efficient when given a *deadlock-freedom assertion* in the failures model. In general, refinement assertions will achieve less reduction. To improve the amount of reduction that can be achieved in refinement checks, the specification alphabet should be made as small as possible.

New in version 3.1.

Tau Priority Over This allows tau to be given priority over a specific set of events. For example, the assertion:

```
assert P [T= Q :[tau priority]: {tock}
```

is equivalent to the assertion:

```
assert prioritise(P, <{}, {tock})>) [T= prioritise(Q, <{}, {tock})>)
```

For further information see `prioritise`.

New in version 2.94.

4.1.9 Transparent and External Functions

A number of functions within FDR are available only if they are imported using `transparent` or `external`. Transparent functions are all applied to processes, and are semantics preserving. External functions are either non-semantics preserving process functions, or are utility functions provided by FDR. See *Compression Functions* for examples of transparent functions, and `mtransclose` for an example of an external function.

Transparent and external functions may be imported into a script as follows: *Transparent and External Functions*

4.1.10 Modules

CSP_M also supports a limited version of modules that allow code to be encapsulated, to avoid leaking. For example, the following declares a simple module `A`:

```
module A
   X = 2
   Y = X + 2
exports
   Z = 2 + Y + X
endmodule

f(x) = A::Z
```

As seen above, modules can have a number of private definitions (in this case `X` and `Y` are private and can only be referred to by `Z`), and public or *exported* definitions (in the above case, just `Z`). Exported definitions are accessible in other parts of the same script using their fully qualified name, which consists of `ModuleName::VariableName`. The general syntax for modules is as follows:

```
module ModuleName
   <private declaration list>
exports
   <public declaration list>
endmodule
```

where both declaration lists can contain any declarations with the exception of *assertions*. Modules may also be nested, as the following example shows:

```
module M1
   X = 1
exports
   Y = 1 + M2::Y

   module M2
      X = 2
   exports
      Y = 1 + X
   endmodule
endmodule

f = 1 + M1::Y + M1::M2::Y
```

Nested modules can either be public or private.

Parameterised Modules

Modules may also be defined to take arguments. For example, consider the module:

```
module M1(X)
   check(x) = member(x, X)
exports
   f(y) = if check(y) then "Inside X" else "Outside of X"

   datatype X = Y | Z
endmodule
```

The above defines a module `M1` that takes a single parameter `X`, which must be of type `{a}`, for some type `a`. The arguments in the module definition are in scope within all of the declarations defined within the module. *Instances* of the module may then be created to call the functions within a particular instance of the module. For example, if the following module instance is declared:

```
instance M2 = M1({0})
```

the expression `M2::f(0)` would evaluate to "Inside X". Note that declarations inside parametrised modules are accessible only via a module instance (thus `M1::f(0)` is an invalid expression). If a datatype is declared inside a parametrised module, then different module instances will contain distinct versions of the datatype. For example give the module instances:

```
instance M2 = M1({0})
instance M3 = M1({0})
```

`M2::X` and `M3::X` are not of the same type and hence the expression `M2::X == M3::X` would result in a type error.

More generally, a module that takes N arguments can be declared by:

```
module ModuleName(Arg1, ..., ArgN)
   <private declaration list>
exports
   <public declaration list>
endmodule
```

An instance of a module that takes N arguments can be declared as follows:

```
instance ModuleInstanceName = ModuleName(E1, ..., EN)
```

where `E1` and `EN` are expressions of a type appropriate for `ModuleName`. Instances of modules that do not take parameters can also be created simply by eliding the `(E1, ..., EN)` portion of the instance declaration.

> **Warning:** An instance declaration can only appear strictly after the definition of the module in the input file.

4.1.11 Timed Sections

Tock CSP is a discretely-timed variant of CSP that uses the event `tock` to model the passage of time. In order to specify tock-CSP processes, CSP_M includes a *timed section* syntax that automatically translates CSP processes to tock-CSP. For example:

```
channel tock

OneStep(_) = 1

Timed(OneStep) {
   P = a -> P
}

P' = a -> tock -> P' [] tock -> P'
```

will translate `P` into tock-CSP. The resulting process will, in fact, be equivalent to `P'` (see below for the full translation).

In general, timed sections are written as:

```
Timed(expression) {
   declarations
}
```

where:

`declarations` This is a list of declarations to be translated into tock-CSP. All declarations, including nested timed sections, are permitted inside timed sections (however, if this timed section is inside a *`Let`* or *module*, then the usual restrictions apply). Note that within these declarations *`timed_priority`* and *`WAIT`* may be used.

`expression` This is an expression, known as the *timing function*, of type `(`*`Event`*`) ->` *`Int`*. This function returns the number of time units that the given event takes to complete. For example, in the above example, each event is defined as taking 1 time unit (note that this can be defined using a *`Lambda`*).

Before using a timed section, `tock` must be declared as an event. Failure to do so, or defining tock as something of an incompatible type, will result in a type-checker error.

The definitions inside the timed section are translated into tock-CSP according to the following translation, assuming that the timing function is called time:

<table>
<tr><th>Process</th><th>Translated To</th></tr>
<tr><td>STOP</td><td>TSTOP, where TSTOP = tock -> TSTOP.</td></tr>
<tr><td>SKIP</td><td>TSKIP, where TSKIP = SKIP [] tock -> TSKIP.</td></tr>
<tr><td>a -> P</td><td>X = tock -> X [] a -> tock -> ... -> tock -> P where time(a) tocks occur after the a.</td></tr>
<tr><td>c?x -> P(x)</td><td>X = tock -> X [] c?x -> tock -> ... -> tock -> P(x) where time(c.x) tocks occur after the c.x.</td></tr>
<tr><td>P [] Q</td><td>P [+ {tock} +] Q</td></tr>
<tr><td>b & P</td><td>if b then P else TSTOP</td></tr>
<tr><td>P [A || B] Q</td><td>P [union({tock},A) || union({tock},B)] Q</td></tr>
<tr><td>P [| A |] Q</td><td>P [| union({tock},A) |] Q</td></tr>
<tr><td>P ||| Q</td><td>P [| {tock} |] Q</td></tr>
<tr><td>P /\ Q</td><td>P /+ {tock} +\ Q</td></tr>
<tr><td>P [+ A +] Q</td><td>P [+ union({tock}, A) +] Q</td></tr>
<tr><td>P /+ A +\ Q</td><td>P /+ union({tock}, A) +\ Q</td></tr>
</table>

The remaining operators, i.e. *Exception*, *Hide*, *Internal Choice*, *Rename*, *Sequential Composition* and *Sliding Choice*, are not affected by the translation. All the replicated operators are translated analogously to the above.

Warning: At the present time, *Linked Parallel* and the Non-deterministic Input of *Prefix* are not permitted inside timed sections.

See also:

Model checking Timed CSP The paper that fully describes the timed CSP translation.

Synchronising External Choice Used in the translation of *External Choice* into tock-CSP.

Synchronising Interrupt Used in the translation of *Interrupt* into tock-CSP.

New in version 2.94.

Changed in version 3.0: In FDR2, `tock` was automatically defined when a timed section was encountered. FDR3 requires `tock` to be defined to be defined by the user as an event, in order to allow timed sections in more contexts. In addition, FDR2 defined TOCKS as a global constant in a file that mentioned timed sections; FDR3 no longer does so.

4.1.12 Print Statements

If a CSP_M file contains a statement of the form:

```
print 1+1
print head(<5..>)
```

then FDR will add the statements to a list on the right hand side of the *session window*, thus allowing the expressions to be easily evaluated.

4.1.13 Include Files

Sometimes it can helpful to split a single CSP_M file into several files, either because some code is common to several problems, or to simply break up large file. This can be accomplished by using an `include "file.csp"` expression in the file. For example, if `file1.csp` and `file2.csp` are in the same directory, and `file1.csp` contains:

```
include "file2.csp"
f = g + 2
```

and `file2.csp` contains:

```
g = 2
```

then this is equivalent to a single file that contains:

```
g = 2
f = g + 2
```

Each `include` statement must be on a separate line and all paths are relative to the file that contains a particular `include` statement.

4.2 Functional Syntax

In this section we give a full overview of the CSP_M functional syntax. This is divided up into *Expressions*, which defines what expressions are, *Patterns*, which defines CSP_M's pattern syntax, and *Statements*, which defines what statements (which appear in the context of various comprehensions) are. The relative binding strengths of the operators is given in *Binding Strength*, and the list of reserved words is documented in *Reserved Words*.

4.2.1 Expressions

Expressions in CSP_M evaluate to values. An expression is either a process expression, which is something that uses one of CSP's process operators, or a non-process expression. In this section we define the syntax of non-process expressions. The syntax of process expressions is defined separately in *Defining Processes*.

syntax Function Application `f(e1, ..., eN)`

Applies the function `f` to the arguments `e1, ..., eN`. The arguments have to be of the type that the function expects and, further, must satisfy any type constraints imposed by the function.

syntax Binary Boolean Function `e1 and e2, e1 or e2`

Return type `<inline translatable="True">Bool</inline>`

Given two expressions `e1` and `e2` of type *`Bool`*, returns either the boolean conjunction or disjunction of the two values.

Both `and` and `or` are lazy in their second argument. Thus, `False and error("Error")` evaluates to False and `True or error("Error")` evaluates to True.

syntax Unary Boolean Function `not e`

Return type `<inline translatable="True">Bool</inline>`

Given an expression of type *`Bool`*, returns the negation of the boolean value.

syntax Comparison `e1 < e2, e1 <= e2, e1 > e2, e1 >= e2`

Return type `<inline translatable="True">Bool</inline>`

Given two values of type a type `t` that satisfies *`Ord`*, returns a boolean giving the result of comparing the two values using the selected comparison operator. The comparison performed is dictated by the argument type, as follows:

Argument Type	Type of Comparison	Examples of True Comparisons
Int	Standard integer comparison.	1 < 2, 2 < 4.
Char	Compares characters according to their integer value.	'a' < 'b', 'f' < 'z'.
<a>	Compares the sequences using the list prefix operator.	<> < <1>, <1,2> < <1,2,3>, not (<1,2> < <1,3,4>).
{a}	Compares the sets using the subset operator.	{} < {1,2}, {4,6} < {4,6,7,8}.
(e1, ..., eN)	Compares the tuples lexicographically.	(1,2) < (2,0), (1,2) < (1,3), not ((1,2) < (1,1)).
(| k => v |)	Compares the maps using the submap relation (i.e. m1 is a submap of m2 if m2 has all keys of m1 and the values are related using the appropriate comparison function).	(| |) < (| 1 => 2 |), (| 1 => 1 |) < (| 1 => 2 |), not (| 1 => 2 |) < (| 1 => 2 |), (| 1 => 2 |) <= (| 1 => 2 |).

syntax Dot `e1.e2`

Given two values of type a and type b returns the result of combining them together using the dot operator. The outcome of this operator depends upon the value of e1. If e1 is anything but a *data constructor* or *channel*, then the value e1.e2 is formed. For example, given a definition of f as f(x,y) = x.y, f(1, true) evaluates to 1.true, whilst f(STOP, false) evaluates to STOP.false. If e1 is a data constructor or channel then, informally, e1.e2 combines e2 into the partially constructed event or datatype. When doing so, it also checks that e2 is permitted by the data or channel declarations. For example, given the following definitions:

```
datatype Type = A.{0} | B.Type.{1}
```

then evaluating A.1 would throw an error, as 1 is not a value permitted by the definition of Type.

More formally, when combining e1 and e2 using ., e1 is examined to find the first *incomplete* datatype or event. For example, given the above definitions, if e1 is B.A then e2 will not be combined with the B, but instead with the A as this is the first incomplete constructor. Thus, B.A.0 evaluates to B.(A.0), whilst B.A.0.1 evalutes to B.(A.0).1, as the A is already complete.

> **Warning:** Dot is not intended for use as a general functional programming construct; it is primarily intended for use as a data or event constructor, although it is occasionally used more generally in the context of the *Prefix* operator. Using it as functional programming construct is not supported and may result in various type-checker errors. Instead, tuples and lists should be used.

syntax Equality Comparison `e1 == e2, e1 != e2`

Return type <inline translatable="True">*Bool*</inline>

Given two values of a type t that satisfies *Eq*, returns a boolean giving the result of comparing the values for equality.

syntax If `if b then e1 else e2`

Parameters

- **b** (Bool) – The branch condition.
- **e1** (*a*) – The expression to evaluate if b is true.
- **e2** (*a*) – The expression to evaluate if b is false.

Evaluates b and then evaluates e1 if b is true and otherwise evaluates e2.

syntax Lambda `\ ps @ e`
Given a comma separated list of *pattern* of type `a1`, ..., `aN` and an expression of type `b`, constructs a function of type `(a1, ..., aN) -> b`. When the function is evaluated with arguments `as`, `as` is matched against the patterns `ps` and, if it succeeds, `e` is evaluated in the resulting environment. If it fails, then an error is thrown.

syntax Let `let <declaration list> within e`
The let definition allows new definitions to be introduced that are usable only in the expression `e`. When a let expression is evaluated, the declarations are made available for `e` during evaluation. The return value of a let expression is equal to the return value of `e`.

The declaration list is formatted exactly as the top-level of a CSP_M-file is, but only *external*, *transparent*, *function*, *pattern* and *timed sections* declarations are allowed. For example, the following is a valid let expression:

```
f(xs) =
   let
      external normal
      <y>^ys = tail(xs)
   within normal(if ys == <> then STOP else SKIP)
```

syntax Literal `0.., True/False, 'c', "string"`
CSP_M allows integer literals to be written in decimal. Boolean literals can are written as `true` and `false`. Character literals are enclosed between `'` brackets whilst string literals are enclosed between `"` brackets. Characters can be escaped using the `\\` character. Thus, `'\''` evaluates to the character `'`, whilst `"\""` evaluates to the string `"`.

syntax Binary Maths Operation `e1 + e2, e1 - e2, e1 * e2, e1 / e2, e1 % e2`

Return type `<inline translatable="True">`*`Int`*`</inline>`

All maths operations take two arguments of type *`Int`* and return a value of type *Int* giving the result of the appropriate function. Note that unlike many programming languages, `e1 / e2` returns the quotient (rounding towards negative infinity), whilst `e1 % e2` returns the modulus.

syntax Unary Maths Operation `-e`

Return type `<inline translatable="True">`*`Int`*`</inline>`

Returns the negation of the expression `e`, which must be of type *`Int`*.

syntax Parenthesis `(e)`
Brackets an existing expression without altering the type or value of the expression.

syntax Tuple `(e1, ..., eN)`
Given `n` expressions of type `a1`, etc, returns a tuple of type `(a1, ..., an)`.

syntax Variable `v`
Returns the value of the variable in the current evaluation environment. These must begin with an alphabetic character and are followed by any number of alphanumeric characters or underscores optionally followed by any number of prime characters (`'`). There is no limit on the length of identifiers and case is significant. Identifiers with a trailing underscore (such as `f_`) are reserved for machine-generated code. Variables in *modules* can be accessed using the `::` operator. Thus, if `M` is a module that exports a variable `X`, then `M::X` can be used to refer to that particular `X`.

Sequences

syntax Concat `e1^e2`
This operator takes two sequences of type `<a>` and returns their concatenation. Thus, `<1,2>^<3,4> == <1,2,3,4>`. This takes time proportional to the length of `e1`.

syntax List <e1, ..., eN>
Given N expressions, each of some common type a, constructs the explicit list where the i^{th} element is ei.

syntax List Comprehension <e1, ..., eN | s1, ..., sN>
For each value generated by the *sequence statements*, s1 to sN, evaluates e1 to eN, which must all be of some common type a, and adds them to the list in order. Thus, <x, x+1 | x <- <0,2>> == <0, 1, 2, 3>. The variables bound by the statements are available for use by the ei.

syntax Infinite Int List <e..>

Return type <inline translatable="True"><*Int*></inline>

Given e, which must be of type *Int*, constructs the infinite list of all integers greater than or equal to e. Thus, <5..> == <5,6,7,...>.

syntax Infinite Ranged List Comprehension <e.. | s1, ..., sN>

Return type <inline translatable="True"><*Int*></inline>

For each value generated by the *sequence statements*, s1 to sN, evaluates e and creates the infinite list of all integers starting at e. Thus, <1.. | false> == <>, whilst <1.. | true> == <1..>. The variables bound by the statements are available for use by e.

syntax Ranged Int List <e1..e2>

Return type <inline translatable="True"><*Int*></inline>

Given e1 and e2 which must both be of type *Int*, constructs the list of all integers between e1 and e2, inclusive. For example, <5..7> == <5,6,7>.

syntax Ranged List Comprehension <e1..e2 | s1, ..., sN>

Return type <inline translatable="True"><*Int*></inline>

For each value generated by the *sequence statements*, s1 to sN, evaluates e1 and e2, which must be of type *Int*, and adds them to the list in order. Thus, <x..y | (x,y) <- <(0,1),(1,2)>> == <0, 1, 1, 2>. The variables bound by the statements are available for use by e1 and e2.

syntax List Length #e

Return type <inline translatable="True">*Int*</inline>

Given a list of type <a>, returns a value of type *Int* indicating the length of the list. This takes time proportional to the length of the list.

Sets

syntax Set {e1, ..., eN}
Given N elements of some common type a that satisfies *Set*, constructs the set containing all of the ei. Thus, for each ei, member(ei, {e1, ..., eN}) evaluates to True.

syntax Set Comprehension {e1, ..., eN | s1, ..., sN}
For each value generated by the *set statements*, s1 to sN, evaluates e1 to eN, which must all be of some common type a that satisfies *Set*, and adds them to the resulting set. Thus, {x, x+1 | x <- {0,2}} == {0, 1, 2, 3}. The variables bound by the statements are available for use by the ei.

syntax Infinite Int Set {e..}

Return type <inline translatable="True">{*Int*}</inline>

Given e, which must be of type *Int*, constructs the infinite set of all integers greater than or equal to e. Thus, {5..} == {5,6,7,...}.

syntax Infinite Ranged List Comprehension {e.. | s1, ..., sN}

Return type `<inline translatable="True">{Int}</inline>`

For each value generated by the *set statements*, `s1` to `sN`, evaluates `e` and creates the infinite set of all integers starting at `e`. Thus, `{1.. | false} == {}`, whilst `{1.. | true} == {1..}`. The variables bound by the statements are available for use by `e`.

syntax Ranged Set Comprehension `{e1..e2 | s1, ..., sN}`

Return type `<inline translatable="True">{Int}</inline>`

For each value generated by the *set statements*, `s1` to `sN`, evaluates `e1` and `e2`, which must be of type `Int`, and adds them to the set in order. Thus, `{x..y | (x,y) <- {(0,1),(1,2)}} == {0, 1, 2}`. The variables bound by the statements are available for use by `e1` and `e2`.

syntax Ranged Int Set `{e1..e2}`

Return type `<inline translatable="True">{Int}</inline>`

Given `e1` and `e2` which must both be of type `Int`, constructs the set of all integers between `e1` and `e2`, inclusive. For example, `{5..7} == {5,6,7}`.

Enumerated Sets

syntax Enumerated Set `{| e1, ..., eN |}`

Parameters **ei** (*ti =>* b*) – The i^th value to compute `productions` of.

Return type `<inline translatable="True">(Yieldable b) => {b}</inline>`

Informally, in the case that the `e1` to `eN` are channels, this operator returns the set of all possible events that can be sent along the channels `e1` to `eN`. Formally, it is equivalent to `Union({productions(e1), ..., productions(eN)})`, i.e. given `N` expressions that are data constructors or channels (possibly partially completed), constructs the set of all values of type `b` that are completions of the expressions, providing `b` satisfies `Yieldable`.

The most common use of this operator is to specify synchronisation sets. For example, suppose we have the following channel definitions:

```
channel c : {0..10}.{0..10}.{0..20}
channel d : {0..10}
channel e
```

and we wanted to put processes `P` and `Q` in parallel, synchronising on all events on channels `d` and `e`, and events on channel `c` that start with a `0`. This synchronisation alphabet can be written as `{| d, e, c.0 |}`, as this returns a set consisting of all events that are on channel `d`, all events on channel `e` (i.e. just `e` itself), and those events on channel `c` that start with a `0`.

syntax Enumerated Set Comprehension `{| e1, ..., eN | s1, ..., sM |}`

Parameters **ei** (*ti =>* b*) – The i^th value to compute `productions` of.

Return type `<inline translatable="True">(Yieldable b) => {b}</inline>`

For each value generated by the *set statements*, `s1` to `sN`, evaluates `productions(e1)` to `productions(eN)`, which must all be of some common type `a` that satisfies `Set`, and adds them to the resulting set. For example, given the following channel definitions:

```
channel c : {0..2}.{0..3}.{0}
channel e
```

Then {| c.x.(x+1), e | x <- {0,2} |} evaluates to {e, c.0.1.0, c.2.3.0}.

As with other comprehensions, the variables bound by the statements are available for use by the ei.

4.2.2 Maps

syntax Map (| k1 => v1, ..., kN => vN |)

Parameters

- **ki** (k) – The i^{th} key.
- **vi** (v) – The i^{th} value.

Return type <inline translatable="True">(| k => v |)</inline>

Constructs a map where each key is mapped to the corresponding value. For example, (| |) == emptyMap and (| 9 => 4 |) == mapFromList(<(9, 4)>). If a key appears more than once then the value that the key is mapped to is picked non-deterministically.

> **Warning:** Note that a space *must* always occur after the initial (| (thus (||) is invalid syntax). This is to ease ambiguity with interleaves and non-deterministic choices that occur after parentheses.

New in version 3.0.

4.2.3 Patterns

CSP_M also allows values to be *matched* against patterns. For example, the following function, which takes an integer, uses pattern matching to specify different behaviour depending upon the argument:

```
f(0) = True
f(1) = False
f(_) = error("Invalid argument")
```

Whilst the above could have been written as an if statement, it is much more convenient to write it in the above format. Patterns also bind variables for use in the resulting expression. For example f(<x>^xs) = e) allows the e to refer to the first element of the list as x and the tail of the list as xs.

Each of the allowed patterns is defined as follows.

syntax Concat Pattern p1^p2

Given two patterns, which must be of a common type <a>, matches a sequence where the start of the sequence matches p1 and the end of the sequence matches p2 . This binds any variable bound by p1 or p2.

> **Warning:** Not every concatenation pattern is valid as it is possible to construct ambiguous patterns. For example, the pattern xs^ys is not valid as there is no way of deciding how to decompose the list into two segments.

syntax Dot Pattern p1.p2

Matches any value of the form a.b where a matches p1 and b matches p2. This, together with *Variable Pattern* allows datatype values and events to be pattern matched. For example, suppose the following declaration is in scope:

```
datatype X = A.Int.Bool
```

Then, `A.y.True` matches any `A` data-value where the last component is `True`, and binds `y` to the integer value. Note that `.` associates to the right, and thus matching `A.0.True` to the pattern `x.y` would bind `x` to `A` and `0.True` to `y`.

> **Warning:** Pattern matching on partially constructed events or datatypes is strongly discouraged, and may be disallowed in a future release.

syntax Double Pattern `p1 @@ p2`

Given two patterns, which must be of a common type `a`, matches any value that matches both `p1` and `p2`. This binds any variable bound by `p1` or `p2`. For example, `1 @@ 2` matches no values, whilst `xs @@ (<y>^ys)` matches any non-empty list, and binds `xs` to the whole list, `y` to the head of the list and `ys` to the tails of the list.

syntax List Pattern `<p1, ..., pN>`

Given `N` patterns, which must be of a common type `a`, matches any list where the i^{th} element matches `pi`. This binds any variable bound by any of the `pi`.

syntax Literal Pattern `0..., True/False, 'c'..., "x"...`

Pattern Literals are written as *`expression literals`* are, and match in the obvious way. They do not bind any variable.

syntax Parenthesis Pattern `(p)`

This matches any value matched by `p`, and binds exactly the same variables as `p`.

syntax Set Pattern `{} or {p}`

The empty-set pattern matches only the empty set (and binds no value), whilst the singleton set pattern matches any value that is a set consisting of a single element that matches `p`.

syntax Tuple Pattern `(p1, ..., pN)`

Given `N` patterns, each of type `ai`, matches any tuple where the i^{th} element matches `pi`. This binds any variable bound by any of the `pi`.

syntax Variable Pattern `v`

If `v` is a *data constructor* or a *channel* then `v` matches only the particular data constructor or channel and binds no value. Otherwise, `v` matches anything any binds `v` to the value it was matched against. For example:

```
channel chan : {0..1}

f(chan.x) = x
```

In the above, `chan` is recognised as a channel name, and therefore matches only events of the form `chan.x`. As `x` is not a data constructor or a channel it matches anything and binds `x` to the value.

> **Warning:** As a result of the above rules, using short channel names or data constructor names is strongly discouraged. For example, if a script contains a channel definition such as `channel x : ...`, then any `x` in the script will only match the channel `x`, rather than any value.

syntax Wildcard Pattern `_`

This matches any value, and does not bind any variable.

4.2.4 Statements

In CSP_M, there are a number of comprehension constructs that generate new sets or sequences based on existing sequences or sets. For example, the `List Comprehension` `<x+1 | x <- xs>` increments every item in the list `xs` by `1`. *Statements* occur on the right hand side of such comprehensions and, conceptually, generate a sequence of values that can be consumed. The different types of sequences can be described as follows.

syntax Generator Statement `p <- e`
Given an expression `e` of type, `{a}` if this should generate sets and `<a>` if this generates sequences, and a pattern `p` of type `a`, generates all values from `e` that match the pattern `p`. Statements to the right of this may use variables bound by `p` whilst `e` may refer to variables bound to the left of it.

syntax Predicate Statement `e`
Selects only those values such that the expression `e`, which must be of type `Bool`, evaluates to `True`. `e` may refer to variables bound to the left of it.

4.2.5 Binding Strength

The binding strength for the CSP_M operators is given below as a list of the operators in descending order of binding strength (i.e. items higher in the list bind tighter). Thus, as `[]` appears below `;`, this means that `P = STOP [] STOP ; STOP` is parsed as `P = STOP [] (STOP ; STOP)`. Multiple entries on a single level means that the operators have equal binding strength, meaning that brackets may be necessary to disambiguate the meaning. The associativity of the operators is given in brackets.

1. `Parenthesis` (non-associative), `Rename` (non-associative)
2. `Concat` (left-associative)
3. `List Length` (left-associative)
4. `* / %` (left-associative)
5. `+`, `-` (left-associative)
6. `Comparison` (non-associative), `Equality Comparison` (non-associative)
7. `not` (left-associative)
8. `and` (left-associative)
9. `or` (left-associative)
10. `:` (non-associative)
11. `Dot` (right-associative)
12. `?`, `!`, `$` (all left-associative)
13. `Guarded Expression`, `Prefix` (all right-associative)
14. `Sequential Composition` (left-associative)
15. `Sliding Choice or Timeout` (left-associative)
16. `Interrupt` (left-associative)
17. `External Choice` (left-associative)
18. `Internal Choice` (left-associative)
19. `Exception`, `Generalised Parallel`, `Alphabetised Parallel` (all non-associative)
20. `Interleave` (left-associative)

21. `Hide` (left-associative)
22. *Replicated Operators* (non-associative)
23. `Double Pattern` (non-associative)
24. `Let`, `If` (non-associative)

4.2.6 Reserved Words

Certain words are reserved in CSP_M, meaning that they cannot be used as identifiers (such as variable names, function names etc). The following is a complete list of the reserved words:

1. `and`
2. `or`
3. `not`
4. `if`
5. `then`
6. `else`
7. `let`
8. `within`
9. `channel`
10. `assert`
11. `datatype`
12. `subtype`
13. `external`
14. `transparent`
15. `nametype`
16. `module`
17. `exports`
18. `endmodule`
19. `instance`
20. `Timed`
21. `include`
22. `false`
23. `true`
24. `print`
25. `type`

4.3 Defining Processes

In this section we define the various operators that are available in $\mathrm{CSP_M}$. We include only the briefest of descriptions of each operator, instead choosing to focus on $\mathrm{CSP_M}$-specific issues. For more information about each of the operators see either *The Theory and Practice of Concurrency* or *Understanding Concurrent Systems*.

Note that regular *expressions* can be mixed with the process expression, providing doing so makes type-sense. For example:

```
P(x) = if x == 0 then STOP else (STOP [] STOP)
Q(x) = let P = STOP [] STOP within P [] P
```

are both valid CSP expressions.

The relative binding strengths of the operators is given in *Binding Strength*.

4.3.1 Basic Operators

operator External Choice `P [] Q`

Blackboard $P \square Q$

Offers the choice of the initial events of `P` and `Q`, which must be of type *`Proc`*.

operator Guarded Expression `b & P`

Blackboard $b \,\&\, P$

Parameters

- **b** (`Bool`) – The process guard.
- **P** (`Proc`) – The process to behave as if `b` is true.

If `b` is true then behaves like `P`, otherwise behaves like *`STOP`*.

operator Hide `P \ A`

Blackboard $P \setminus A$

Behaves like `P`, which must be of type *`Proc`*, but if `P` performs any event from `A`, which must be of type {*`Event`*}, the event is hidden and *turned into* an internal event (i.e. a tau).

operator Internal Choice `P |~| Q`

Blackboard $P \sqcap Q$

This non-deterministically picks one of `P` and `Q`, which must be of type *`Proc`*, and then runs the chosen process.

operator Prefix `e -> P`

Blackboard $e \rightarrow P$

This process performs the event `e`, which must be of type *`Event`* and then runs `P`, which must be of type *`Proc`*. FDR also supports a number of more general forms of the prefix operator, which are particularly useful for offering the choice of several events. For example, suppose the following channel declarations are in scope:

```
channel c : {0..1}
channel d : {0..1}.Bool
```

The choice of `c.0` and `c.1` can be written as using *External Choice* as `c.0 -> P [] c.1 -> P`, or more concisely using *Prefix* as `c?x : {0, 1} -> P` or `c?x -> P` (in this case the set of allowed values is automatically deduced from the channel declaration). It is also possible to write `d.0.y -> P [] d.1.y -> P` more concisely as `d?x!y -> P`. Note that `d?x.y -> P` would not be equivalent as the second `.` would be transformed to a `?` (see *here* for more information). Writing `?x` causes a new variable `x` to be introduced that is bound to the value that was communicated. For example, a simple Echo process could be defined by `Echo(c) = c?x -> c!x -> Echo(c)`.

The general form of a Prefix consists of an expression followed by a number of *fields*, each of which matches some component of the event. In particular, the general form is e<f_1><f_2><...><f_n> where `e` is an *expression* of type `a => Event` and each `f_i` is a *field* of type `t_i` such that a = t_1 . t_2 . <...> . t_n. The available field types are as follows.

Input `?p` Offers the choice (using *External Choice*) of any value that matches the *pattern* `p`, which must be of a type that satisfies *Complete*. The set of allowed values is deduced from the channel or datatype declaration. For example, given the above channel declarations then the set of events that `c?x` ranges over is `c.0` and `c.1` as `x` matches the first field of the channel `c`.

Restricted Input `?p : S` Offers the choice (using *External Choice*) of any value from the set `S`, which must be an *expression* of type `{a}`, that matches the *pattern* `p`, which must be of type `a` and satisfy *Complete*.

Output `!e` This allows only the value of `e`, which must be an *expression*.

Non-deterministic Input `$p` This behaves exactly as `?p`, but instead offers the non-deterministic choice of the available events (using *Internal Choice*). At the present time, fields containing `$` may only appear *before* fields containing `?` or `!`.

New in version 3.0.

Non-deterministic Restricted Input `$p : S` This behaves exactly as `?p : S`, but instead offers the non-deterministic choice of the available events (using *Internal Choice*). At the present time, fields containing `$` may only appear *before* fields containing `?` or `!`.

New in version 3.0.

The set expressions in each of the fields are able to use the value of variables bound by patterns to the left of the field. This means that expressions such as `d?x?y:f(x) -> P` are allowed.

The following table gives a number of prefix forms and the explicit process that they are equivalent to.

Process	Equivalent To
`c?x -> P(x)`	`c.0 -> P(0) [] c.1 -> P(1)`
`c?x : {0} -> P(x)`	`c.0 -> P(0)`
`d?x : {0.True, 1.False} -> P(x)`	`d.0.True -> P(0.True) [] d.1.False -> P(1.False)`
`d?x!False -> P(x)`	`d.0.False -> P(0) [] d.1.False -> P(0)`
`d?x.y -> P(x,y)`	`d?x?y -> P(x,y)`
`c$x -> P(x)`	`c.0 -> P(0) |~| c.1 -> P(1)`
`d$x?y -> P(x,y)`	`(d.0.False -> P(0,False) [] d.0.True -> P(0,True)) |~| (d.1.False -> P(1,False) [] d.1.True -> P(1,True))`
`d?x?y:{x==0} -> P(x,y)`	`d.0.True -> P(0,True) [] d.1.False -> P(1,False)`

> **Warning:** Note that any `.` that occurs after a `?` essentially becomes a `?` (equally, any `.` after a `!` becomes a `!` and any `.` after a `$` becomes a `$`). For example, `d?x.y -> STOP` is not equivalent to `d?x!y`, but instead it is equivalent to `d?x?y`. This is because `.` binds tighter than `?`, meaning that `d?x.y` is bracketed as `d?(x.y)`.

operator Project `P |\ A`

Blackboard $P \upharpoonright A$

Behaves like `P`, which must be of type *`Proc`*, but if `P` performs any event *not in* `A`, which must be of type `{`*`Event`*`}`, the event is hidden and *turned into* an internal event (i.e. a tau).

See also:

`Hide` It is equivalent to `P \ diff(Events, A)`, but may be more efficient when *`Events`* is large.

operator Rename `P[[from <- to]]`

Blackboard $P[[from \leftarrow to]]$

Parameters

- **P** (`Proc`) – The process to rename.
- **from** (*a* `=>* Event`) – The channel to rename events from.
- **to** (*a* `=>* Event`) – The channel to rename events to.

The renaming operator renames all events that P performs according to the given relation. The relation is specified, as above, and causes all events of the form `from.x` that `P` performs to `to.x` (or, if `from` and `to` are events, renames `from` to `to`). For example, in the following, `P` and `Q` are equivalent:

```
channel c, d : {0..1}
channel e : Bool.{0..2}

P = (c.0 -> STOP [] d.1 -> STOP)[[c <- e.false, d <- e.true]]
Q = e.false.0 -> STOP [] e.true.1 -> STOP
```

It is also possible to combine the above using set *statements*, as follows:

```
P[[c.x <- e.false.(x+1), d.x <- e.true.(x+1) | x <- {0..1}, x == 0]]
```

This renames `c.0` to `e.false.1` and `d.0` to `e.true.1`. Note that the generators in the above must be set generators.

> **Warning:** If `from` and `to` are channels, rather than events, care must be taken to ensure that the renaming will not result in invalid events being created, otherwise a runtime error will occur. For example, given the above definitions, evaluating:
>
> ```
> (d.true.2 -> STOP)[[d.true <- c]]
> ```
>
> would result in an error, as `2` is not in the set of values that can be sent down `d`.

Note: Unlike the blackboard CSP operator, the renaming relation is not required to be total. Events that are not in the domain of the renaming relation are unaffected by the renaming.

4.3.2 Parallel Operators

operator Alphabetised Parallel `P [A || B] Q`

Blackboard $P \;{}_{A}\|_{B}\; Q$

Parameters

- **P** (Proc) – The left process.
- **A** ({Event}) – The set of events that P is allowed to perform.
- **B** ({Event}) – The set of events that Q is allowed to perform.
- **Q** (Proc) – The right process.

Runs P and Q in parallel, allowing P to only perform events from A, Q to only perform events from B and forcing P and Q to synchronise on $A \cap B$. Equivalent to `(P [| diff(Events, A) |] STOP) [| inter(A, B) |] (Q [| diff(Events, B) |] STOP)`.

See also:

Enumerated Sets For details on how to easily construct synchronisation sets.

operator Generalised Parallel `P [| A |] Q`

Blackboard $P \parallel_{A} Q$

Parameters

- **P** (Proc) – The left process.
- **A** ({Event}) – The synchronisation alphabet.
- **Q** (Proc) – The right process.

Runs P and Q in parallel forcing them to synchronise on events in A. Any event not in A may be performed by either process.

See also:

Enumerated Sets For details on how to easily construct synchronisation sets.

operator Interleave `P ||| Q`

Blackboard $P \,|||\, Q$

Parameters

- **P** (Proc) – The left process.
- **Q** (Proc) – The right process.

Runs P and Q in parallel without any synchronisation. Equivalent to `P [| {} |] Q`.

See also:

Enumerated Sets For details on how to easily construct synchronisation sets.

4.3.3 Replicated Operators

FDR also has *replicated*, or *indexed*, version of a number of operators. These provide an easy way to construct a process that consists of a number of processes composed using the same operator. For example, suppose `P :: (Int) -> Proc` then `||| x : {0..2} @ P(x)` evaluates `P` for each value of `x` in the given set and then interleaves them. Thus, the above is equivalent to `P(0) ||| P(1) ||| P(2)`.

The general form of a replicated operator is `op <statements> @ P` where `op` is a piece of operator syntax, statements are a list of *Statements* and `P` is the process definition (which can make use of the variables bound by the statements). Each of the operators evaluates `P` for each value the statements take before composing them together using `op`.

operator Replicated Alphabetised Parallel `|| <set statements> @ [A] P`

Evaluates `P` and `A` for each value of the *Statements* and composes the resulting processes together using `Alphabetised Parallel`, where each process has the corresponding alphabet `A`. If the resulting set of processes is empty then this evaluates to `SKIP`. For example, in the following `Q` and `R` are equivalent:

```
channel a : {0..3}

P(x) = a.x -> STOP
A(x) = {a.x}

Q = || x : {0..3} @ [A(x)] P(x)
R = a.0 -> STOP ||| a.1 -> STOP ||| a.2 -> STOP ||| a.3 -> STOP
```

operator Replicated External Choice `[] <set statements> @ P`

Replicated external choice evaluates `P` for each value of the *Statements* and composes the resulting processes together using `External Choice`. If the resulting set of processes is empty (e.g. `[] x : {} @ x`), then `STOP` is returned.

Hint: In CSP_M there is no way of writing a process equivalent to the blackboard CSP $?ev : X \rightarrow P(ev)$, which offers the choice of all events ev that are in X, just using the prefixing operator. However, using the replicated external choice operator, an equivalent process can be defined as `[] ev : X @ ev -> STOP`.

operator Replicated Generalised Parallel `[| A |] <set statements> @ P(x)`

Evaluates `P` for each value of the *Statements* and composes the resulting processes together using `Generalised Parallel`, synchronising them on the set `A`. If the resulting set of processes is empty then this evaluates to `SKIP`.

operator Replicated Interleave `||| <set statements> @ P(x)`

Evaluates `P` for each value of the *Statements* and composes the resulting processes together using `Interleave`. If the resulting set of processes is empty then this evaluates to `SKIP`.

operator Replicated Internal Choice `|~| <set statements> @ P(x)`

Replicated internal choice evaluates `P` for each value of the *Statements* and composes the resulting processes together using `Internal Choice`. If the resulting set of processes is empty then an error is thrown.

operator Replicated Linked Parallel `[l <-> r] <sequence statements> @ P(x)`

Evaluates `P`, `l` and `r` for each value of the *Statements* and composes the resulting processes together in the same order as the statements using `Linked Parallel`. In particular suppose `P`, `l` and `r` evaluate to `P1`, `P2`, etc then the following process is constructed `P1 [l1 <-> r1] P2 [l2 <-> r2] ....` If the resulting sequence of processes is empty then an error is thrown.

As with `Linked Parallel`, the linking of events may be specified using comprehensions. For example:

```
channel c, d : Bool

Q(0) = c.True -> STOP
Q(1) = d.True -> STOP

P = [c.x <-> d.x | x <- <False, True>, x] y : <0,1> @ Q(y)
```

then `P` is equivalent to `Q(0) [c.True <-> d.True] Q(1)`.

operator Replicated Sequential Composition ; `<sequence statements> @ P(x)`
Evaluates `P` for each value of the *Statements* and composes the resulting processes together in the same order as the statements using *Sequential Composition*. If the resulting sequence of processes is empty then this evaluates to *SKIP*.

operator Replicated Synchronising Parallel `[+ A +] <set statements> @ P(x)`
Evaluates `P` for each value of the *Statements* and composes the resulting processes together using *Synchronising External Choice*. If the resulting sequence of processes is empty then this evaluates to *STOP*.

New in version 2.94.

4.3.4 Advanced Operators

operator Exception `P [| A |> Q`

Blackboard $P \Theta_A Q$

Parameters

- **P** (`Proc`) – The initial process.
- **A** (`{Event}`) – The set of *exception* events.
- **Q** (`Proc`) – The process to behave like once an exception has been thrown.

This process initially behaves like `P`, but if `P` ever performs an event from the set `A`, then the process `Q` is started.

New in version 2.91.

operator Interrupt `P /\ Q`

Blackboard $P \triangle Q$

This operator initially behaves like `P ||| Q`, but if any action of `Q` is performed `P` is discarded and the process behaves as `Q`. Both `P` and `Q` must be of type *Proc*.

operator Linked Parallel `P [c <-> d] Q`

Blackboard $P[c \leftrightarrow d]Q$

Parameters

- **P** (`Proc`) – The left process.
- **Q** (`Proc`) – The right process.
- **c** (*a* `=>* Event`) – The channel of the left process to synchronise.
- **d** (*a* `=>* Event`) – The channel of the right process to synchronise.

Linked parallel runs `P` and `Q` in parallel, forcing them to synchronise on the `c` and `d` events and then hides the synchronised events. Assuming that `f` is a fresh name it is equivalent to `(P[[c <- f]] [| {| f |} |]`

`Q[[d <- f]]) \ {| f |}`. However, compiling linked parallel will be significantly faster than the above simulation.

As with `Rename`, multiple channels may be linked and set *statements* may be used to specify the linking. For example, `P [a <-> b, c <-> d] Q` and `P [a.x <-> b.x, c.x <-> d.x | x <- X, f(x)] Q` are both valid syntax. Again, as with `Rename`, care must be taken to ensure that the channels have the same allowed values, otherwise a runtime error will occur.

operator Sequential Composition `P ; Q`

Blackboard $P\,;Q$

Behaves like `P` until it terminates (by turning into `SKIP`) at which point `Q` is run. Both `P` and `Q` must be of type `Proc`.

operator Sliding Choice or Timeout `P [> Q`

Blackboard $P \triangleright Q$

Initially it offers the choice of the initial events of `P`, but can non-deterministically change into a state where only the events of `Q` are available. If `P` performs a tau to `P'`, then `P [> Q` will perform a tau transition to `P' [> Q` (i.e. timeout is not resolved by taus). Both `P` and `Q` must be of type `Proc`.

operator Synchronising External Choice `P [+ A +] Q`

Blackboard $P \,\square_A\, Q$

This operator is a hybrid of `External Choice` and :op'Generalised Parallel'. Initially it offers the initial events of both `P` and `Q`, which must be of type `Proc`. As with `Generalised Parallel`, `P` and `Q` are required to synchronise on events from `A`, which must be of type `{Event}`. If either `P` or `Q` performs any event not in `A` (including a tick or a tau) then the operator behaves like `External Choice` and discards the argument that did not do an event. Thus, `[+ A +]` is resolved only by performing events not in `A`, whilst events from `A` are performed simultaneously by both branches. For example, given the following:

```
P = a -> b -> STOP [+ {a} +] a -> c -> STOP
Q = a -> (b -> STOP [] c -> STOP)
```

`P` and `Q` are equivalent.

New in version 2.94.

operator Synchronising Interrupt `P /+ A +\ Q`

Blackboard $P \,\triangle_A\, Q$

This operator is analogous to `Synchronising External Choice` in that it is a hybrid of `Interrupt` and `Generalised Parallel`. Initially it offers the initial events of both `P` and `Q`, which must be of type `Proc`. As with `Generalised Parallel`, `P` and `Q` are required to synchronise on events from `A`, which must be of type `{Event}`. As with `Interrupt`, any event that `P` performs does not resolve the operator (except for tick, which terminates it). If `Q` ever performs a visible event that is not in `A` then this resolves the choice and the process behaves as `Q`. For example, given the following:

```
P = a -> b -> STOP /+ {a} +\ a -> c -> STOP
Q = a -> (b -> STOP /\ c -> STOP)
R = a -> (b -> c -> STOP [] c -> STOP)
```

`P`, `Q` and `R'` are equivalent.

New in version 2.94.

4.4 Type System

One of the new features of FDR3 is a builtin *type-checker*. The type-checker is not complete in that it will reject some programs that are correctly typed, but permits virtually all *reasonable* CSP_M scripts. In this section we document the type system, starting with the *type atoms*, then documenting the *constructed types* before lastly documenting the *type constraints*.

4.4.1 Type Atoms

`type a`
A type variable that represents some type. Like `variables`, these must begin with an alphabetic character and are followed by any number of alphanumeric characters or underscores optionally followed by any number of prime characters (`'`). There is no limit on the length of type variables and case is significant.

`type Bool`
The type of boolean values, i.e. `True` and `False`.

`type Char`
The type of characters. All Unicode (or equivalently ISO/IEC 10646) characters are supported.

`type Datatype` n
The type of a user defined datatype. For example, given the following:

```
datatype X = Y.{0..1} | Z
```

then `Z :: n` and `Y.0 :: n`.

`type Event`
The type of fully constructed events. For example, given the following definitions:

```
channel c : {0..1}
channel done
```

then `c.0 :: Event` and `done :: Event`.

`type Int`
The type of integers. This is defined as supporting values between $-2^{31}+1$ and $2^{31}-1$, inclusive.

`type Proc`
The type of constructed processes, such as `STOP`.

4.4.2 Type Constructors

`type Dot` a.b
The type of two items `x :: a` and `y :: b` that have been combined together using the dot operator. For example, `1.1 :: Int.Int`.

`type Dotable` a => b
If `x :: a => b` then `x` can be dotted with something of type `a` to yield a value of type `b`. Thus, if `y :: a` then `x.y :: b`. This type is used in channel and datatype declarations as follows:

```
channel c : Int.Int
datatype X = Y.Int.Bool
```

In the above, `c :: Int=>Int=>Event` and `Y :: Int=>Bool=>X`.

type Extendable a =>* b

A value of type `a =>* b` is something that can be *extended* to a value of type `b`. In particular, it is either of type `b`, or is something of type `a => b`. Note that if something is of type `a =>* b` then `a` is guaranteed to be an atomic type variable and `b` will satisfy *`Yieldable`*. This type most commonly appears in the context of *enumerated sets*. For example if `f(x) = {|x|}` then `f :: Yieldable b => (a=>*b) -> {b}`.

type Function (C_1 a_1, C_2 a_2, . . .) => (a_1, a_2, . . . , a_n) -> b

The type of a function that takes `n` arguments of type `a_1`, `a_2`, respectively, each of which satisfies the appropriate *type constraints* `C_1`, `C_2` etc, and returns something of type `b`. For example, given the following definitions:

```
plus(x, y) = x + y
singleton(x) = {x}
```

then `plus :: (Int, Int) -> Int` and `singleton :: Set a => (a) -> {a}`.

type Map (| k => v |)

The type of a map from type `k` to type `v`.

New in version 3.0.

type Sequence <a>

The type of sequences where each element is of type `a`.

type Set {a}

The type of sets that contain items of type `a`. Constructing a set requires the inner type to satisfy *`Set`*.

type Tuple (a_1, a_2, . . . , a_n)

Something of type `(a_1, a_2, ..., a_n)` is a tuple where the i^{th} item is of type `a_i`.

4.4.3 Type Constraints

type constraint Eq

A type satisfying `Eq` can be compared using `==`. All of the *type atoms* except for *`Proc`* satisfy *`Eq`*, although *`Datatype`* only satisfies *`Eq`* if every field satisfies *`Eq`*. For example, if `Z` was a datatype defined as `datatype Z = A.Proc | B.Int`, then `Z` would not satisfy *`Eq`* as *`Proc`* does not. All *constructed types*, except for *`Function`*, satisfy *`Eq`* providing their type arguments do so.

type constraint Complete

A type satisfying `Complete` is something that is not of the type `a => b`. That is, it represents a value that cannot be extended to a datatype or channel by applying *`Dot`*.

type constraint Ord

A type satisfying `Ord` can be compared using `<`, `<=`, `>=` and `>`. Of the *type atoms*, only *`Char`* and *`Int`* satisfy *`Ord`*. Of the *constructed types*, *`Map`*, *`Sequence`*, *`Set`* and *`Tuple`* satisfy *`Ord`* providing all their type arguments satisfy the type constraint *`Eq`* (**bold** not *`Ord`*).

See also:

`Comparison` for more details on the ordering relation that each type uses.

type constraint Set

This indicates that a type variable must be something that sets can contain. Any type that satisfies *`Eq`* also satisfies *`Set`* but, in addition, *`Proc`* also satisfies *`Set`*. Further, All *constructed types*, except for *`Function`*, satisfy *`Set`* providing their type arguments do so.

Other functional languages do not generally require such a type constraint, as their set implementations merely require that items are comparable using *`Eq`*, and possibly *`Ord`*. In CSP_M this is not an option as processes are not comparable, but it is often useful (and indeed necessary) to construct sets of processes.

Note that some of the *set functions* require the set to contain items that satisfy `Eq`. This is done to prevent the equality of two processes being checked via other means, such as via the function `areEqual(p1, p2) = card({p1, p2}) == 1`, which checks if `p1` and `p2` are equal.

type constraint Yieldable
This type constraint is satisfied only by `Event` and `Datatype`. It indicates that the type variable must be something that can be yielded by dotting together several values.

4.4.4 Binding Strength

The binding strength for the CSP_M type constructors is given below as a list of the constructors in descending order of binding strength (i.e. items higher in the list bind tighter). Multiple entries on a single level means that the operators have equal binding strength, meaning that brackets may be necessary to disambiguate the meaning. The associativity of the operators is given in brackets.

1. `Dot` (right-associative)
2. `Dotable`, `Extendable` (all right-associative)

4.5 Built-In Definitions

4.5.1 Constants

constant Bool :: `{Bool}`
The set of all booleans, i.e. `Bool = {True, False}`.

constant Char :: `Char`
The set of all characters. See `Char` for more details on the range of characters supported.

constant Events :: `{Event}`
The set of all events that are currently defined. For example, if a CSP_M file included the following definitions:

```
channel a : {0, 1}
channel b
```

then `Events` would evaluate to `{a.0, a.1, b}`.

constant Int :: `{Int}`
The set of all integers. See `Int` for more details on the range of integers supported.

constant Proc :: `Proc`
The set of all processes that are defined. This set may not be manipulated in any way, but is provided to allow processes to be used in datatypes:

```
datatype X = C.Proc

f = C.STOP
```

constant False :: `Bool`
Evaluates to the literal `false`.

constant True :: `Bool`
Evaluates to the literal `true`.

4.5.2 Set Functions

function card :: (Eq *a*) => `({a}) -> Int`
Returns the cardinality, or size, of the given set. For example, `card({}) = 0` and `card({0}) = 1`.

function diff :: (Set *a*) => `({a}, {a}) -> {a}`
Returns the relative complement of two sets (i.e. $X \setminus Y$ is written as `diff(X,Y)`). For example, `diff({1}, {1}) = {}` and `diff({1,2}, {1}) = {2}`.

function empty :: (Eq *a*) => `({a}) -> Bool`
Returns true if the set is empty.

function inter :: (Set *a*) => `({a}, {a}) -> {a}`
Returns the intersection of the two sets.

function Inter :: (Set *a*) => `({{a}}) -> {a}`
Given a set of sets, returns the intersection of all of the sets. For example, `Inter({{1}, {1,2}, {1,2, 3}}) = {1}`.

function member :: (Eq *a*) => `(a, {a}) -> Bool`
Returns true if the given element is a member of the given set.

function seq :: (Eq *a*) => `({a}) -> <a>`
Returns a sequence consisting of all the elements in the given set. The ordering is undefined, although equal sets will return their elements in the same order.

New in version 2.91.

function Seq :: (Set *a*) => `({a}) -> {<a>}.`
Given a set, returns the set of all finite sequences of elements from the set. If the input set is non-empty, then the output of this function is always infinite.

function Set :: (Set *a*) => `({a}) -> {{a}}`
Returns the powerset of the input set.

function union :: (Set *a*) => `({a}, {a}) -> {a}`
Returns the union of the two sets.

function Union :: (Set *a*) => `({{a}}) -> {a}`
Given a set of sets, returns the union of all the sets. For example, `Union({{1}, {2}}) = {1, 2}`.

4.5.3 Sequence Functions

function concat :: `(<<a>>) -> <a>`
Given a sequence of sequence, concatenates all the sequences into one. For example, `concat(<<1>, <2>, <3>>) = <1, 2, 3>`.

function elem :: (Eq *a*) => `(a, <a>) -> <a>`
Returns true if the first argument occurs anywhere in the given list.

function head :: `(<a>) -> a`
Returns the first element of the given list, throwing an error if the list is empty.

function length :: `(<a>) -> Int`
Returns the length of list.

function null :: `(<a>) -> Bool`
True if the list is empty.

function set :: (Set *a*) => `(<a>) -> {a}`
Converts the given list into a set.

function tail :: (<a>) -> <a>

Returns the tail of the list starting at the second element, throwing an error if the list is empty. Thus, xs = <head(xs)>^tail(xs).

4.5.4 Map Functions

function emptyMap :: (Set *k*) => Map k v

Returns an empty map. O(1).

New in version 3.0.

function mapDelete :: (Set *k*) => (Map k v, k) -> Map k v

Removes the specified key from the map if it is present. O(log n).

New in version 3.2.

function mapFromList :: (Set *k*) => (<(k, v)>) -> Map k v

Constructs a map given a sequence of associative pairs. If the same key appears more than once, then the last value in the list will be used (i.e. later entries will overwrite earlier entries). O(n log n).

New in version 3.0.

function mapLookup :: (Set *k*) => (Map k v, k) -> v

Returns the value associated with the specified key in the map. If the key is not in the domain of the map then an error is thrown. O(log n)

New in version 3.0.

function mapMember :: (Set *k*) => (Map k v, k) -> *Bool*

Returns true if the key is in the map. O(log n).

New in version 3.0.

function mapToList :: (Set *k*) => (Map k v) -> <(k, v)>

Converts a map to a sequence of associative pairs. O(n).

New in version 3.0.

function mapUpdate :: (Set *k*) => (Map k v, k, v) -> Map k v

Inserts the specified key value pair into the map, overwriting the current value if the specified key is already in the map. O(log n).

New in version 3.0.

function mapUpdateMultiple :: (Set *k*) => (Map k v, <(k, v)>) -> Map k v

Inserts each key-value pair into the map (as per mapUpdate). If the same key appears more than once, then the last value in the list will be used (i.e. later entries will overwrite earlier entries). O(m log n), where m is the length of the list to be inserted.

New in version 3.0.

function Map :: (Set *k*, Set *v*) => ({k}, {v}) -> {Map k v}

Returns the set of all maps from the given domain to the given image.

New in version 3.0.

4.5.5 Error Handling

function error :: (<*Char*>) -> a

Displays the specified string as an error message. For exmaple:

```
f(0) = error("f is not defined for 0.")
f(n) = n-1
```

New in version 3.0.

function show :: `(a) -> <Char>`
Converts the specified object to a string in a human-readable format. For example:

```
f(x) = show(x)^" % 2 == "^show(x % 2)
```

would print `4 %2 == 0` when `f(4)` is called.

New in version 3.0.

4.5.6 Processes

function CHAOS :: `({Event}) -> Proc`
`CHAOS(A)` offers the non-deterministic choice over all events in `A`, but may also deadlock at any point.

function DIV :: `Proc`
`DIV` immediately diverges. It is equivalent to `DIV = STOP [> DIV`, or `DIV = |~| _ : {0} @ DIV`.

external **function loop** :: `(Proc) -> Proc`
`loop(P)` computes a process that repeatedly runs the given process using `Sequential Composition`. In particular, `loop(P)` is equal to `X`, where `X = P ; X`.

Note: This function was required in prior versions of FDR to enable a more optimised representation for certain processes. The new compiler included in 3.0 automatically recognises and optimises definitions of the form `X = P ; X`, negating the need for this function to be used.

---

function RUN :: `({Event}) -> Proc`
`RUN(A)` offers the choice over all events in `A`, perpetually.

constant SKIP :: `Proc`
The process that immediately terminates. Thus, `SKIP ; P` = `P`, for all `P`.

constant STOP :: `Proc`
The process that offers no events and therefore represents a deadlocked process. It is equivalent to `[] _ : {} @ error("This is not called.")`.

function WAIT :: `(Int) -> Proc`
`WAIT(n)` performs precisely `n` `tock` events and then behaves like `SKIP`. Note that this function is only in scope within *timed sections*.

4.5.7 Compression Functions

FDR has a number of *compression functions* that can be applied to processes to either attempt to reduce the number of states that a process has (and thus make refinement checking faster), or to change the semantics in useful ways (cf. `chase` or `prioritise`).

Generally, compressions that are designed to reduce the number of states a process has are best applied to the arguments of parallel compositions, or other similar operators. There is no point applying one of the semantics-preserving compressions to the top-level of a process since the compression will need to visit every state of the inner process in order to calculate the compressed machine. Clearly this is precisely what the original refinement check would have done. For example:

```
-- Potentially a useful compression
assert normal(Q) ||| normal(R)
-- Not a useful compression
assert normal(Q ||| R) :[deadlock free]
```

Further, note that by default FDR applies compressions to the arguments of parallel compositions, meaning there is no need to apply compression to such processes. By default, FDR uses `sbisim` to compress the leaf machines, but the compression used can be configured using `compiler.leaf_compression`.

The best method of assessing the effectiveness of compressions is to use the *machine structure viewer*, which can display details regarding the number of states and transitions that are saved by compression. In general, when attempting to reduce the state space of a process, the best compression functions to try are `dbisim`, `diamond` (followed by `sbisim`), `normal`, and `wbisim`. It is often hard to predict which compression function will perform well on a given process: in general, it is best to perform several experiments to determine which results in the best reduction in the state space size. In general, `diamond` followed by `sbisim` is the fastest of the compressions to perform. On most systems, `wbisim` only gives a small extra reduction in the number of states versus `dbisim`, which is a very effective compression. When `normal` works efficiently it generally achieves very good compression ratios, but unlike the other compressions, it can actually cause the number of states to increase.

The compression functions functions are all transparent or external functions, and thus need to be explicitly imported. See *Transparent and External Functions* for more information. Compressions that are semantics preserving are transparent, whilst compressions that alter the semantics and are thus unsafe are marked as external.

The algorithms that are used to compute the compressions detailed below are described in *Understanding Concurrent Systems*.

external **function chase** :: `(Proc) -> Proc`

Not semantics preserving - this should only be used when it has been proven that its application is safe, or if the behaviour is desired.

`chase(P)` is best defined by considering how to chase an individual state `s` of `P`. If `s` can perform a tau to some state `s'`, then `chase(s) = chase(s')`. If there are multiple tau transitions, then the tau transition that is picked is not defined: it is picked according to internal implementation details. If `s` cannot perform a tau, then `chase(s) = s`. For example, if `P = a -> STOP |~| b -> STOP` then `chase(P)` could either equal `a -> STOP` or `b -> STOP`. Thus, `chase` is a form of manual partial order reduction on invisible events.

`chase` is primarily useful when the result of performing one tau is known to not cause other taus to become unavailable. For example, consider `chase((a -> STOP ||| b -> STOP) \ {a, b})`: applying `chase` to this system does not change the semantics, since whenever a hidden `a` or `b` occurs the other action is still allowed in the resulting state.

This compression is lazy, in that it is computed on-the-fly as the process is explored. It can therefore be sensibly applied at the top-level of a system.

Warning: Applying `chase` to divergent processes is unsafe any may cause FDR to crash (it will certainly cause checking not to terminate if a divergent state is reached by `chase`).

Warning: As with `chase`, this function will not perform well once the bounds of RAM have been exceeded. This will be addressed in a future release of FDR3.

external **function chase_nocache** :: `(Proc) -> Proc`

Not semantics preserving - this should only be used when it has been proven that its application is safe.

This behaves exactly as per `chase`, but optimises the internal representation to cache less information and thus consume less memory (and will give a small speed up). This should only be used if it is relatively unlikely that a state of the chased machine will be visited twice.

external **function deter** :: `(Proc) -> Proc`

Not semantics preserving - this should only be used when it has been proven that its application is safe, or if the behaviour is desired.

This is only valid in the failures-divergences model, and returns a deterministic version of the process using the algorithm specified in *Understanding Concurrent Systems*. This is used internally by FDR to implement determinism checking in the failures-divergences model, and is unlikely to be of more general use.

transparent **function diamond** :: `(Proc) -> Proc`

`diamond` returns a new process where essentially, given a state `s` of the original process, as many as possible of the transitions of states that are reachable via a series of taus from `s` are added to `s` itself. The idea behind this is that it will potentially allow states that are reachable via chains of taus to be elided.

It is a useful compression to apply since it can never increase the size of a compressed process (unlike `normal`), results in a LTS that contains no taus (which can be a useful property), and is often quick to compute. Further, the output of `diamond` is guaranteed to contain no taus.

`diamond` can only be used in the traces, failures and failures-divergences models. It applies `tau_loop_factor` and `explicate` as preprocessing steps.

The output of this often benefits from being strongly bisimulated. In particular, it may be useful to define the function `sbdia(P) = sbisim(diamond(P))` to use as a compression function.

transparent **function dbisim** :: `(Proc) -> Proc`

`dbisim(P)` computes the maximal delay bisimulation of `P` and then returns and LTS that has been reduced using this. In particular, it identifies any states of `P` that can perform the same visible events after performing zero or more taus, and such that performing any such event leads to states, that are also delay bisimilar. This differs from `wbisim` in that it does not require the states immediately reached after performing the visible event to be bisimilar, but instead allows zero or more taus to occur.

`dbisim` is slower than `sbisim` to compute, but can achieve substantially more compression than `sbisim` on some processes. It is faster than `wbisim` to compute, but `wbisim` can achieve a small amount of extra compression on some processes.

New in version 3.0.

transparent **function explicate** :: `(Proc) -> Proc`

If the provided machine is a *high-level machine*, converts it to a *low-level machine*. Whilst the transitions of this machine will be accessible more quickly, the resulting machine will usually use up far more memory and the time taken to do the explication will exceed the time taken to simply access the original machine's transitions. Therefore, this is only of use when FDR's *compilation* algorithm incorrectly selects the level to compile a machine at. It is also used as a preprocessing stage for a number of the other compressions.

transparent **function lazyenumerate** :: `(Proc) -> Proc`

This behaves like `explicate`, but lazily computes the low-level transition as it is being used, rather than up-front. It is not normally necessary to use this compression function, but it is used internally by a number of compression functions.

external **function failure_watchdog** :: `(Proc, {Event}, Event) -> Proc`

This function returns the failures watchdog of the given process, as specified in *Watchdog Transformations for Property-Oriented Model-Checking*. In particular, `failure_watchdog(P, evs, bark)` alters `P` and returns a process `P'` such that, in any state `s` of `P` that offers events `inits`, instead offers $evs \cup inits$ and, if any event in $evs \setminus inits$ is performed, transitions to a process equivalent to `bark -> STOP`. Further, the watchdog will also monitor the failures of the process it is put in parallel with, and will deadlock if the process has a disallowed failure.

This transformation can be used to turn a refinement check of the form:

```
assert Spec [F= Impl
```

into a check:

```
assert failure_watchdog(Spec, A, bark) [| A |] Impl :[deadlock free [F]]
```

assuming that `A` is the alphabet of `Impl`. The usefulness of this is as per *`trace_watchdog`*. However, note that unlike `trace_watchdog`, `failure_watchdog` can cause the size of the specification to dramatically increase and therefore the resulting check can be slower. In particular, specifications that require lots of events to be simultaneously offered will be less efficient (conversely, specifications that are very nondeterministic will be more efficient).

New in version 3.1.

transparent **function normal** :: `(Proc) -> Proc`

`normal(P)` returns a normalised version of `P` in which there are no taus and such that each state contains at most one transition labelled with a particular event. For example, the process:

```
P = a -> P |~| a -> STOP
```

is normalised to the process `P'` where:

```
P' = a -> Q'
Q' = a -> Q'
```

where `Q'` is labelled with the refusals `{}`, `{a}` (i.e. either `Q'` refuses nothing, or it refuses `a`).

The output of `normal` automatically has `sbisim` applied to it.

external **function prioritise** :: `(Proc, <{Event}>) -> Proc`

Not semantics preserving - this should only be used when it has been proven that its application is safe, or if the behaviour is desired.

`prioritise(P, <S1, ..., SN>)` takes a process and a non-empty sequence of disjoint sets of events (note that the latter condition is not checked). It returns a machine whose operational semantics have been altered to only allow actions from `Si` providing no event from `Sj` is offered for all `j < i`. Note that tau and tick are implicitly added to `S1`. Any event that is not in any of the `Si` is unaffected by this transformation. For example, in the following each `Pi` is equivalent to `Qi`:

```
P1 = prioritise(a -> STOP [] b -> STOP, <{a}, {b}>)
Q1 = a -> STOP

P2 = prioritise(a -> STOP |~| b -> STOP, <{a}, {b}>)
Q2 = a -> STOP |~| b -> STOP

P3 = prioritise(a -> STOP [> b -> STOP, <{}, {a}>)
Q3 = b -> STOP
```

Note that this compression is lazy and can therefore be applied to the outer level of a system (indeed, this is the most common usage). If there is only one set of events specified then the above function is the identity function.

This function is most commonly used when modelling timed systems using *tock CSP* since `prioritise(P, {},{tock})` ensures that `tock` cannot occur when tau or tick are available.

Note that `prioritise(P, ...)` cannot be applied to processes `P` that contain certain compressions that are not *prioritise safe*. The only compressions that are prioritise safe are `dbisim`, `sbisim` and `wbisim`. FDR will detect unsafe applications of compressions and report these as errors when necessary.

New in version 2.94.

Changed in version 3.0: In FDR2, this function took either a variable number of arguments or a sequence of sets. In FDR3 this function has been changed to only allow a sequence of sets to be passed.

> **Warning:** As with `chase`, this function will not perform well once the bounds of RAM have been exceeded. This will be addressed in a future release of FDR3.

external **`function prioritise_nocache`** :: `(Proc, <{Event}>) -> Proc`

Not semantics preserving - this should only be used when it has been proven that its application is safe, or if the behaviour is desired.

This behaves exactly as per `prioritise`, but optimises the internal representation to cache less information and thus consume less memory (and will give a small speed up). This should only be used if it is relatively unlikely that a state of the chased machine will be visited twice.

New in version 2.94.

Changed in version 3.0: In FDR2, this function took either a variable number of arguments or a sequence of sets. In FDR3 this function has been changed to only allow a sequence of sets to be passed.

external **`function prioritisepo`** :: `(Proc, {Event}, {(Event, Event)}, {Event}) -> Proc`

Not semantics preserving - this should only be used when it has been proven that its application is safe, or if the behaviour is desired.

`prioritisepo(P, E, O, M)` takes a process and a specification of a partial order (the format of which is defined below). It returns a machine whose operational semantics have been altered to only events `e` providing no event above `e` in the specified partial order is offered. This is essentially a more advanced version of `prioritise` that permits additional control.

The partial order is specified by three sets: `E`, `O`, and `M`. `E` is the set of set of all events that should be prioritised: events not in `E` will be unaffected by `prioritisepo`. `O` is a set of pairs such that if `(u, l)` is present in `O`, then `l` is defined as being below `u` in the ordering (i.e. `u` > `l`, and thus `u` would be prioritised over `l`). The actual order used is the transitive closure of the order `O`. Any event in `E` is assumed to be below tau and tick, but the set `M` can be used to specify which events should be considered equal to tau and tick in the ordering. For example, in the following each `Pi` is equivalent to `Qi`:

```
P1 = prioritisepo(a -> STOP [] b -> STOP, {a,b}, {(a, b)}, {})
Q1 = a -> STOP

P2 = prioritisepo(a -> STOP |~| b -> STOP, {a, b}, {(a, b)}, {})
Q2 = a -> STOP |~| b -> STOP

P3 = prioritisepo(a -> STOP [> b -> STOP, {a}, {}, {})
Q3 = b -> STOP

P4 = prioritisepo(a -> STOP [> b -> STOP, {a}, {}, {a})
Q4 = a -> STOP [> b -> STOP
```

As per `prioritise`, this compression is lazy and can therefore be applied to the outer level of a system (indeed, this is the most common usage). If there is only one set of events specified then the above function is the identity function.

As per `prioritise`, `prioritisepo(P, ...)` cannot be applied to processes `P` that contain certain compressions that are not *prioritise safe*. The only compressions that are prioritise safe are `dbisim`, `sbisim` and `wbisim`. FDR will detect unsafe applications of compressions and report these as errors when necessary.

New in version 3.2.

> **Warning:** As with `chase`, this function will not perform well once the bounds of RAM have been exceeded. This will be addressed in a future release of FDR3.

transparent **function sbisim** :: (*Proc*) -> *Proc*

`sbisim(P)` computes the maximal strong bisimulation of `P` and then returns and LTS that has been reduced using this. In particular, this means that it identifies states that have *identical* behaviour. In particular, it identifies any states of `P` that can perform the same events and such that performing them leads to states that are also strongly bisimilar.

Generally, `sbisim` is able to compress a process very quickly, but will often not reduce the number of states that much compared to other compressions. It is automatically applied to the result of `normal`, is often useful to apply to the result of `diamond`, and is the default compression used by FDR on leaf machines (see `compiler.leaf_compression`).

transparent **function tau_loop_factor** :: (*Proc*) -> *Proc*

`tau_loop_factor(P)` identifies any states that are on a tau loop (i.e. it identifies any two states s and s' such that s can reach s' via a sequence of taus and s' can reach s via a sequence of taus).

Generally this compression is not that useful on its own, but it is used as a preprocessing step by a number of other compressions.

external **function trace_watchdog** :: (*Proc*, {*Event*}, *Event*) -> *Proc*

This function returns the traces watchdog of the given process, as specified in *Watchdog Transformations for Property-Oriented Model-Checking*. In particular, `trace_watchdog(P, evs, bark)` alters `P` and returns a process `P'` such that, in any state `s` of `P` that offers events `inits`, instead offers $evs \cup inits$ and, if any event in $evs \setminus inits$ is performed, transitions to a process equivalent to `bark -> STOP`. For example, `trace_watchdog(a -> STOP, {a, b}, bark)` is equivalent to the process:

```
a -> (a -> bark -> STOP [] b -> bark -> STOP)
[] b -> bark -> STOP
```

This transformation can be used to turn a refinement check of the form:

```
assert Spec [T= Impl
```

into a check:

```
assert STOP [T= (trace_watchdog(Spec, A, bark) [| A |] Impl) \ A
```

assuming that `A` is the alphabet of `Impl`. There are two reasons why this may be useful. Firstly, it allows various hierarchical compression techniques to be applied to the combination of the specification and the implementation. Further, if FDR finds a counterexample to the transformed assertion then, providing `Spec` is deterministic, it will be possible to divide this into behaviours of both the specification and the implementation, thus allowing a limited form of specification debugging to occur.

> **Note:** Counterexample can only be divided through `trace_watchdog` when the check is being done in the traces model. Counterexamples for stronger models cannot be divided through `trace_watchdog` since there is no guarantee that the behaviour is indeed a behaviour of the old machine.

New in version 3.1.

function timed_priority :: (*Proc*) -> *Proc*

Not semantics preserving - this should only be used when it has been proven that its application is safe, or when the behaviour is desired.

`timed_priority(P)` is equivalent to `prioritise(P, <{}, {tock}>)` and thus gives priority to tau and tick over `tock`. Note that this function is only in scope within *timed sections* and, in contrast to the other compression functions, does not need to be imported using `transparent` or `external`.

New in version 2.94.

Changed in version 3.0: In FDR2, this function was available anywhere in a file that used the *timed section* syntax. In FDR3, this is only available within a timed section itself.

transparent **function wbisim** :: `(Proc) -> Proc`

`wbisim(P)` computes the maximal weak bisimulation of `P` and then returns and LTS that has been reduced using this. In particular, it identifies any states of `P` that can perform the same visible events after performing zero or more taus, and such that performing any such event leads to states, again after possibly performing some more taus, that are also weakly bisimilar. This differs from `dbisim` in that it does not require the states immediately reached after performing the visible event to be bisimilar, but instead allows zero or more taus to occur.

`wbisim` is slower than both `dbisim` and `sbisim`, but is able to achieve a small amount of extra compression compared to `dbisim` (which in turn can achieve more compression that `sbisim`). In the worst case, it will take twice as long as `dbisim` to compute.

New in version 2.94.

4.5.8 Relation Functions

external **function mtransclose** :: (`Eq` *a*) => `({(a, a)}, {a}) -> {(a, a)}`

Given a relation `R`, expressed as a set of pairs, computes the symmetric transitive closure of the relation. Then, for each element of the second set `S` it computes a representative member of its equivalence class (under the symmetric transitive closure). It returns a set of tuples consisting of each element from `S` along with its representative (nb. the representative is the first element of the pair), providing the element is not equal to its representative, in which case it is omitted.

external **function relational_image** :: (`Eq` *a*, `Set` *b*) => `({(a, b)}) -> (a) -> {b}`

Given a relation, expressed as a set of pairs, returns a function that takes an element of the domain of the relation and returns the set of all elements of the image that it is related to. For example, `relational_image({(1, 2), (1,3)})(1) = {2,3}`.

This function benefits from being partially applied to its first argument.

external **function relational_inverse_image** :: (`Set` *a*, `Eq` *b*) => `({(a, b)}) -> (b) -> {a}`

This function is the opposite to relational_image. In particular, given a relation, expressed as a set of pairs, it returns a function that takes an element of the *image* of the relation and returns the set of all elements of the domain that it is related to. For example, `relational_inverse_image({(2,1), (3,1)})(1) = {2,3}`.

external **function transpose** :: (`Set` *a*, `Set` *b*) => `({(a, b)}) -> {(b, a)}`

Given a relation, expressed as a set of tuples, returns the transpose of the relation. For example, `transpose({(1,2)}) = {(2,1)}`.

4.5.9 Dot-Related Functions

function extensions :: (`Set` *a*, `Yieldable` *b*) => `(a => b) -> {a}`

Given a partially completed datatype or channel definition `d`, this returns the set of all `x` such that `d.x` is a completed datatype or channel definition. For example, given the following definitions:

```
datatype T = X.Bool
channel c : Int.T
```

`extensions(c.0) = {X.False, X.True}` and `extensions(c.0.X) = {false, true}`.

Changed in version 3.0: In previous versions of FDR `extensions` could be called on fully completed events and datatypes. This is no longer the case as it would cause the type system to be undecidable.

function productions :: (`Set` *b*, `Yieldable` *b*) => `(a =>* b) -> {b}`

Much like *extensions*, given a partially completed datatype or channel definition `d`, this returns the set of all `d.x` such that `d.x` is a completed datatype or channel definition. For example, given the following definitions:

```
datatype T = X.Bool
channel c : Int.T
```

`productions(c.0) = {c.0.X.False, c.0.X.True}` and `productions(c.0.X) = {c.0.X.false, c.0.X.true}`.

4.6 Profiling

FDR provides a tool `cspmprofiler` that is a time sampling profiler for CSP_M. This means it is possible to attribute how much time is spent in each function within a file whilst checking a refinement assertion, allowing performance problems to be diagnosed.

`cspmprofiler` can be invoked on the command line by passing it a single CSP_M file:

```
$ cspmprofiler file.csp
```

`cspmprofiler` will evaluate all processes contained in the assertions in the specified file whilst profiling how much time is taken by each function call. Once this has completed, FDR will output a summary of the profiling data, including the total time taken, the total number of allocations performed, a list of the most expensive functions, as well as a hierarchical breakdown of where time and allocations were spent.

For example, suppose `cspmprofiler` was invoked on *Dining Philosophers* (i.e. by running `cspmprofiler phils6.csp`). This would outptu:

```
Exploring phils6.csp
Exploring SYSTEM :[deadlock free [F]]
Exploring SYSTEMs :[deadlock free [F]]
Exploring BSYSTEM :[deadlock free [F]]
Exploring ASSYSTEM :[deadlock free [F]]
Exploring ASSYSTEMs :[deadlock free [F]]




TOTAL RUNTIME     0.01 secs   (10 ticks @ 1000 us, 1 processor)
TOTAL ALLOCATIONS 10,886,392 bytes  (excludes profiling overheads)


TOP FUNCTIONS

COST CENTRE ENTRIES %time  %alloc
AlphaP      24      100.0  31.6
ASPHILS     1       0.0    0.7
ASPHILSs    1       0.0    0.7
```

(continues on next page)

(continued from previous page)

```
ASSYSTEM    1       0.0    3.3
ASSYSTEMs   1       0.0    3.3
AlphaF      24      0.0    21.1
BSYSTEM     1       0.0    0.7
BUTLER      11      0.0    9.9
FORK        30      0.0    7.9
FORKNAMES   1       0.0    0.0


ALL FUNCTIONS

COST CENTRE ENTRIES  individual          inherited
                    %time  %alloc      %time  %alloc
SYSTEMs     1       0.0    2.0         100.0  34.9
 AlphaP     12      100.0  15.8        100.0  15.8
 AlphaF     12      0.0    10.5        0.0    10.5
 FORK       6       0.0    0.7         0.0    0.7
 PHILs      12      0.0    5.9         0.0    5.9
 union      6       0.0    0.0         0.0    0.0
ASPHILS     1       0.0    0.7         0.0    2.0
 LPHIL      2       0.0    1.3         0.0    1.3
 PHIL       5       0.0    0.0         0.0    0.0
ASPHILSs    1       0.0    0.7         0.0    1.3
 LPHILs     2       0.0    0.7         0.0    0.7
 PHILs      5       0.0    0.0         0.0    0.0
ASSYSTEM    1       0.0    3.3         0.0    3.3
ASSYSTEMs   1       0.0    3.3         0.0    3.3
BSYSTEM     1       0.0    0.7         0.0    10.5
 BUTLER     11      0.0    9.9         0.0    9.9
eats        1       0.0    0.0         0.0    0.0
FORKNAMES   1       0.0    0.0         0.0    0.0
FORKS       1       0.0    0.7         0.0    1.3
 FORK       6       0.0    0.7         0.0    0.7
getsup      1       0.0    0.0         0.0    0.0
N           1       0.0    0.0         0.0    0.0
PHILNAMES   1       0.0    0.0         0.0    0.0
picks       1       0.0    0.0         0.0    0.0
putsdown    1       0.0    0.0         0.0    0.0
sits        1       0.0    0.0         0.0    0.0
SYSTEM      1       0.0    2.0         0.0    43.4
 AlphaF     12      0.0    10.5        0.0    10.5
 AlphaP     12      0.0    15.8        0.0    15.8
 FORK       18      0.0    6.6         0.0    6.6
 PHIL       12      0.0    8.6         0.0    8.6
 union      6       0.0    0.0         0.0    0.0
thinks      1       0.0    0.0         0.0    0.0
```

We explain the output of each section in turn.

Firstly, the output lists which processes it explored during the profiling. If there were a large number of processes, it would also display its progress. It then outputs a summary of how long it took to explore all the processes, and the total number of bytes allocated during the exploration.

The first import section is the `TOP FUNCTIONS` section. This is a list of the 10 functions in the script that take the most time overall. In this case, we can see that `AlphaP` is the most expensive function, in fact taking `100%` of the overall execution time. (This example is really too small for profiling to show anything notable.)

The last section displays the time taken by each individual function hierarchically. For example, consider the first few lines

```
COST CENTRE ENTRIES  individual         inherited
                    %time  %alloc     %time  %alloc
SYSTEMs      1      0.0    2.0        100.0  34.9
 AlphaP      12     100.0  15.8       100.0  15.8
 AlphaF      12     0.0    10.5       0.0    10.5
 FORK        6      0.0    0.7        0.0    0.7
 PHILs       12     0.0    5.9        0.0    5.9
 union       6      0.0    0.0        0.0    0.0
```

The columns are as follows:

- `COST CENTRE ENTRIES` displays the CSP_M name to which this row refers to.
- `ENTRIES` indicates how many times each entry was called.
- `individual %time` refers to the amount of time was spent in this function. It excludes any time made during calls to other functions.
- `individual %alloc` refers to the number of memory allocations that were made in this function. It excludes any allocations made during calls to other functions.
- `inherited %time` refers to the total amount of time that was spent in this function, *including* inside any calls to other functions. For example, for `SYSTEMs` we see that the inherited time is `100%`, but the individual time is `0%`. This is because the inherited time includes the time inside the recursive call to `AlphaP`.
- `inherited %alloc` this is similar to `inherited %time`, but for allocations.

The hierarchy displays which functions are called by which other functions. For example, the above indicates that `SYSTEMs` calls `AlphaP` etc. As another example, consider the following profile:

```
COST CENTRE ENTRIES

f           1
 g          12
  h         12
```

This indicates that `f` calls `g` which calls `h`.

Note that because `cspmprofiler` is a sampling profiler, this means that erroneous results can be obtained (although this happens rarely in practice). This is because a sampling profiler periodically records the function at the top of the execution stack. Clearly it is possible that, somehow, the profiler always picks the wrong moment to record this and therefore creates incorrect results.

Internally, `cspmprofiler` is implemented using the GHC's profiler for Haskell.

CHAPTER

FIVE

INTEGRATING FDR INTO OTHER TOOLS

FDR can also be integrated into other tools by utilising either a simple machine-readable output format, or using the more powerful API. These both allow for FDR to be used as a verification back-end for other tools. These APIs both expose various capabilities, including the ability to run assertions and then inspect the counterexamples. There are two ways of integrating with FDR:

1. Use the *machine-readable output* of the command-line tool.
2. Use the *FDR API*, which is available for C++, Java, and Python.

For anything but the most trivial applications, we strongly recommend the use of the API over the command-line tool. This is because the API allows for much finer control over FDR, such as choosing which assertions to run and when. Further, whilst the machine-readable interface will not be augmented in the future, the API can be extended if additional functionality is required.

Warning: Note that you may not bundle FDR with your tools (i.e. include a copy of FDR as part of the download of your own tool). You must instead direct your users to download FDR directly from the website, thus ensuring that the user is aware of the FDR license terms.

5.1 The FDR API

`libfdr` is a 64-bit only library available for C++, Java, and Python that exposes part of FDR's internals to external tools. This API is designed to be stable, and we will endeavour to make backwards-incompatible changes only when absolutely necessary. `libfdr` currently exposes functionality that allows files to be loaded, particular assertions to be executed, counterexamples to be interrogated, and arbitrary expressions to be evaluated. As such, it exposes a strict superset of functionality as compared to the machine-readable output of the command-line interface.

As a simple example, the following python loads a file, executes all assertions, and then begins to inspect any counterexamples:

```
session = fdr.Session()
session.load_file("phils8.csp")

# Evaluate an expression
print session.evaluate_expression("<0,1>^<2,3>", None).result

# Run the assertions
for assertion in session.assertions():
    assertion.execute(None)

    for counterexample in assertion.counterexamples():
```

(continues on next page)

(continued from previous page)

```
        debug_context = fdr.DebugContext(counterexample, True)
        debug_context.initialise(None);

        spec = debug_context.specification()
        impl = debug_context.implementation()
```

Full API documentation for the C++ version is available. Stub API documentation for the Java version is available. At the present time there is no full documentation available for the Java and Python APIs, however, the C++ documentation should be a suitable reference. In particular, the Python function names are identical, and the Java function names are simply converted into camel-case (e.g. `node_path()` becomes `nodePath()`).

If you are interested in additional functionality please contact us and describe the additional functionality that you would like.

5.1.1 Getting Started with C++

In order to use the C++ API, a C++11 compatible compiler, such as `g++` 4.8, or `clang++` 3.1 is required. The following file demonstrates the basics of using `libfdr`. Firstly, libfdr is initialised, then `Session` is created, into which the file `phils8.csp` is loaded. Lastly, all the assertions are executed.

```cpp
#include <fdr/fdr.h>
#include <iostream>

int main(int argc, char** argv)
{
    FDR::library_init(&argc, &argv);

    try
    {
        FDR::Session session;
        session.load_file("phils8.csp");
        for (const std::shared_ptr<FDR::Assertions::Assertion>& assertion
                : session.assertions())
        {
            assertion->execute(nullptr);
            std::cout << assertion->to_string() << " "
                << (assertion->passed() ? "Passed" : "Failed");
        }
    }
    catch (const FDR::Error& error)
    {
        std::cout << error.what() << std::endl;
    }

    FDR::library_exit();

    return 0;
}
```

If FDR is installed at /opt/fdr (which it is by default on Linux), and the above file is saved as `main.cc`, then the file above can be compiled and linked using:

```
g++ -std=c++11 -I/opt/fdr/include -L/opt/fdr/lib -D_GLIBCXX_USE_CXX11_ABI=0
    -o libfdr_demo main.cc
    -Wl,-rpath=/opt/fdr/lib -lfdr
```

A more interesting example is included in *API Examples*. The C++ API is fully documented and is available here.

Note that all strings used by libfdr are UTF-8 encoded. The use of the boost.nowide is strongly recommended.

ABI Compatibility

Different C++ implementations are not always compatible with each other. Below, we list the compiler version that `libfdr` was compiled with, and also versions that it will be compatible with.

On Linux, `libfdr` was compiled with `g++` 4.8. We believe that this should be compatible with all C++-11 compliant releases of `g++`. It should also be compatible with any other C++11 compliant compiler that also uses libstdc++.

On Mac OS X, `libfdr` was compiled using the system-provided `clang++` using `libc++` as the standard library. This means that when compiling under Mac OS X, the compiler flag `-stdlib=libc++` must be used. Failure to do so will lead to various link errors.

On Windows, `libfdr` was compiled using `g++` 4.8 using the mingw-w64 build with POSIX threads and Structured Exception Handling (SEH). Unfortunately, on Windows, compiler compatibility is currently very limited. We hope to provide ABI compatibility with MSVC at some point in the future.

5.1.2 Getting Started with Java

FDR also provides a Java interface, which was generated using SWIG and provides equivalent functionality to the C++ interface. In order to use the FDR Java interface, at least Java 1.6 is required. The following file gives a simple example of how the API can be used. This initialises FDR, loads the file `phils8.csp`, and then executes all of the assertions in the file.

```
import uk.ac.ox.cs.fdr.*;

package FDRDemo {

public static void main(String[] argv)
{
    try {
        Session session = new Session();
        session.loadFile("phils8.csp");
        for (Assertion assertion : session.assertions()) {
            assertion.execute(null);
            System.out.println(assertion.toString()+" "+
                (assertion.passed() ? "Passed" : "Failed"));
        }
    }
    catch (InputFileError error) {
        System.out.println(error);
    }
    catch (FileLoadError error) {
        System.out.println(error);
    }

    fdr.libraryExit();
}

}
```

The above example can be compiled and executed as follows, assuming that FDR has been installed into its usual location on Linux:

```
javac -classpath /opt/fdr/lib/fdr.jar FDRDemo.java
java -classpath /opt/fdr/lib/fdr.jar:. FDRDemo
```

A more interesting example is included in *API Examples*, and can be compiled in similar fashion. The Java API has stub documentation available here, and therefore the C++ documentation may also be useful to refer to.

> **Warning:** Many of the classes contain a `delete()` method. This is considered an internal implementation detail, and as such should not be manually called. It is likely to be removed in a future release.

5.1.3 Getting Started with Python

FDR also provides a Python interface, which was generated using SWIG and provides equivalent functionality to the C++ interface. In order to use the FDR Python interface, at least Python 2.6 is required. The following file gives a simple example of how the API can be used. This initialises FDR, loads the file `phils8.csp`, and then executes all of the assertions in the file.

```
import os
import platform
import sys

if platform.system() == "Linux":
    for bin_dir in os.environ.get("PATH", "").split(":"):
        fdr4_binary = os.path.join(bin_dir, "fdr4")
        if os.path.exists(fdr4_binary):
            real_fdr4 = os.path.realpath(os.path.join(bin_dir, "fdr4"))
            sys.path.append(os.path.join(os.path.basename(os.path.basename(real_
↪fdr4)), "lib"))
            break
elif platform.system() == "Darwin":
    for app_dir in ["/Applications", os.path.join("~", "Applications")]:
        if os.path.exists(os.path.join(app_dir, "FDR4.app")):
            sys.path.append(os.path.join(app_dir, "FDR4.app", "Contents", "Frameworks
↪"))
            break

import fdr

fdr.library_init()

session = fdr.Session()
try:
    session.load_file("phils8.csp")
    for assertion in session.assertions():
        assertion.execute(None)
        print "%s: %s" % (assertion, "Passed" if assertion.passed() else "Failed")
catch FDRError, e:
    print e

fdr.library_exit()
```

The above example can be executed using `python fdr_demo.py`.

A more interesting example is included in *API Examples*, and can be executed in similar fashion.

5.2 API Examples

For each language a simple command line tool has been produced that will check all assertions in a file, and then print the counterexample that is produced. This includes dividing the counterexample into sub-behaviours of the various components of the system. You may use these files as the basis of your integration into FDR3.

The files below (namely, `command_line.cc`, `CommandLine.java` and `command_line.py`) are hereby placed in the public domain. This means that any parts of these files may be incorporated into your own files that you then license under different means.

5.2.1 C++

Compilation instructions: `g++ -std=c++11 -I/opt/fdr/include -L/opt/fdr/lib -lfdr -o libfdr_demo command_line.cc`.

`Download`

```cpp
#include <iostream>

#include <fdr/fdr.h>

/// Pretty prints the specified behaviour to stdout
static void describe_behaviour(
    const FDR::Session& session,
    const FDR::Assertions::DebugContext& debug_context,
    const FDR::Assertions::Behaviour& behaviour,
    unsigned int indent,
    const bool recurse)
{
    // Describe the behaviour type
    std::cout << std::string(indent, ' ') << "behaviour type: ";
    indent += 2;
    if (dynamic_cast<const FDR::Assertions::ExplicitDivergenceBehaviour*>(&behaviour))
        std::cout << "explicit divergence after trace";
    else if (dynamic_cast<const FDR::Assertions::IrrelevantBehaviour*>(&behaviour))
        std::cout << "irrelevant";
    else if (auto loop =
            dynamic_cast<const FDR::Assertions::LoopBehaviour*>(&behaviour))
        std::cout << "loops after index " << loop->loop_index();
    else if (auto min_acceptance =
        dynamic_cast<const FDR::Assertions::MinAcceptanceBehaviour*>(&behaviour))
    {
        std::cout << "minimal acceptance refusing {";
        for (const FDR::LTS::CompiledEvent event : min_acceptance->min_acceptance())
            std::cout << session.uncompile_event(event)->to_string() << ", ";
        std::cout << "}";
    }
    else if (auto segmented =
        dynamic_cast<const FDR::Assertions::SegmentedBehaviour*>(&behaviour))
    {
        std::cout << "Segmented behaviour consisting of:\n";
        // Describe the sections of this behaviour. Note that it is very
        // important that false is passed to the the descibe methods below
        // because segments themselves cannot be divded via the DebugContext.
        // That is, asking for behaviour_children for a behaviour of a
        // SegmentedBehaviour is not allowed.
```

(continues on next page)

(continued from previous page)

```
        for (const std::shared_ptr<FDR::Assertions::Behaviour>& child :
                segmented->prior_sections())
            describe_behaviour(session, debug_context, *child, indent + 2, false);
        describe_behaviour(session, debug_context, *segmented->last(),
            indent + 2, false);
    }
    else if (auto trace = dynamic_cast<const FDR::Assertions::TraceBehaviour*>(&behaviour))
        std::cout << "performs event " << session.uncompile_event(trace->error_event())->to_string();
    std::cout << std::endl;

    // Describe the trace of the behaviour
    std::cout << std::string(indent, ' ') << "Trace: ";
    for (const FDR::LTS::CompiledEvent event : behaviour.trace())
    {
        if (event == FDR::LTS::INVALID_EVENT)
            std::cout << "-, ";
        else
            std::cout << session.uncompile_event(event)->to_string() << ", ";
    }
    std::cout << std::endl;

    // Describe any named states of the behaviour
    std::cout << std::string(indent, ' ') << "States: ";
    for (const std::shared_ptr<FDR::LTS::Node>& node : behaviour.node_path())
    {
        if (node == nullptr)
            std::cout << "-, ";
        else
        {
            std::shared_ptr<FDR::Evaluator::ProcessName> process_name =
                session.machine_node_name(*behaviour.machine(), *node);
            if (process_name == nullptr)
                std::cout << "(unknown), ";
            else
                std::cout << process_name->to_string() << ", ";
        }
    }
    std::cout << std::endl;

    // Describe our own children recursively
    if (recurse)
    {
        for (const std::shared_ptr<FDR::Assertions::Behaviour>& child :
                debug_context.behaviour_children(behaviour))
        {
            describe_behaviour(session, debug_context, *child, indent + 2, true);
        }
    }
}

/// Pretty prints the specified counterexample to stdout
static void describe_counterexample(
    const FDR::Session& session,
    const FDR::Assertions::Counterexample& counterexample)
{
```

(continues on next page)

(continued from previous page)

```
    // Firstly, just print a simple description of the counterexample
    std::cout << "Counterexample type: ";
    if (dynamic_cast<const FDR::Assertions::DeadlockCounterexample*>(
            &counterexample))
        std::cout << "deadlock";
    else if (dynamic_cast<const FDR::Assertions::DeterminismCounterexample*>(
                &counterexample))
        std::cout << "determinism";
    else if (dynamic_cast<const FDR::Assertions::DivergenceCounterexample*>(
                &counterexample))
        std::cout << "divergence";
    else if (auto min_acceptance =
            dynamic_cast<const FDR::Assertions::MinAcceptanceCounterexample*>(
                &counterexample))
    {
        std::cout << "minimal acceptance refusing {";
        for (const FDR::LTS::CompiledEvent event : min_acceptance->min_acceptance())
            std::cout << session.uncompile_event(event)->to_string() << ", ";
        std::cout << "}";
    }
    else if (auto trace =
            dynamic_cast<const FDR::Assertions::TraceCounterexample*>(
                &counterexample))
        std::cout << "trace with event " << session.uncompile_event(
            trace->error_event())->to_string();
    else
        std::cout << "unknown";
    std::cout << std::endl;

    // In order to print the children we use a DebugContext. This allows for
    // division of behaviours into their component behaviours, and also ensures
    // proper alignment amongst the child components.
    if (auto refinement_counterexample =
            dynamic_cast<const FDR::Assertions::RefinementCounterexample*>(
                &counterexample))
    {
        std::cout << "Children:" << std::endl;
        FDR::Assertions::DebugContext debug_context(*refinement_counterexample,
            false);
        debug_context.initialise(nullptr);
        describe_behaviour(session, debug_context,
            *debug_context.root_behaviours()[0], 2, true);
        describe_behaviour(session, debug_context,
            *debug_context.root_behaviours()[1], 2, true);
    }
    else if (auto property_counterexample =
            dynamic_cast<const FDR::Assertions::PropertyCounterexample*>(
                &counterexample))
    {
        std::cout << "Children:" << std::endl;
        FDR::Assertions::DebugContext debug_context(*property_counterexample,
            false);
        debug_context.initialise(nullptr);
        describe_behaviour(session, debug_context,
            *debug_context.root_behaviours()[0], 2, true);
    }
}
```

(continues on next page)

(continued from previous page)

```

/// The actual main function.
static int real_main(int& argc, char**& argv)
{
    if (!FDR::has_valid_license())
    {
        std::cout << "Please run refines or FDR4 to obtain a valid license before␣
↪running this." << std::endl;
        return EXIT_FAILURE;
    }

    std::cout << "Using FDR version " << FDR::version() << std::endl;

    if (argc != 2)
    {
        std::cerr << "Expected exactly one argument." << std::endl;
        return EXIT_FAILURE;
    }

    const std::string file_name = argv[1];

    std::cout << "Loading " << file_name << std::endl;

    FDR::Session session;
    try
    {
        session.load_file(file_name);
    }
    catch (const FDR::FileLoadError& error)
    {
        std::cout << "Could not load. Error: " << error.what() << std::endl;
        return EXIT_FAILURE;
    }

    // Check each of the assertions
    for (const std::shared_ptr<FDR::Assertions::Assertion>& assertion
            : session.assertions())
    {
        std::cout << "Checking: " << assertion->to_string() << std::endl;
        try
        {
            assertion->execute(nullptr);
            std::cout
                << (assertion->passed() ? "Passed" : "Failed")
                << ", found " << assertion->counterexamples().size()
                << " counterexamples" << std::endl;

            // Pretty print the counterexamples
            for (const std::shared_ptr<FDR::Assertions::Counterexample>&
                    counterexample : assertion->counterexamples())
            {
                describe_counterexample(session, *counterexample);
            }
        }
        catch (const FDR::InputFileError& error)
        {
            std::cout << "Could not compile: " << error.what() << std::endl;
```

(continues on next page)

(continued from previous page)

```
                return EXIT_FAILURE;
            }
        }

    return EXIT_SUCCESS;
}

int main(int argc, char** argv)
{
    FDR::library_init(&argc, &argv);

    int return_code = real_main(argc, argv);

    FDR::library_exit();

    return return_code;
}
```

5.2.2 Java

Compilation instructions:

```
javac -classpath /opt/fdr/lib/fdr.jar CommandLine.java
java -classpath /opt/fdr/lib/fdr.jar:. CommandLine
```

Download

```
import java.io.File;
import java.io.PrintStream;
import uk.ac.ox.cs.fdr.*;

public class CommandLine {

public static void main(String argv[]) {
    int returnCode;
    try {
        returnCode = realMain(argv);
    }
    finally {
        // Shutdown FDR
        fdr.libraryExit();
    }

    System.exit(returnCode);
}

/**
 * The actual main function.
 */
private static int realMain(String[] argv)
{
    PrintStream out = System.out;

    if (!fdr.hasValidLicense())
    {
```

(continues on next page)

(continued from previous page)

```
        out.println("Please run refines or FDR4 to obtain a valid license before␣
↪running this.");
        return 1;
    }

    out.println("Using FDR version "+ fdr.version());

    if (argv.length != 1) {
        out.println("Expected exactly one argument.");
        return 1;
    }

    String fileName = argv[0];
    out.println("Loading "+ fileName);

    Session session = new Session();
    try {
        session.loadFile(fileName);
    }
    catch (FileLoadError error) {
        out.println("Could not load. Error: "+error.toString());
        return 1;
    }

    // Check each of the assertions
    for (Assertion assertion : session.assertions()) {
        out.println("Checking: "+assertion.toString());
        try {
            assertion.execute(null);
            out.println(
                (assertion.passed() ? "Passed" : "Failed")
                +", found "+(assertion.counterexamples().size())
                +" counterexamples");

            // Pretty print the counterexamples
            for (Counterexample counterexample : assertion.counterexamples()) {
                describeCounterexample(out, session, counterexample);
            }
        }
        catch (InputFileError error) {
            out.println("Could not compile: "+error.toString());
            return 1;
        }
    }

    return 0;
}

/**
 * Pretty prints the specified counterexample to out.
 */
private static void describeCounterexample(PrintStream out, Session session,
    Counterexample counterexample)
{
    // Firstly, just print a simple description of the counterexample
    //
    // This uses dynamic casting to check the assertion type.
```

(continues on next page)

(continued from previous page)

```
    out.print("Counterexample type: ");
    if (counterexample instanceof DeadlockCounterexample)
        out.println("deadlock");
    else if (counterexample instanceof DeterminismCounterexample)
        out.println("determinism");
    else if (counterexample instanceof DivergenceCounterexample)
        out.println("divergence");
    else if (counterexample instanceof MinAcceptanceCounterexample)
    {
        MinAcceptanceCounterexample minAcceptance =
            (MinAcceptanceCounterexample) counterexample;
        out.print("minimal acceptance refusing {");
        for (Long event : minAcceptance.minAcceptance())
            out.print(session.uncompileEvent(event).toString() + ", ");
        out.println("}");
    }
    else if (counterexample instanceof TraceCounterexample)
    {
        TraceCounterexample trace = (TraceCounterexample) counterexample;
        out.println("trace with event "+ session.uncompileEvent(
            trace.errorEvent()).toString());
    }
    else
        out.println("unknown");

    out.println("Children:");

    // In order to print the children we use a DebugContext. This allows for
    // division of behaviours into their component behaviours, and also ensures
    // proper alignment amongst the child components.
    DebugContext debugContext = null;

    if (counterexample instanceof RefinementCounterexample)
        debugContext = new DebugContext((RefinementCounterexample) counterexample, false);
    else if (counterexample instanceof PropertyCounterexample)
        debugContext = new DebugContext((PropertyCounterexample) counterexample, false);

    debugContext.initialise(null);
    for (Behaviour root : debugContext.rootBehaviours())
        describeBehaviour(out, session, debugContext, root, 2, true);
}

/**
 * Prints a vaguely human readable description of the given behaviour to out.
 */
private static void describeBehaviour(PrintStream out, Session session,
    DebugContext debugContext, Behaviour behaviour, int indent,
    boolean recurse)
{
    // Describe the behaviour type
    printIndent(out, indent); out.print("behaviour type: ");
    indent += 2;
    if (behaviour instanceof ExplicitDivergenceBehaviour)
        out.println("explicit divergence after trace");
    else if (behaviour instanceof IrrelevantBehaviour)
```

(continues on next page)

(continued from previous page)

```
            out.println("irrelevant");
        else if (behaviour instanceof LoopBehaviour)
        {
            LoopBehaviour loop = (LoopBehaviour) behaviour;
            out.println("loops after index " + loop.loopIndex());
        }
        else if (behaviour instanceof MinAcceptanceBehaviour)
        {
            MinAcceptanceBehaviour minAcceptance = (MinAcceptanceBehaviour) behaviour;
            out.print("minimal acceptance refusing {");
            for (Long event : minAcceptance.minAcceptance())
                out.print(session.uncompileEvent(event).toString() + ", ");
            out.println("}");
        }
        else if (behaviour instanceof SegmentedBehaviour)
        {
            SegmentedBehaviour segmented = (SegmentedBehaviour) behaviour;
            out.println("Segmented behaviour consisting of:");
            // Describe the sections of this behaviour. Note that it is very
            // important that false is passed to the the descibe methods below
            // because segments themselves cannot be divded via the DebugContext.
            // That is, asking for behaviourChildren for a behaviour of a
            // SegmentedBehaviour is not allowed.
            for (TraceBehaviour child : segmented.priorSections())
                describeBehaviour(out, session, debugContext, child, indent + 2, false);
            describeBehaviour(out, session, debugContext, segmented.last(),
                indent + 2, false);
        }
        else if (behaviour instanceof TraceBehaviour)
        {
            TraceBehaviour trace = (TraceBehaviour) behaviour;
            out.println("performs event " +
                session.uncompileEvent(trace.errorEvent()).toString());
        }

        // Describe the trace of the behaviour
        printIndent(out, indent); out.print("Trace: ");
        for (Long event : behaviour.trace())
        {
            // INVALIDEVENT indiciates that this machine did not perform an event at
            // the specified index (i.e. it was not synchronised with the machines
            // that actually did perform the event).
            if (event == fdr.INVALIDEVENT)
                out.print("-, ");
            else
                out.print(session.uncompileEvent(event).toString() + ", ");
        }
        out.println();

        // Describe any named states of the behaviour
        printIndent(out, indent); out.print("States: ");
        for (Node node : behaviour.nodePath())
        {
            if (node == null)
                out.print("-, ");
            else
            {
```

(continues on next page)

(continued from previous page)

```
            ProcessName processName = session.machineNodeName(behaviour.machine(),␣
↪node);
            if (processName == null)
                out.print("(unknown), ");
            else
                out.print(processName.toString()+", ");
        }
    }
    out.println();

    // Describe our own children recursively
    if (recurse) {
        for (Behaviour child : debugContext.behaviourChildren(behaviour))
            describeBehaviour(out, session, debugContext, child, indent + 2, true);
    }
}

/**
 * Prints a number of spaces to out.
 */
private static void printIndent(PrintStream out, int indent) {
    for (int i = 0; i < indent; ++i)
        out.print(' ');
}

}
```

5.2.3 Python

Execute using `python command_line.py`.

Download

```
import os
import platform
import sys

fdr_to_use = None
if sys.argv[1].startswith("--fdr="):
    fdr_to_use = sys.argv[1][len("--fdr="):]
    del sys.argv[1]
elif platform.system() == "Linux":
    for bin_dir in os.environ.get("PATH", "").split(":"):
        fdr4_binary = os.path.join(bin_dir, "fdr4")
        if os.path.exists(fdr4_binary):
            real_fdr4 = os.path.realpath(os.path.join(bin_dir, "fdr4"))
            fdr_to_use = os.path.join(os.path.basename(os.path.basename(real_fdr4)),
↪"lib")
            break
elif platform.system() == "Darwin":
    for app_dir in ["/Applications", os.path.join("~", "Applications")]:
        if os.path.exists(os.path.join(app_dir, "FDR4.app")):
            fdr_to_use = os.path.join(app_dir, "FDR4.app", "Contents", "Frameworks")
            break

```

(continues on next page)

(continued from previous page)

```
print fdr_to_use
sys.path.append(fdr_to_use)
import fdr

def main():
    fdr.library_init()
    return_code = real_main()
    fdr.library_exit()
    sys.exit(return_code)

def real_main():
    if not fdr.has_valid_license():
        print "Please run refines or FDR4 to obtain a valid license before running this."
        return 1

    print "Using FDR version %s" % fdr.version()

    if len(sys.argv) != 2:
        print "Expected exactly one argument."
        return 1

    file_name = sys.argv[1];
    print "Loading %s" % file_name

    session = fdr.Session()
    try:
        session.load_file(file_name)
    except fdr.FileLoadError, error:
        print "Could not load. Error: %s" % error
        return 1

    # Check each of the assertions
    for assertion in session.assertions():
        print "Checking: %s" % assertion
        try:
            assertion.execute(None)
            print "%s, found %s counterexamples" % \
                ("Passed" if assertion.passed() else "Failed",
                    len(assertion.counterexamples()))

            # Pretty print the counterexamples
            for counterexample in assertion.counterexamples():
                describe_counterexample(session, counterexample)
        except fdr.InputFileError, error:
            print "Could not compile: %s" % error
            return 1

    return 0

"""
Pretty prints the specified counterexample to out.
"""
def describe_counterexample(session, counterexample):
    # Firstly, just print a simple description of the counterexample
    if isinstance(counterexample, fdr.DeadlockCounterexample):
        t = "deadlock"
```

(continues on next page)

(continued from previous page)

```
    elif isinstance(counterexample, fdr.DeterminismCounterexample):
        t = "determinism"
    elif isinstance(counterexample, fdr.DivergenceCounterexample):
        t = "divergence"
    elif isinstance(counterexample, fdr.MinAcceptanceCounterexample):
        t = "minimal acceptance refusing {"
        for event in counterexample.min_acceptance():
            t += str(session.uncompile_event(event)) + ", "
        t += "}"
    elif isinstance(counterexample, fdr.TraceCounterexample):
        t = "trace with event "+str(session.uncompile_event(
                counterexample.error_event()))
    else:
        t = "unknown"

    print "Counterexample type: "+t
    print "Children:"

    # In order to print the children we use a DebugContext. This allows for
    # division of behaviours into their component behaviours, and also ensures
    # proper alignment amongst the child components.
    debug_context = fdr.DebugContext(counterexample, False)
    debug_context.initialise(None)

    for behaviour in debug_context.root_behaviours():
        describe_behaviour(session, debug_context, behaviour, 2, True)

"""
Prints a vaguely human readable description of the given behaviour to out.
"""
def describe_behaviour(session, debug_context, behaviour, indent, recurse):
    # Describe the behaviour type
    indent += 2;
    if isinstance(behaviour, fdr.ExplicitDivergenceBehaviour):
        print "%sbehaviour type: explicit divergence after trace" % (" "*indent)
    elif isinstance(behaviour, fdr.IrrelevantBehaviour):
        print "%sbehaviour type: irrelevant" % (" "*indent)
    elif isinstance(behaviour, fdr.LoopBehaviour):
        print "%sbehaviour type: loops after index %s" % (" "*indent, behaviour.loop_index())
    elif isinstance(behaviour, fdr.MinAcceptanceBehaviour):
        s = ""
        for event in behaviour.min_acceptance():
            s += str(session.uncompile_event(event)) + ", "
        print "%sbehaviour type: minimal acceptance refusing {%s}" % (" "*indent, s)
    elif isinstance(behaviour, fdr.SegmentedBehaviour):
        print "%sbehaviour type: Segmented behaviour consisting of:" % (" "*indent)
        # Describe the sections of this behaviour. Note that it is very
        # important that false is passed to the the descibe methods below
        # because segments themselves cannot be divded via the DebugContext.
        # That is, asking for behaviourChildren for a behaviour of a
        # SegmentedBehaviour is not allowed.
        for child in behaviour.prior_sections():
            describe_behaviour(session, debug_context, child, indent + 2, False)
        describe_behaviour(session, debug_context, behaviour.last(), indent + 2, False)
    elif isinstance(behaviour, fdr.TraceBehaviour):
```

(continues on next page)

(continued from previous page)

```
        print "%sbehaviour type: loops after index %s" % (" "*indent,
            session.uncompile_event(behaviour.error_event()))

    # Describe the trace of the behaviour
    t = ""
    for event in behaviour.trace():
        if event == fdr.INVALID_EVENT:
            t += "-, "
        else:
            t += str(session.uncompile_event(event)) + ", "
    print "%sTrace: %s" % (" "*indent, t)

    # Describe any named states of the behaviour
    t = ""
    for node in behaviour.node_path():
        if node == None:
            t += "-, "
        else:
            process_name = session.machine_node_name(behaviour.machine(), node)
            if process_name == None:
                t += "(unknown), "
            else:
                t += str(process_name)+", "
    print "%sStates: %s" % (" "*indent, t)

    # Describe our own children recursively
    if recurse:
        for child in debug_context.behaviour_children(behaviour):
            describe_behaviour(session, debug_context, child, indent + 2, recurse)

if __name__ == "__main__":
    main()
```

CHAPTER

SIX

OPTIMISING

When used properly, FDR is capable of verifying systems with billions (even trillions of states, if a cluster is being used) of explicit states. This section explains the different ways in which scripts can be optimised to take full advantage of FDR, whilst also explaining how to best configure and setup FDR to check a given problem.

This section assumes that you have an extremely large check that you wish to perform, but are unable to do so on your standard computer.

6.1 Overview

6.1.1 Identifying the Problem

The first step is to identify which in which part of FDR your performance problems are occurring. There are three main stages that FDR goes through when verifying any property (refinement, deadlock etc.):

1. *Evaluation* of the CSP_M processes. During Evaluation, FDR evaluates the CSP_M into a pure CSP process. If this stage is the bottleneck, the graphical user interface will show a task titled "Evaluating process" taking a long time to complete (see *The Task List* for details on where to locate the task list). See *Optimising Evaluation* for tips on how to optimise this stage.
2. *Compilation* of the pure CSP process. During compilation, FDR converts the CSP processes into labelled-transition systems (LTSs) and will be indicated by a task named "Constructing machines" in the task list. See *Optimising Compilation* for tips on how to optimise this stage.
3. *Checking* of the compiled LTS. During this stage FDR is actually verifying the property in question. This is indicated by a task named "Checking Refinement". See *Optimising Checking* for tips on how to optimise this stage.

6.1.2 Optimising Evaluation

If evaluation is the problem it generally indicates that the CSPm you have constructed is too complex for FDR to efficiently represent. Firstly, it is worth checking that all processes are finite state, since any infinite state process (such as the standard CSP processes `COUNT(n)`) will cause FDR to spin on this stage. The other common reason is using sets that are too large for channels or datatypes. For example, a channel such as `channel c : Int` means that FDR has to expore 2^{32} different branches whenever it sees a statement of the form `c?x`. Using small finite sets is critical to ensure that FDR can efficiently evaluate the model.

Note that results about infinite types can sometimes be established by small finite state checks by using the theory of *data independence*. See, for example, *Understanding Concurrent Systems* for further details.

6.1.3 Optimising Compilation

Optimising compilation is difficult since, generally, the causes of this are difficult to determine. One common cause is using too many events, so reducing the size of channel types as far as possible can help. Further, if too much renaming is be performed (particularly renaming a single event to many different copies of it), this can also cause performance to decrease.

Another way to improve performance is by using a carefully placed application of *`explicate`*: generally the best place to use it is in an argument `P` of a parallel composition, where `P` has a very complex definition, but actually has a relatively small state space (at most a million states or so). Also, it can be worth experimenting by disabling *`compiler.recursive_high_level`* as this can occasionally cause an increase in compilation time.

6.1.4 Optimising Checking

If the bottleneck is in the checking stage, there are essentially two different mitigations: either more computing resources can be directed at the problem, or the model can be optimised to attempt to reduce its size, or increase the speed, at which it can be verified. Often a combination of both of these is required. We review both of these techniques below, in turn.

Using more Computing Resources

There are several ways in which FDR can make use of more computing: a larger machine can be utilised, on-disk storage can be used, or a cluster of computers can be utilised. These are all reviewed below.

Since FDR is able to linearly scale to machines with many cores (we have tested with machines of up to 40 cores) on many problems, using a larger machine is the first step. Note that it is also possible to rent such machines from providers such as Amazon Web Services, or Google Compute Cloud for a small amount each hour.

If the problem is lack of memory, FDR is also able to make use of on-disk storage (although we would only recommend SSDs, particularly on large machine). We have been able to verify problems where the on-disk storage is a factor of 4 larger than the amount of RAM without a noticeable slowdown. In order for this to be effective, the machine needs to be configured correctly. In general, if the machine in question has multiple drives, we strongly recommend the use of no RAID (i.e. each drive is independently mounted by the operating system). Further, numerous small drives are better than a small number of large drives.

Once the system has been properly configured with disk storage, FDR can be configured be setting the option: *`refinement.storage.file.path`* to a comma separated list of paths. For example, passing the command line flag `--refinement-storage-file-path=/scratch/disk0/,/scratch/disk1/,/scratch/disk2/` to the command line version will cause FDR to store temporary data in the three different directories. By default, FDR also uses 90% of the free memory at the time the check starts to store data in-memory. This value can be changed (see *`refinement.storage.file.cache_size`*), but generally should not need to be.

The other solution is to use FDR on a cluster instead. For a wide variety of problems, FDR is capable of scaling linearly as the cluster size increases providing sufficient network bandwidth is available (we achieved linear scaling on a cluster of 64 computes connected by a full-bisection 10 gigabit network on Amazon's EC2 compute cloud). Full details on how to use the cluster version are in *Using a Cluster*.

Optimising the Model

Firstly, check to see if the semantic model that the check is being performed in is necessary. Checks in the failures-divergences model are slower than those in the failures model, which are in turn slower than those in the traces model. Further, the failures-divergences model is actually particularly slow and does not scale as well as the number of cores increases (which will reduce the benefit of more computing power). If it is possible to statically determine that divergence is impossible, or if divergence is not relevant to the property in question, then a potentially large performance boost can be obtained by moving to the failures or traces model. Clearly, care must be taken when choosing a different denotational model, since this clearly remove violations of the specification.

The most powerful technique that FDR has for optimising models is *compression*. Compression takes a LTS and returns an equivalent LTS (according to the relevant denotational model) that, hopefully has fewer states. Compression is a complex technique to utilise effectively, but is by far the most powerful method of reducing the number of states that have to be verified. Compression is described in *Compression*.

Another possibility for optimising models is to use *partial order reduction*. This attempts to remove redundant reoderings of events (for example, if a process can perform both `a` and `b` in any order, then partial-order reduction would only consider one ordering). Thus, this is similar to compression, but can be applied much more easily. Equally, partial-order reduction is unable to reduce some systems that compression can reduce. Since it is trivial to enable partial-order reduction (see the instructions at *partial order reduction*), it is often worth enabling it just to see if it helps.

Lastly, since CSP is *compositional*, it may sometimes be possible to decompose a check of a large system into a series of smaller checks. In particular, CSP is compositional in the sense that if $P \sqsubseteq Q$, then for any CSP context C, $C[P] \sqsubseteq C[Q]$. For example, suppose a system contained a complex model of a finite state buffer. Separately, a verification could be done that showed that the complex finite state buffer satisfies the standard CSP finite-state buffer specification. Then, the original system could be verified with the standard CSP finite-state buffer specification taking the place of the complex buffer. This should reduce the number of states that need to be verified.

6.2 Compression

Compression converts a labelled-transition system (LTS) into an equivalent LTS that, hopefully, is smaller and thus more efficient to use in a check. Generally, one of the many compression functions that FDR provides is applied to a component of a parallel composition. For example, *`normal`* is one commonly-used compression functions, as could, for instance be applied as `normal(Q) ||| R`. Compression will have little affect on a process that contains no hiding. Thus, it makes sense to compress a process that has just had a significant amount of hiding applied to it. For example, if `P \ A` is used at some point in a system and `A` contains a large number of the events `P` can perform, then compressing `P \ A` may significantly reduce size of the system that needs to be verified. Intuitively, compression *removes* the hidden events in such a way as to preserve sufficient behaviour, but not too much. This also implies that it is worth hiding as much as possible as early as possible (i.e. as far down the process tree as possible), since that gives compression more options to remove events.

Note it is not worth compressing a whole system. For example, when verifying if `Q` is deadlock-free, there is no point in verifying `normal(Q)` is deadlock-free instead, since FDR has to traverse the whole of `Q` to construct the normalised version anyway.

FDR provides many compression functions, as listed in *Compression Functions*, all of which are CSP_M functions of type `(Proc) -> Proc`. Generally, the most useful functions to apply are *`normal`*, *`wbisim`*, and the function `sbdia(P) = sbisim(diamond(P))` which combines *`sbisim`* and *`diamond`*. Further advice on how to choose a compression function is given in *Compression Functions*, but note that it is difficult to predict in advance which compression function will be most effective, and thus some experimentation will be required. The best method of assessing the effectiveness of compressions is to use the *machine structure viewer*, which can display details regarding the number of states and transitions that are saved by compression.

A more advanced usage of compression is *inductive compression*. In this, the system is constructed in stages, with each intermediate stage being compressed. For example, when modelling a token ring, only the events on the very left-most and very right-most processes must not be hidden. Therefore, the system can be constructed inductively by taking the existing system, adding one more node, hiding the events between the old system and the new node, and then compressing it. *Understanding Concurrent Systems* describes this in more detail. Further, the file *Inductive Compression* contains some utility functions, written by Roscoe, for constructing systems in such a way.

CHAPTER

SEVEN

IMPLEMENTATION NOTES

7.1 Semantic Models

7.1.1 Traces Model

Todo: Write

7.1.2 Failures Model

Todo: Write

7.1.3 Failures-Divergences Model

Todo: Write

7.2 Compilation

Compilation is the process of turning a tree of CSP operator applications, as produced by the evaluator, into concrete *state machines*, on which refinement checking can be performed.

We start by specifying the different types of machines that FDR can produce, before giving a high-level overview of how compilation proceeds.

7.2.1 Machine Types

Low-Level Machine

Todo: Write

Generalised Low-Level Machine

Todo: Write

High-Level Machine

Todo: Write

7.2.2 Compiling Processes

Todo: Write

7.3 Refinement Checking

In order to check if a process `Impl` refines (in a particular semantic model) a process `Spec`, FDR firstly converts `Impl` and `Spec` into labelled transition systems, as described in *Compilation*. FDR then normalises the specification machine, as described in *`normal`*. FDR then explores the states of these processes in breadth-first order, checking that the states are related according to the semantic model. As soon as FDR finds a counterexample it immediately reports it. Thus, thanks to the breadth-first order, any counterexample that FDR returns will be minimal in the length of the trace (including taus).

In order to understand exactly how FDR visits state pairs during a refinement check, consider the following script:

```
channel a, b, c

P1 = a -> P2
P2 = P3 |~| P4
P3 = b -> P2
P4 = a -> P1

Q1 = a -> Q2
Q2 = b -> Q1

assert Q1 [T= P1
```

FDR initialises the search by looking at the state pair `(Q1, P1)` (nb. state pairs consist of a specification state and an implementation state). In the traces model, FDR only has to check that the events (excluding tau) offered by `P1` are a subset of those offered by `Q1`, which is clearly the cases here. Hence, FDR now adds the successor state pairs for each event offered by `P1` to the search. In this case, there is only one successor state pair after `a`, `(Q2, P2)`.

When considering the state pair `(Q2, P2)` note that `P2` does not offer any visible events, and therefore the event check is trivially true. FDR now needs to add any new state pairs to the search. In this case, the two state pairs `(Q2, P3)` and `(Q2, P4)` are added to the search. Note that the specification part of the state pair is kept the same because the implementation performs a tau. FDR will now consider the new state pairs. In particular, when it considers `(Q2, P4)` FDR will find a counterexample since the events offered by `P4` are not a subset of those offered by `Q2`.

In order to reconstruct the trace that the invalid state pair was reached by, FDR stores a parent state pair of each state pair that it finds during the search. Thus, in the above example, FDR will set the parent of `(Q2, P4)` as `(Q2, P2)` and the parent of `(Q2, P2)` to `(Q1, P1)`. Thus, FDR can reconstruct the counterexample trace by finding events that transition between `(Q1, P1)` and `(Q2, P2)`, and `(Q2, P2)` and `(Q2, P4)`. This will yield the trace `<a, tau>`, in this case.

7.3.1 Counterexamples

A counterexample to a refinement assertion consists of a trace leading to a particular state pair that are not related according the to semantic model in use. Thus, if the traces model is in use, then this must mean that the implementation can perform an event that the specification cannot. If the failures model is in use, then either the above case applies, or the implementation can refuse a set of events that the specification cannot. If the failures-divergences model is in use, then this must mean that either one of the above cases applies, or the implementation is divergent but the specification is not.

Uniqueness

If FDR is asked to generate multiple counterexamples then this motivates the question of what FDR considers distinct counterexamples to be. In all cases, only a single counterexample will be generated from a single state pair. Thus, if a state pair is reachable via two different traces then only one counterexample will be generated, regardless of the number requested. Further, once FDR has found a counterexample for a given state pair, it does not look at successors of the state pair to see if they are counterexamples. Thus, for FDR to report two distinct counterexamples the counterexamples must be distinct state pairs, and the second state pair must be reachable via a path that does not include the first state pair. For example, the following has two counterexamples since the two STOP states are reachable via distinct paths:

```
P1 = STOP |~| P2
P2 = STOP

assert P1 :[deadlock free [F]]
```

However, the following only has one counterexample, since the second invalid state pair is only reachable via the first invalid state pair:

```
P1 = a -> P2 [] b -> STOP
P2 = a -> P1

Spec = a -> Spec

assert Spec [T= P1
```

Note that both `P1` and‘ `P2` violate the property, but `P2` is only reachable via `P1` and thus is not considered.

Non-Determinism

As noted above, thanks to the breadth-first ordering that FDR visits the state pairs in, FDR will always pick the counterexample that is reachable in the fewest events (either tau or a visible event). Whilst this might imply that the counterexample returned is chosen deterministically, this is not the case when FDR is being used in parallel mode (i.e. *refinement.bfs.workers* is greater than 1) for two reasons:

1. The predecessor state pair of each state pair is chosen by a race between the different processor cores on each ply. For example, if two different cores both discover the same state pair on the same ply, then the state pair that is marked as the new state pair's predecessor is chosen non-deterministically. This can result in a different trace being returned. The following script should exhibit this behaviour:

```
channel a, b, c

P = a -> b -> Q [] b -> a -> Q
Q = c -> STOP

R = a -> R [] b -> R [] c -> R

assert R [F= P
```

Note that there is precisely one state that violates the refinement, `Q`, but there are two different routes to `Q`, via either `<a, b>` or `<b, a>`. Hence, since the states `b -> Q` and `a -> Q` are discovered on the same ply of the breadth-first search the winner will be chosen non-deterministically (assuming that they are visited by different cores). This should result in FDR non-deterministically returning either the counterexample trace `<a, b, c>` or `<b, a, c>`.

2. On each ply of the search, there is essentially a race between the processor cores to find a counterexample: the first core that finds a counterexample wins. For example, consider the following refinement check:

```
channel a, b

P = a -> STOP |~| b -> STOP

assert STOP [T= P
```

There are two different counterexamples to this assertion: either the event `a` can be performed, or the event `b`. Since these counterexamples are both found on the second ply of the search (the first ply simply explores the tau transitions available from the starting state), FDR will pick the counterexample non-deterministically, providing the two different states are picked by different cores.

In general use, the fact that different counterexamples are produced should hopefully not be an issue. Unfortunately, there is no efficient way to make a parallel version of FDR deterministically return the same counterexample. If it is important that the same counterexample is returned each time then *refinement.bfs.workers* should be set to 1, thus disabling the parallel refinement checker. This will obviously result in a decrease in performance, but it does mean that the same counterexample will always be returned.

7.3.2 Implementation

Todo: Write

BTrees

Log-Space Merge Trees

7.4 Type Checking

Todo: Write

CHAPTER

EIGHT

RELEASE NOTES

8.1 4.2.7 (11/05/2020)

- Fixes an additional issue that meant FDR4 could not connect to the licensing servers.

8.2 4.2.6 (10/05/2020)

- Fixes an issue that meant FDR4 could not connect to the licensing servers.

8.3 4.2.5 (27/04/2020)

- Fixes a crash on recent Mac OS versions.
- Fixes a bug where `Inter` would return incorrect results.

8.4 4.2.4 (19/02/2019)

- Fixes an uncommon issue where FDR4 would spin when compiling processes that used union in a recursive argument.
- Fixes a crash when using failures-divergences determinism checking on processes that have no stable failures.
- Fixed a bug where FDR4 would hang when using pattern matching against infinite lists.
- Fixed using module instances together with maps.
- Fixed a rare issue where where FDR4 would ignore live variables, resulting in processes being incorrectly equated by the compiler.
- Fixed an issue where cspmprofiler would crash on scripts containing project.

8.5 4.2.3 (26/10/2017)

- Fixes an issue where FDR was unable to connect to the license server.
- Fixes a crash that occurred when compiling processes involving certain combinations of sequential composition and parallel.

- Ensures the typechecker rejects scripts where nametype declarations are over sets include datatype constructors. This would have previously resulted in a crash.
- Fixed a crash that could occur when CSPM stack tracing was enabled.

8.6 4.2.2 (09/10/2017)

- Fixes a bug in the compiler that would cause already compiled machines to be reused. This could cause some refinement checks to given incorrect results if they were run after other checks in the same session, but only when the state machines involved were extremely small.

8.7 4.2.1 (19/09/2017)

- Fixes a bug in `diamond` that would erroneously mark divergent processes as non-divergent.
- Add optional cspm stack tracing to record errors.

8.8 4.2.0 (20/12/2016)

This release includes a large number of internal changes that affect the way FDR is organised, and lay the groundwork for future extensions that are currently being worked on.

8.9 3.4.0 (09/03/2016)

- Added `refines --remote` which can automatically run *refines* on a remote server (but using a local file), connecting to it via SSH.
- Added the ability for *refines* to read its input from stdin.
- Added a licensing system that requires you to provide a name and email address before using FDR.
- Improved the presentation of loop counterexamples in the GUI.
- Enhanced the API to add:
  - A method *load_strings_as_file* that allows a file to be loaded from a set of files.
  - Methods *task_id* that return the task id for long running tasks, such as refinement checks. This can be used to provide better progress reporting.
- Fixed an issue where some recursive datatypes would erroneously fail to satisfy the `Eq` constraint.
- Fixed an error that could cause duplicate states to be created by `wbisim`.
- Fixed an error that could occur when evaluating polymorphic functions that used dot in a particular way.
- Fixed an error that could cause superfluous compressions to be performed.
- Fixed several crashes that could occur when debugging counterexamples in the GUI.

8.10 3.3.1 (17/06/2015)

- Fixed a problem that prevented FDR3 from being opened under Windows.
- Added support for a new machine-readable output format, `framed_json` (*refines --format*).
- Added a new method `reveal_taus_in_trace` to the `DebugContext` object in the API.

8.11 3.3.0 (15/06/2015)

- Greatly improved performance when evaluating CSPM. 3.3.0 can evaluate CSPM scripts in half the time that FDR 3.2.0 could.
- Added a new assertion, intended for unit-testing, to assert that a machine *has a trace*.
- Refinement checking performance has been further improved on large machines with 16 or more cores; checks can be between 10-20% faster.
- Added a new time-sampling *profiler for CSPM* that replaces the previous profiler.
- Removed popovers from various places in the GUI. They have been replaced by a combination of easier-to-use windows.
- Updated to QT5 which fixes various issues with the GUI.
- API: added support for dynamically created assertions.
- API: fixed the hash implementation on various objects.
- Added a new built-in process, *RUN*.
- Added a new built-in process, *DIV*.
- Added a new operator *Project* that is like *Hide*, but instead allows you to specify what events should be left unhidden, rather than what must be hidden.
- Added a new flag *refines --reveal-taus* to the command-line version that prints the visible event corresponding to any tau in a counterexample trace that resulted from a hiding.
- Banned the use of *Double Pattern* within prefix statements, due to the complexity of correctly and efficiently supporting it.
- Renamed the type constraint `Inputable` to *Complete*.
- Fixes several performance issues with very large renamings, and with calls to `CHAOS(X)` where `X` is very large.
- Fixed two issues with the low-level version of *Alphabetised Parallel*.
- Fixed a issue on Windows whereby the command line version would appear to finish early and would give the wrong return code.
- Fixed an issue with Probe where it would appear to hang.
- Fixes several issues where type-incorrect scripts would pass the typechecker.
- Fixed a crash during division of some *chase* counterexamples.
- Fixed a crash that could occur compiling complex processes using the recursive high-level strategy.

8.12 3.2.1 – 3.2.3 (06/01/2015)

- Fixed an issue that prevented graphs from being viewed under Linux and Mac OS.
- Fixed a performance problem with compiling certain large non-recursive processes.
- Fixed a issue that prevented type signatures that used datatypes from being used.

8.13 3.2.0 (30/01/2015)

- Added an beta Windows release, which is compatible with Windows 7 and above.
- Vastly improved partial-order reduction performance. This improves the performance by at least one order of magnitude. On certain examples, the performance improvements is two orders of magnitude.
- The performance of `dbisim` and `wbisim` has been improved by a factor of 2-4, depending on the example.
- Added a *machine structure viewer* to the user interface that shows the structure of a process.
- Added a *communication graph viewer* that allows the communication graph of a process to be visualised.
- Added a new version of prioritise, `prioritisepo` that takes a partial order to prioritise the events over.
- Added a function `mapDelete` that deletes a key from a map.
- Changed the *machine-readable command-line interface* to also serialise the results of evaluating print statements *Print Statements*.

Bug Fixes:

- Fixed a bug that prevented the C++ API being used under Mac OS X.
- Improve the compiler performance on certain examples with nested uses of the sequential composition operator.
- Fix some problems with the deduction of which MPI version is being used.
- Fixed a bug where the cluster version would never terminate if counterexamples were not being tracked.
- Fixed parsing of string literals.
- Fixed pretty-printing of map values.
- Fixed a bug where type signatures that were too liberal were erroneously accepted.
- Fixed a bug with pattern matching of complex nested datatypes.
- Fixed a bug that occurred when parametrised modules were instantiated with the wrong number of arguments.
- Fixed a bug that occurred when subtypes were instantiated with the wrong number of arguments.
- Fixed type-checking of parameterless modules in various obscure cases.
- Fixed an issue that caused terms of the form `chase(chase(...(chase(P)...)` to appear when probing `chase(P)`.

8.14 3.1.0 (11/08/2014)

Major New Features:

- Added a cluster based refinement-checking algorithm that is able to scale linearly to clusters of at least 1024 cores, providing a suitable interconnect is available. See *Cluster Refinement Checking* for further details.

- An *API* for C++, Java, and Python has been made available, allowing FDR to be easily embedded into external tools.
- Added a function `failure_watchdog` that constructs the failures watchdog for the given specification process.
- Added a function `trace_watchdog` that constructs the trace watchdog for the given specification process.
- Improved the output of the type checker so that it provides more useufl information regarding why a program is type incorrect.

Performance Improvements:

- Reduced the runtime for FDR across a range of problems by between 25 and 50%.
- Improved the performance of FDR on machines with 8 or more cores by between 10 and 20 percent, depending on the problem and the machine. After this change we have observed FDR3 scaling linearly to 40 cores.
- For some problems, reduced the memory consumption by 25%.
- For deadlock freedom and divergence freedom assertions, reduced the runtime by anywhere between 10 and 80%.
- Improved the performance of `explicate` by a factor of 3.
- Improved the performance and memory usage of the on-disk storage engine.

Miscellaneous Features:

- Added experimental support for partial order reduction which can automatically reduce the state space size on some problems. See *Partial Order Reduction* for more details.
- Added an `refines --archive` that archives all CSP files required to load a particular CSP file into a single, easy to transfer file.
- Improved counterexample division so more counterexamples can be divided.
- Added an option `refinement.track_history` that allows history tracking to be disabled. This will mean that FDR consumes less memory at the cost of not being able to reconstruct counterexamples if an error is found.
- Modified the option `refinement.storage.file.path` to allow a comma-separated list of paths to be specified. FDR evenly distributes writes over the paths that have sufficient space available.

Bug Fixes:

- Fixed an issue that prevented graphs containing tick from being rendered under Linux.
- Fixed a crash that would randomly occur when an assertion was started.
- Fixed a bug that caused a crash when a compressed process was probed when debugging a refinement check.
- Fixed performance issues that could arise when given extremely large CSP files with over 100,000 lines of code.
- Fixed a memory usage issue with very long running checks.
- Fixed several problems with type-checking parameterised modules.
- Fixed an issue that would cause FDR to go into an infinite loop when compiling some processes with lots of singleton taus.

8.15 3.0.0 (09/12/2013)

FDR3 is a complete rewrite of FDR2 that includes a number of exciting new features.

- A brand new refinement checking engine that:

  - Is multi-threaded, allowing it to make full use of multiple cores.
  - Has been heavily optimised, meaning that single core performance tends to be around double that of FDR2.
  - Includes alternative data structures for storage of states.
- Has a fully integrated type-checker, that permits the vast majority of *reasonable* CSP programs, whilst keeping errors readable.
- A compiler that optimises the representation of some CSP processes (compared to FDR2) and also compiles machines in parallel.
- A completely redesigned GUI that includes:
  - A redesigned and more powerful *Debug Viewer* that, in particular, explicitly indicates how events synchronise, even through renaming.
  - A fully integrated version of *Probe*.
  - A full *interactive prompt*, allowing for easy experimentation with $\mathrm{CSP_M}$.
- An enhanced version of $\mathrm{CSP_M}$ with:
  - Support for a new efficient key-value *Map datatype*.
  - Support for explicit *type annotations*.

Compared to FDR2 there are the following differences:

- The compression function `model_compress` has been removed.
- The batch mode has been removed and replaced with a new output format (see *Machine-Readable Formats*).
- Only the Traces, Failures and Failures-Divergences models are supported. Support for the other models that FDR2 supported will return in a subsequent release.

Compared to 3.0-beta-7, the following changes have been made:

- Enhanced the refinement checker to terminate as soon as a counterexample as found.
- Fixed an issue where the debug viewer could sometimes elide rows that were important.
- Fixed an issue that caused machines compressed using `dbisim` and `wbisim` to have more transitions than necessary.

8.15.1 3.0-beta-7 (15/11/2013)

- Created RPM and Apt repositories to allow for easier installation on Linux. We strongly recommend that all existing users install FDR3 in this way, if possible. Instructions for doing so can be found on the FDR3 home page.
- Improved the performance of the on-disk storage option during refinement checks. Several new options have been added to control it, including: `refinement.storage.file.path`, `refinement.storage.file.cache_size`, `refinement.storage.file.checksum`, and `refinement.storage.file.compressor`.
- Modified the behaviour of `wbisim` to be full weak bisimulation and renamed the old version of `wbisim` to `dbisim`, which computes a delay bisimulation. `wbisim` can compress more than `dbisim`, but in the worst case takes twice as long to compute.
- Adapted the compiler to elide some unnecessary taus, thus reducing the state space size of some processes. This change is to match a feature in FDR2.
- Improved the performance of `sbisim`.

- Improved the performance of checks that use *`chase`* and *`prioritise`*.
- Parallelised divergence checking. This allows multiple threads to proceed in parallel, checking for divergences that start from different nodes. This will does not parallelise divergence checks that start from a single node.
- Improved the usability and stability of the *debug viewer*.
- Added the *`cd`* command to the session window that changes the current directory.
- Fixed several crashes in the graphical user interface.
- Fixed a few performance issues and one incorrect error message that sometimes appeared when evaluating complex CSPm scripts that make use of large recursive datatypes.
- Fixed ghosting that could occur in the debug viewer.

8.15.2 3.0-beta-6 (18/10/2013)

- Many bugs have been fixed in the graphical user interface, including:
  - Fixed several issues that caused FDR3 to steal focus on Ubuntu.
  - Refusal sets are now correctly calculated.
  - Fixed an alignment issue with behaviours in the debug viewer.
  - Fixed a bug that caused the user interface to lockup if Run All was clicked whilst another assertion was running.
  - Removed a warning that QT emitted on startup.
  - Fixed an issue that prevented the user interface from shutting down on Ctrl+C.
  - Fixed a crash when unchecking `Show Taus`.
- Added support for opening files either by dragging them to the application icon (Mac OS X only), or by opening them using the operating system's file browser.
- Improved the performance of refinement checks on large machines with multiple processor sockets.
- Optimised the *`diamond`* compression.

8.15.3 3.0-beta-5 (12/09/2013)

- Many improvements to the *Debug Viewer*, including:
  - All behaviours are now always correctly aligned in the debug viewer (i.e. it is now always the case that events in the same column are synchronised).
  - Improved the presentation of divergence and deadlock counterexamples.
  - Names of named compressed processes, such as `normal(P)`, are now displayed.
  - Added an option to hide taus in the trace.
  - Added the ability to highlight a row and column.
  - Added a new Transition Popover that appears when hovering over an event in the debug viewer. If the event is a tau this will display the hidden event that was performed. In all cases it will display what leaf machines synchronise to perform the event.
  - When zoomed in, the machine name for a given row is now always visible, even after scrolling.
  - Fixed a crash that could occur when dividing loop counterexamples.

- Added reporting of transition rates during refinement checks.
- Improved the presentation of the speed and memory graphs that appear in the refinement status popover.
- Added machine-readable output to the command-line tool; see *Machine-Readable Formats* for further details. This replaces FDR2's batch mode.
- Added two new commands, `counterexamples command` that pretty-prints a textual representation of the counterexamples to the given assertion to the prompt, and `statistics command` that prints statistics about a completed or running refinement check to the prompt.
- Improved the performance of strong bisimulation, typically resulting in a fourfold speedup.
- Fixed parsing of expressions that mix hiding and parallel operators (such as `X \ Y ||| Z`).
- Fixed an issue that prevented the file-backed storage from allocating a file that could appear on extremely large checks.
- Fixed a crash caused by cancelling a check during evaluation.
- Fixed a hang caused by a cancelling a check during compilation.

8.15.4 3.0-beta-4 (09/08/2013)

- Add support for checking determinism using the failures model.
- Added support for simple *profiling* of CSPM scripts.
- Added an option to disable runtime bounds checks for performance reasons (see `cspm.runtime_range_checks`).
- Added some parallelisation to the CSPM evaluator, resulting in a speed up when complex CSPM expressions are evaluated as part of checking an assertion.
- Allow refinement checks to use on-disk temporary storage (see `refinement.storage.file.path`).
- Renamed the `refinement.compressor` option to `refinement.storage.compressor`.
- Added a *Node Inspector*.
- Added an efficient representation of key-value maps (see *Map Functions*) to CSPM.
- Allow more resizing of the main window.
- Added an option to close all windows whenever a load or reload command is executed (see `gui.close_windows_on_load`).
- Allow multi counterexamples to be viewed in the *Debug Viewer* by setting the option `refinement.desired_counterexample_count`.
- Added the ability to view refusals when viewing minimal acceptance information in the *Debug Viewer*.
- Added an indication to windows that were created using previous versions of a file (i.e. before a reload/load).
- Changed the behaviour of print statements to allow them to be evaluated on demand, rather than always evaluating them when a file is loaded.
- Enhanced the graph viewer to allow layout to be cancelled.

8.15.5 3.0-beta-3 (17/7/2013)

- Added charts for memory and checking speed to the refinement status popover.
- Enhanced *type annotations* to allow types defined inside modules to be used.

- Added support for nested *parametrised modules*. Note: instance declarations must now occur lexically after the module that is being instantiated.
- Added support for type annotations to mention datatypes defined in modules.
- Allow refinement checks etc. to be cancelled.
- Improved performance of processes that contain large numbers of transitions.
- Improved some of the error messages emitted by the compiler.
- Fixed parsing of minuses immediately followed by a newline.
- Fixed a bug that could potentially caused a compiled machine to be reused even if it had not been compiled in the correct semantic model.
- Prevent a crash that could occur if load/reload was executed whilst a refinement check was running.

8.15.6 3.0-beta-2 (15/6/2013)

- Made the compiler emit an error if `prioritise` is applied to a process in such a way that makes the result semantically invalid.
- Add support for *print statements*.
- Reduce memory consumption during refinement checks by around 1/3.
- Added an auto-updater for Mac OS X and alert Linux users to new releases.
- Fixed a crash that could occur when loop counterexamples were divided.
- Enhanced the *process graph viewer* to allow behaviours from the debug viewer to be viewed.
- Fixed a bug that prevented the Mac OS X version from being opened.
- Fixed some problems with parsing files with includes inside modules.
- Expanded `Lambda` to allow multiple arguments to be specified.
- Increased the resilience of the parser when dealing with malformed includes.

8.15.7 3.0-beta-1 (23/5/2013)

- Add a rewritten compiler (the component which produces LTSs from processes). This should result in reduced memory usage during compilation, as well as reduced compilation time. Further, it fixes a number of bugs that prevented various processes from being compiled.
- Allow multiple assertions to be compiled simultaneously in the GUI.
- Added more feedback in the GUI on how compilation is proceeding.
- Add support for *parametrised modules*.
- Fixed the definition of `WAIT` in timed sections.
- Fixed the evaluation of empty replicated operators in timed sections.

8.15.8 3.0-alpha-6 (02/4/2013)

- Added support for manual *type annotations*, to aid with code documentation and to make the type-checker give more accurate errors.

- Fixed a crash that could occur when dividing repeat behaviours through compressions.

8.15.9 3.0-alpha-5 (26/3/2013)

- Add more information to the machine popover to allow long traces to be easily viewed.
- Enhanced the range of programs that can be successfully type-checked.
- Allow the previous and next word keyboard shortcuts to be used from the GUI terminal.
- Fixed several issues that would mean `wbisim` return processes that were not semantically equivalent to the original process.
- Fixed an issue with timed CSP that would result in incorrect processes being generated.
- Fixed the low-level implementation of `Alphabetised Parallel` and `prioritise`.
- Fixed a crash that could occur when a behaviour involving `chase` was divided.

8.15.10 3.0-alpha-4 (15/3/2013)

- Fixed a bug that caused high-level machines with compressed process arguments to have incorrect minimal acceptances. This may have caused some assertions to erroneously pass.
- Add support for tock-CSP via *timed sections*, `Synchronising External Choice` and `Synchronising Interrupt`.
- Enhanced the refinement status popover to include storage statistics, in addition to details on the ply. Also fix a bug where the per node storage estimate was under-reported.
- Added support for different compression algorithms which can be used during the refinement checking stage, including LZ4HC and zlib (see `refinement.storage.compressor`).
- Enhanced the refinement engine to reconstruct counterexamples in parallel.
- Added support for the `chase`, `deter`, `lazyenumerate`, `prioritise` and `wbisim` compressions.
- Fixed the low-level implementation of `Rename` and `Linked Parallel`, which may have caused incorrect transitions to be reported in Probe under certain circumstances.
- Added support for determinism checking in the failures-divergences model.

8.15.11 3.0-alpha-3 (19/2/2013)

- Added basic support for the failures-divergences model, including divergence freedom checks and failures-divergences refinement checks. Note that the current implementation has not been optimised to take advantage of multiple cores.
- Added support for `ranged set` and `ranged list` comprehensions, such as `<1.. | p == 1>` and `{1..2 | p == 1}`. Both finite and infinite variants are supported for both the lists and the sets.
- Implemented `diamond` compression.
- Implemented `tau_loop_factor` compression.
- Added a popover to display performance statistics about refinement checks.
- Fixed loading of files where the path includes ~ in FDR3.
- Improved tab completion of file names in the prompt.

- Fixes FDR3 running on older Mac's without `POPCNT`.
- Fixed the use of the forward delete key, and other deletion shortcuts, at the prompt.
- Fixed an issue where the node highlighting in *Probe* was not updated after navigating away.
- Fixed a crash that occurred when launching FDR3 with too many arguments.
- Correctly return a trace error if an assertion can fail because of both an acceptance and a trace error.
- Allowed the semantic model that a process is compiled in to be specified when using the `graph command`.

8.15.12 3.0-alpha-2 (11/1/2013)

- Improved the pretty printing of processes in *Probe*.
- Improved the performance of *Probe*.
- Added a manual.
- Allowed `extensions` and `productions` to be type-checked.
- Added a new primitive datatype: `Char` that represents a character, along with associated character and string literals.
- Added two new functions: `error` and `show` that display a given string as an error and pretty print a value, respectively.
- Restricted the use of prefixing to make it compatible with FDR2.
- Fixed several minor GUI issues.
- Fixed a type-checker issue that allowed incorrect programs that used `Extendable` to pass.
- Fixed evaluation of replicated link parallel of an empty sequence.
- Fixed evaluation of prefixing where `?` occurred before `$`.

8.15.13 3.0-alpha-1 (21/12/2012)

The initial alpha release.

CHAPTER

NINE

EXAMPLE FILES

9.1 FDR4 Introduction

Download

```
-- Introducing FDR4.0
-- Bill Roscoe, November 2013

-- A file to illustrate the functionality of FDR4.0.

-- Note that this file is necessarily basic and does not stretch the
-- capabilities of the tool.

-- To run FDR4 with this file just type "fdr4 intro.csp" in the directory
-- containing intro.csp, assuming that fdr4 is in your $PATH or has been aliased
-- to run the tool.

-- Alternatively run FDR4 and enter the command ":load intro.csp".

-- You will see that all the assertions included in this file appear on the RHS
-- of the window as prompts. This allows you to run them.

-- This file contains some examples based on playing a game of tennis between A
-- and B.

channel pointA, pointB, gameA, gameB

Scorepairs = {(x,y) | x  <- {0,15,30,40}, y <- {0,15,30,40}, (x,y) != (40,40)}

datatype scores = NUM.Scorepairs | Deuce | AdvantageA | AdvantageB

Game(p) = pointA -> IncA(p)
        [] pointB -> IncB(p)

IncA(AdvantageA) = gameA -> Game(NUM.(0,0))
IncA(NUM.(40,_)) = gameA -> Game(NUM.(0,0))
IncA(AdvantageB) = Game(Deuce)
IncA(Deuce) = Game(AdvantageA)
IncA(NUM.(30,40)) = Game(Deuce)
IncA(NUM.(x,y)) =  Game(NUM.(next(x),y))
IncB(AdvantageB) = gameB -> Game(NUM.(0,0))
IncB(NUM.(_,40)) = gameB -> Game(NUM.(0,0))
IncB(AdvantageA) = Game(Deuce)
IncB(Deuce) = Game(AdvantageB)
```

(continues on next page)

(continued from previous page)

```
IncB(NUM.(40,30)) = Game(Deuce)
IncB(NUM.(x,y)) = Game(NUM.(x,next(y)))
-- If you uncomment the following line it will introduce a type error to
-- illustrate the typechecker.
-- IncB((x,y)) = Game(NUM.(next(x),y))

next(0) = 15
next(15) = 30
next(30) = 40

-- Note that you can check on non-process functions you have written. Try typing
-- next(15) at the command prompt of FDR4.

-- Game(NUM.(0,0)) thus represents a game which records when A and B win
-- successive games, we can abbreviate it as

Scorer = Game(NUM.(0,0))

-- Type ":probe Scorer" to animate this process.
-- Type ":graph Scorer" to show the transition system of this process

-- We can compare this process with some others:

assert Scorer [T= STOP
assert Scorer [F= Scorer
assert STOP [T= Scorer

-- The results of all these are all obvious.

-- Also, compare the states of this process

assert Scorer [T= Game(NUM.(15,0))
assert Game(NUM.(30,30)) [FD=  Game(Deuce)

-- The second of these gives a result you might not expect: can you explain why?
-- (Answer below....)

-- For the checks that fail, you can run the debugger, which illustrates why the
-- given implementation (right-hand side) of the check can behave in a way that
-- the specification (LHS) cannot.  Because the examples so far are all
-- sequential processes, you cannot subdivide the implementation behaviours into
-- sub-behaviours within the debugger.

-- One way of imagining the above process is as a scorer (hence the name) that
-- keeps track of the results of the points that A and B score.  We could put a
-- choice mechanism in parallel: the most obvious picks the winner of each point
-- nondeterministically:

ND = pointA -> ND |~| pointB -> ND

-- We can imagine one where B gets at least one point every time A gets one:

Bgood = pointA -> pointB -> Bgood |~| pointB -> Bgood

-- and one where B gets two points for every two that A get, so allowing A to
-- get two consecutive points:

```

(continues on next page)

(continued from previous page)

```
Bg = pointA -> Bg1 |~| pointB -> Bg

Bg1 = pointA -> pointB ->  Bg1 |~| pointB -> Bg

assert Bg [FD= Bgood
assert Bgood [FD= Bg

-- We might ask what effect these choice mechanisms have on our game of tennis:
-- do you think that B can win a game in these two cases?

BgoodS = Bgood [|{pointA,pointB}|] Scorer
BgS = Bg [|{pointA,pointB}|] Scorer

assert STOP [T= BgoodS \diff(Events,{gameA})
assert STOP [T= BgS \diff(Events,{gameA})

-- You will find that A can in the second case, and in fact can win the very
-- first game.  You can now see how the debugger explains the behaviours inside
-- hiding and of different parallel components.

-- Do you think that in this case A can ever get two games ahead?  In order to
-- avoid an infinite-state specification, the following one actually says that A
-- can't get two games ahead when it has never been as many as 6 games behind:

Level = gameA -> Awinning(1)
        [] gameB -> Bwinning(1)

Awinning(1) = gameB -> Level -- A not permitted to win here

Bwinning(6) = gameA -> Bwinning(6) [] gameB -> Bwinning(6)
Bwinning(1) = gameA -> Level [] gameB -> Bwinning(2)
Bwinning(n) = gameA -> Bwinning(n-1) [] gameB -> Bwinning(n+1)

assert Level [T= BgS \{pointA,pointB}

-- Exercise for the interested: see how this result is affected by changing Bg
-- to become yet more liberal. Try Bgn(n) as n copies of Bgood in ||| parallel.

-- Games of tennis can of course go on for ever, as is illustrated by the check

assert BgS\{pointA,pointB} :[divergence-free]

-- Notice that here, for the infinite behaviour that is a divergence, the
-- debugger shows you a loop.

-- Finally, the answer to the question above about the similarity of
-- Game(NUM.(30,30)) and Game(Deuce).

-- Intuitively these processes represent different states in the game: notice
-- that 4 points have occurred in the first and at least 6 in the second. But
-- actually the meaning (semantics) of a state only depend on behaviour going
-- forward, and both 30-all and deuce are scores from which A or B win just when
-- they get two points ahead.  So these states are, in our formulation,
-- equivalent processes.

-- FDR has compression functions that try to cut the number of states of
-- processes: read the books for why this is a good idea. Perhaps the simplest
```

(continues on next page)

(continued from previous page)

```
-- compression is strong bisimulation, and you can see the effect of this by
-- comparing the graphs of Scorer and

transparent sbisim, wbisim, diamond

BScorer = sbisim(Scorer)

-- Note that FDR automatically applies bisimulation in various places.

-- To see how effective compressions can sometimes be, but that
-- sometimes one compression is better than another compare

NDS = (ND [|{pointA,pointB}|] Scorer)\{pointA,pointB}

wbNDS = wbisim(NDS)
sbNDS = sbisim(NDS)
nNDS = sbisim(diamond(NDS))
```

9.2 Dining Philosophers

Download

```
-- The five dining philosophers for FDR

-- Bill Roscoe

-- The most standard example of them all.  We can determine how many
-- (with the conventional number being 5):

N = 6

PHILNAMES= {0..N-1}
FORKNAMES = {0..N-1}

channel thinks, sits, eats, getsup:PHILNAMES
channel picks, putsdown:PHILNAMES.FORKNAMES

-- A philosopher thinks, sits down, picks up two forks, eats, puts down forks
-- and gets up, in an unending cycle.

PHIL(i) = thinks.i -> sits!i -> picks!i!i -> picks!i!((i+1)%N) ->
          eats!i -> putsdown!i!((i+1)%N) -> putsdown!i!i -> getsup!i -> PHIL(i)

-- Of course the only events relevant to deadlock are the picks and putsdown
-- ones.  Try the alternative "stripped down" definition

PHILs(i) =  picks!i!i -> picks!i!((i+1)%N) ->
           putsdown!i!((i+1)%N) -> putsdown!i!i -> PHILs(i)



-- Its alphabet is

AlphaP(i) = {thinks.i, sits.i,picks.i.i,picks.i.(i+1)%N,eats.i,putsdown.i.i,
```

(continues on next page)

(continued from previous page)

```
              putsdown.i.(i+1)%N,getsup.i}

-- A fork can only be picked up by one neighbour at once!

FORK(i) = picks!i!i -> putsdown!i!i -> FORK(i)
          []  picks!((i-1)%N)!i -> putsdown!((i-1)%N)!i -> FORK(i)

AlphaF(i) = {picks.i.i, picks.(i-1)%N.i, putsdown.i.i, putsdown.(i-1)%N.i}

-- We can build the system up in several ways, but certainly
-- have to use some form of parallel that allows us to
-- build a network parameterized by N.  The following uses
-- a composition of N philosopher/fork pairs, each individually
-- a parallel composition.

SYSTEM = || i:PHILNAMES@[union(AlphaP(i),AlphaF(i))]
                        (PHIL(i)[AlphaP(i)|| AlphaF(i)] FORK(i))

-- or stripped down

SYSTEMs = || i:PHILNAMES@[union(AlphaP(i),AlphaF(i))]
                        (PHILs(i)[AlphaP(i)|| AlphaF(i)] FORK(i))


-- As an alternative (see Section 2.3) we can create separate
-- collections of the philosophers and forks, each composed
-- using interleaving ||| since there is no communication inside
-- these groups.

PHILS = ||| i:PHILNAMES@ PHIL(i)
FORKS = ||| i:FORKNAMES@ FORK(i)

SYSTEM' = PHILS[|{|picks, putsdown|}|]FORKS

-- The potential for deadlock is illustrated by

assert SYSTEM :[deadlock free [F]]

-- or equivalently in the stripped down
assert SYSTEMs :[deadlock free [F]]

-- which will find the same deadlock a lot faster.

-- There are several well-known solutions to the problem.  One involves a
-- butler who must co-operate on the sitting down and getting up events,
-- and always ensures that no more than four of the five
-- philosophers are seated.

BUTLER(j) = j>0 & getsup?i -> BUTLER(j-1)
            []j<N-1 & sits?i -> BUTLER(j+1)

BSYSTEM = SYSTEM [|{|sits, getsup|}|] BUTLER(0)

assert BSYSTEM :[deadlock free [F]]

-- We would have to reduce the amount of stripping down for this,
-- since it makes the sits and getsup events useful...try this.
```

(continues on next page)

(continued from previous page)

```

-- A second solution involves replacing one of the above right-handed (say)
-- philosophers by a left-handed one:

LPHIL(i)= thinks.i -> sits.i -> picks.i.((i+1)%N) -> picks.i.i ->
          eats.i -> putsdown.i.((i+1)%N) -> putsdown.i.i -> getsup.i -> LPHIL(i)

ASPHILS = ||| i:PHILNAMES @ if i==0 then LPHIL(i) else PHIL(i)

ASSYSTEM = ASPHILS[|{|picks, putsdown|}|]FORKS

-- This asymmetric system is deadlock free, as can be proved using Check.

assert ASSYSTEM :[deadlock free [F]]

-- If you want to run a lot of dining philosophers, the best results will
-- probably be obtained by removing the events irrelevant to ASSYSTEM
-- (leaving only picks and putsdown) in:
LPHILs(i)=  picks.i.((i+1)%N) -> picks.i.i ->
           putsdown.i.((i+1)%N) -> putsdown.i.i -> LPHILs(i)

ASPHILSs = ||| i:PHILNAMES @ if i==0 then LPHILs(i) else PHILs(i)

ASSYSTEMs = ASPHILSs[|{|picks, putsdown|}|]FORKS

assert ASSYSTEMs :[deadlock free [F]]

-- Setting N=10 will show the spectacular difference in running the
-- stripped down version.  Try to undertand why there is such an
-- enormous difference.

-- Compare the stripped down versions with the idea of "Leaf Compression"
-- discussed in Chapter 8.
```

9.3 Inductive Compression

Download

```
-- compression09.csp

-- This DRAFT file supports various semi-automated compression techniques over
-- CSP networks for use with FDR.

-- It it is designed to accompany the author's forthcoming book
-- "Understanding Concurrency"
-- and is an updated version of the 1997 file "compresion.csp" that
-- accompanied "Theory and Practice of Concurrency".

-- Bill Roscoe

-- We assume that networks are presented to us as
-- structures comprising process/alphabet pairs arranged in list
-- arrangements,

```

(continues on next page)

(continued from previous page)

```
-- or (09) as members of the structured datatype

datatype PStruct = PSLeaf.(Proc,Set(Events)) | PSNode.Seq(PStruct)

-- This can only be used with FDR 2.91 and up where processes (Proc) are allowed
-- as parts of user-defined types.

-- We may (subject to alterations to FDR) be able to support more complex
-- structured types over processes.

-- The alphabet of any such list is the union of the alphabets of
-- the component processes:

alphabet(ps) = Union(set(<A | (P,A) <- ps>))

-- The vocabulary of a list is the set of events that are synchronised
-- between at least two members:

vocabulary(ps) = if #ps<2 then {} else
                 let A = snd(head(ps))
             V = vocabulary(tail(ps))
             A' = alphabet(tail(ps))
         within
             union(V,inter(A,A'))

-- The following is a function
-- that composes a process/alphabet list without any
-- compression:

ListPar(ps) = let N=#ps within
              || i:{0..N-1} @ [snd(cnth(i,ps))] fst(cnth(i,ps))

-- The most elementary transformation we can do on a network is to
-- hide all events in individual processes that are neither relevant to
-- the specification nor are required for higher synchronisation.
-- The following function takes as its (curried) arguments a compression
-- function to apply at the leaves, a process/alphabet list to compose
-- in parallel and a set of events which it is desired to hide (either
-- because they are genuinely internal events or irrelevant to the spec).
-- It hides as much as it can in the processes, but does not combine them

CompressLeaves(compress)(ps)(X) = let V = vocabulary(ps)
                                    N = #ps
                    H = diff(X,V)
                within
          <(compress(P\inter(A,H)),diff(A,H)) | (P,A) <- ps>

-- The following uses this to produce a combined process

LeafCompress(compress)(ps)(X) = ListPar(CompressLeaves(compress)(ps)(X))\X

-- It is often advantageous to be able to apply lazy or mixed abstraction
-- operators in the same sort of way as the above does for hiding.  The
-- following are two functions that generalize the above: they take a
-- pair of event-sets (X,S): X is the set we want to abstract and S is
-- the set of signal events (which need not be a subset of X).  The
-- result is that inter(X,S) is hidden and diff(X,S) is lazily
```

(continues on next page)

(continued from previous page)

```
-- abstracted.   Note that you can get the effect of pure hiding (eager
-- abstraction by setting S=Events) and pure lazy abstraction by setting
-- S={}.  Note also, however, that if you are trying to lazily abstract
-- a network with some natural hiding in it, that all these hidden events
-- should be treated as signals.

LeafMixedAbs(compress)(ps)(X,S)  =
                           let V = vocabulary(ps)
                               N = #ps
                               D = diff(X,S)
                               H'= diff(X,V)
                           within
          <(compress((P[|inter(A,D)|]
            compress(CHAOS(inter(A,D))))\inter(A,H')),diff(A,H'))
                 | (P,A) <- ps>

-- The substantive function is then:

MixedAbsLeafCompress(compress)(ps)(X,S)  =
       ListPar(LeafMixedAbs(compress)(ps)(X,S))\X


-- The next transformation builds up a list network in the order defined
-- in the (reverse of) the list, applying a specified compression function
-- to each partially constructed unit.

InductiveCompress(compress)(ps)(X)  =
        compress(IComp(compress)(CompressLeaves(compress)(ps)(X))(X))

IComp(compress)(ps)(X)  = let  p = head(ps)
                   P  = fst(p)
                   A  = snd(p)
                   A' = alphabet(ps')
                   ps' = tail(ps)
              within
                           if #ps == 1 then P\X
               else
               let Q = IComp(compress)(ps')(diff(X,A))
                           within
                    (P[A||A']compress(Q))\inter(X,A)

InductiveMixedAbs(compress)(ps)(X,S)  =
        compress(IComp(compress)(LeafMixedAbs(compress)(ps)(X,S))(X))

-- Sometimes compressed subnetworks grow to big to make the above
-- function conveniently applicable.  The following function allows you
-- to compress each of a list-of-lists of processes, and then
-- combine them all without trying to compress any further.

StructuredCompress(compress)(pss)(X)  =
                         let N = #pss
             as = <alphabet(ps) | ps <- pss>
             ss = <Union({inter(cnth(i,as),cnth(j,as)) |
                j <- {0..N-1}, j!=i}) | i <- <0..N-1>>
             within
             (ListPar(<(compress(
                 InductiveCompress(compress)(cnth(i,
```

(continues on next page)

(continued from previous page)

```
                        pss))(diff(X,cnth(i,ss)))
                           \(diff(X,cnth(i,ss)))),
                      cnth(i,as)) | i <- <0..N-1>>))\X

-- The analogue of ListPar

StructuredPar(pss) =  ListPar(<(ListPar(ps),alphabet(ps)) | ps <- pss>)

-- and the mixed abstraction analogue:

StructuredMixedAbs(compress)(pss)(X,S) =
                           let N = #pss
             as = <alphabet(ps) | ps <- pss>
             ss = <Union({inter(cnth(i,as),cnth(j,as)) |
                j <- {0..N-1}, j!=i}) | i <- <0..N-1>>
             within
             (ListPar(<(compress(
                 InductiveMixedAbs(compress)(cnth(i,
                      pss))(diff(X,cnth(i,ss)),S)
                         \(diff(X,cnth(i,ss)))),
                    cnth(i,as)) | i <- <0..N-1>>))\X

-- The following are some functional programming constructs used above

cnth(i,xs) = if i==0 then head(xs)
                    else cnth(i-1,tail(xs))

fst((x,y)) = x
snd((x,y)) = y

-- The following function can be useful for partitioning a process list
-- into roughly equal-sized pieces for structured compression

groupsof(n)(xs) = let xl=#xs within
                  if xl==0 then <> else
                  if xl<=n or n==0 then <xs>
                  else let
                         m=if (xl/n)*n==xl then n else (n+1)
                    within
                    <take(m)(xs)>^groupsof(n)(drop(m)(xs))

take(n)(xs)  = if n==0 then <> else <head(xs)>^take(n-1)(tail(xs))

drop(n)(xs)  = if n==0 then xs else drop(n-1)(tail(xs))

-- The following define some similar compression functions for PStruct


StructPar(t) = let (P,_) = SPA(t) within P

SPA(PSLeaf.(P,A)) = (P,A)

SPA(PSNode.ts) = let ps = <SPA(t) | t <- ts>
                     A = Union(set(<a_ | (_,a_)  <- ps>))
                  within
                  (ListPar(ps),A)

```

(continues on next page)

(continued from previous page)

```
PSmap(f,PSLeaf.p) = PSLeaf.(f(p))
PSmap(f,PSNode.ts) = PSNode.<PSmap(f,t) | t <- ts>

PSvocab(t) = let as = psalphas(t)
             within
               Union({inter(cnth(i,as),cnth(j,as)) |
                              i <- {1..(#as)-1}, j <- {0..i-1}})

psalphas(PSLeaf.(P,A)) = <A>
psalphas(PSNode.ts) = <A | u <- ts, A <- psalphas(u)>
--psalphas(PSNode.ts) = <>

CompressPSLeaves(compress)(t)(X) = let V = PSvocab(t)
                                       H = diff(X,V)
                                       f((P,A)) = (compress(P\H),A)
                                   within
                                   PSmap(f,t)


PSLeafCompress(compress)(t)(X) = let ct = CompressPSLeaves(compress)(t)(X)
                                 within
                                 StructPar(ct)\X

psalphabet(PSLeaf.(P,A)) = A
psalphabet(PSNode.ts) = let AS = <psalphabet(t) | t <- ts>
                            within Union(set(AS))


PSStructCompress(compress) =
      let G(PSLeaf.(P,A)) = let f(X) = P\X within f
          G(PSNode.ts) =  \X @
                             let as = <psalphabet(t) | t <- ts>
                                 tlv = Union({inter(cnth(i,as),cnth(j,as)) |
                                             i <- {1..#ts-1}, j <- {0..i-1}})

                                 ps = <(compress(PSStructCompress(compress)(t)(
                                       inter(psalphabet(t), diff(X,tlv)))),
                                       psalphabet(t))
                                       | t <- ts >
                              within
                               ListPar(ps)\X
        within G
```

CHAPTER

TEN

REFERENCES

CHAPTER

ELEVEN

LICENSES

FDR incorporates software from a number of other sources. The licenses of any piece of software that requires its license to be reproduced are given below.

11.1 boost

```
Boost Software License - Version 1.0 - August 17th, 2003

Permission is hereby granted, free of charge, to any person or organization
obtaining a copy of the software and accompanying documentation covered by
this license (the "Software") to use, reproduce, display, distribute,
execute, and transmit the Software, and to prepare derivative works of the
Software, and to permit third-parties to whom the Software is furnished to
do so, all subject to the following:

The copyright notices in the Software and this entire statement, including
the above license grant, this restriction and the following disclaimer,
must be included in all copies of the Software, in whole or in part, and
all derivative works of the Software, unless such copies or derivative
works are solely in the form of machine-executable object code generated by
a source language processor.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT
SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE
FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE,
ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

11.2 boost.nowide

```
Boost Software License - Version 1.0 - August 17th, 2003

Permission is hereby granted, free of charge, to any person or organization
obtaining a copy of the software and accompanying documentation covered by
this license (the "Software") to use, reproduce, display, distribute,
execute, and transmit the Software, and to prepare derivative works of the
Software, and to permit third-parties to whom the Software is furnished to
do so, all subject to the following:
```

(continues on next page)

(continued from previous page)

```
The copyright notices in the Software and this entire statement, including
the above license grant, this restriction and the following disclaimer,
must be included in all copies of the Software, in whole or in part, and
all derivative works of the Software, unless such copies or derivative
works are solely in the form of machine-executable object code generated by
a source language processor.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT
SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE
FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE,
ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

11.3 CityHash

```
Copyright (c) 2011 Google, Inc.

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
THE SOFTWARE.
```

11.4 google-sparsehash

```
Copyright (c) 2005, Google Inc.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are
met:

    * Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.
    * Redistributions in binary form must reproduce the above
copyright notice, this list of conditions and the following disclaimer
```

(continues on next page)

(continued from previous page)

```
in the documentation and/or other materials provided with the
distribution.
    * Neither the name of Google Inc. nor the names of its
contributors may be used to endorse or promote products derived from
this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

11.5 graphviz

The source code from GraphViz as used in FDR3 can be obtained directly from the GraphViz website (version 2.30.1 is currently in use).

```
Eclipse Public License - v 1.0
THE ACCOMPANYING PROGRAM IS PROVIDED UNDER THE TERMS OF THIS ECLIPSE PUBLIC LICENSE (
↪"AGREEMENT"). ANY USE, REPRODUCTION OR DISTRIBUTION OF THE PROGRAM CONSTITUTES
↪RECIPIENT'S ACCEPTANCE OF THIS AGREEMENT.

1. DEFINITIONS

"Contribution" means:

a) in the case of the initial Contributor, the initial code and documentation
↪distributed under this Agreement, and

b) in the case of each subsequent Contributor:

i) changes to the Program, and

ii) additions to the Program;

where such changes and/or additions to the Program originate from and are distributed
↪by that particular Contributor. A Contribution 'originates' from a Contributor if
↪it was added to the Program by such Contributor itself or anyone acting on such
↪Contributor's behalf. Contributions do not include additions to the Program which:
↪(i) are separate modules of software distributed in conjunction with the Program
↪under their own license agreement, and (ii) are not derivative works of the Program.

"Contributor" means any person or entity that distributes the Program.

"Licensed Patents" mean patent claims licensable by a Contributor which are
↪necessarily infringed by the use or sale of its Contribution alone or when combined
↪with the Program.
```

(continues on next page)

(continued from previous page)

```
"Program" means the Contributions distributed in accordance with this Agreement.

"Recipient" means anyone who receives the Program under this Agreement, including all␣
↪Contributors.

2. GRANT OF RIGHTS

a) Subject to the terms of this Agreement, each Contributor hereby grants Recipient a␣
↪non-exclusive, worldwide, royalty-free copyright license to reproduce, prepare␣
↪derivative works of, publicly display, publicly perform, distribute and sublicense␣
↪the Contribution of such Contributor, if any, and such derivative works, in source␣
↪code and object code form.

b) Subject to the terms of this Agreement, each Contributor hereby grants Recipient a␣
↪non-exclusive, worldwide, royalty-free patent license under Licensed Patents to␣
↪make, use, sell, offer to sell, import and otherwise transfer the Contribution of␣
↪such Contributor, if any, in source code and object code form. This patent license␣
↪shall apply to the combination of the Contribution and the Program if, at the time␣
↪the Contribution is added by the Contributor, such addition of the Contribution␣
↪causes such combination to be covered by the Licensed Patents. The patent license␣
↪shall not apply to any other combinations which include the Contribution. No␣
↪hardware per se is licensed hereunder.

c) Recipient understands that although each Contributor grants the licenses to its␣
↪Contributions set forth herein, no assurances are provided by any Contributor that␣
↪the Program does not infringe the patent or other intellectual property rights of␣
↪any other entity. Each Contributor disclaims any liability to Recipient for claims␣
↪brought by any other entity based on infringement of intellectual property rights␣
↪or otherwise. As a condition to exercising the rights and licenses granted␣
↪hereunder, each Recipient hereby assumes sole responsibility to secure any other␣
↪intellectual property rights needed, if any. For example, if a third party patent␣
↪license is required to allow Recipient to distribute the Program, it is Recipient's␣
↪responsibility to acquire that license before distributing the Program.

d) Each Contributor represents that to its knowledge it has sufficient copyright␣
↪rights in its Contribution, if any, to grant the copyright license set forth in␣
↪this Agreement.

3. REQUIREMENTS

A Contributor may choose to distribute the Program in object code form under its own␣
↪license agreement, provided that:

a) it complies with the terms and conditions of this Agreement; and

b) its license agreement:

i) effectively disclaims on behalf of all Contributors all warranties and conditions,␣
↪express and implied, including warranties or conditions of title and non-
↪infringement, and implied warranties or conditions of merchantability and fitness␣
↪for a particular purpose;

ii) effectively excludes on behalf of all Contributors all liability for damages,␣
↪including direct, indirect, special, incidental and consequential damages, such as␣
↪lost profits;

iii) states that any provisions which differ from this Agreement are offered by that␣
↪Contributor alone and not by any other party; and
```

(continues on next page)

(continued from previous page)

```
iv) states that source code for the Program is available from such Contributor, and␣
↪informs licensees how to obtain it in a reasonable manner on or through a medium␣
↪customarily used for software exchange.

When the Program is made available in source code form:

a) it must be made available under this Agreement; and

b) a copy of this Agreement must be included with each copy of the Program.

Contributors may not remove or alter any copyright notices contained within the␣
↪Program.

Each Contributor must identify itself as the originator of its Contribution, if any,␣
↪in a manner that reasonably allows subsequent Recipients to identify the originator␣
↪of the Contribution.

4. COMMERCIAL DISTRIBUTION

Commercial distributors of software may accept certain responsibilities with respect␣
↪to end users, business partners and the like. While this license is intended to␣
↪facilitate the commercial use of the Program, the Contributor who includes the␣
↪Program in a commercial product offering should do so in a manner which does not␣
↪create potential liability for other Contributors. Therefore, if a Contributor␣
↪includes the Program in a commercial product offering, such Contributor (
↪"Commercial Contributor") hereby agrees to defend and indemnify every other␣
↪Contributor ("Indemnified Contributor") against any losses, damages and costs␣
↪(collectively "Losses") arising from claims, lawsuits and other legal actions␣
↪brought by a third party against the Indemnified Contributor to the extent caused␣
↪by the acts or omissions of such Commercial Contributor in connection with its␣
↪distribution of the Program in a commercial product offering. The obligations in␣
↪this section do not apply to any claims or Losses relating to any actual or alleged␣
↪intellectual property infringement. In order to qualify, an Indemnified Contributor␣
↪must: a) promptly notify the Commercial Contributor in writing of such claim, and␣
↪b) allow the Commercial Contributor to control, and cooperate with the Commercial␣
↪Contributor in, the defense and any related settlement negotiations. The␣
↪Indemnified Contributor may participate in any such claim at its own expense.

For example, a Contributor might include the Program in a commercial product offering,
↪ Product X. That Contributor is then a Commercial Contributor. If that Commercial␣
↪Contributor then makes performance claims, or offers warranties related to Product␣
↪X, those performance claims and warranties are such Commercial Contributor's␣
↪responsibility alone. Under this section, the Commercial Contributor would have to␣
↪defend claims against the other Contributors related to those performance claims␣
↪and warranties, and if a court requires any other Contributor to pay any damages as␣
↪a result, the Commercial Contributor must pay those damages.

5. NO WARRANTY

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, THE PROGRAM IS PROVIDED ON AN "AS IS
↪" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, EITHER EXPRESS OR IMPLIED␣
↪INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES OR CONDITIONS OF TITLE, NON-
↪INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Each Recipient␣
↪is solely responsible for determining the appropriateness of using and distributing␣
↪the Program and assumes all risks associated with its exercise of rights under this␣
↪Agreement , including but not limited to the risks and costs of program errors,␣
↪compliance with applicable laws, damage to or loss of data, programs or equipment,␣
↪and unavailability or interruption of operations.
```

(continues on next page)

(continued from previous page)

```
6. DISCLAIMER OF LIABILITY

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, NEITHER RECIPIENT NOR ANY␣
↪CONTRIBUTORS SHALL HAVE ANY LIABILITY FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
↪ EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING WITHOUT LIMITATION LOST PROFITS),␣
↪HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT␣
↪LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF␣
↪THE USE OR DISTRIBUTION OF THE PROGRAM OR THE EXERCISE OF ANY RIGHTS GRANTED␣
↪HEREUNDER, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. GENERAL

If any provision of this Agreement is invalid or unenforceable under applicable law,␣
↪it shall not affect the validity or enforceability of the remainder of the terms of␣
↪this Agreement, and without further action by the parties hereto, such provision␣
↪shall be reformed to the minimum extent necessary to make such provision valid and␣
↪enforceable.

If Recipient institutes patent litigation against any entity (including a cross-claim␣
↪or counterclaim in a lawsuit) alleging that the Program itself (excluding␣
↪combinations of the Program with other software or hardware) infringes such␣
↪Recipient's patent(s), then such Recipient's rights granted under Section 2(b)␣
↪shall terminate as of the date such litigation is filed.

All Recipient's rights under this Agreement shall terminate if it fails to comply␣
↪with any of the material terms or conditions of this Agreement and does not cure␣
↪such failure in a reasonable period of time after becoming aware of such␣
↪noncompliance. If all Recipient's rights under this Agreement terminate, Recipient␣
↪agrees to cease use and distribution of the Program as soon as reasonably␣
↪practicable. However, Recipient's obligations under this Agreement and any licenses␣
↪granted by Recipient relating to the Program shall continue and survive.

Everyone is permitted to copy and distribute copies of this Agreement, but in order␣
↪to avoid inconsistency the Agreement is copyrighted and may only be modified in the␣
↪following manner. The Agreement Steward reserves the right to publish new versions␣
↪(including revisions) of this Agreement from time to time. No one other than the␣
↪Agreement Steward has the right to modify this Agreement. The Eclipse Foundation is␣
↪the initial Agreement Steward. The Eclipse Foundation may assign the responsibility␣
↪to serve as the Agreement Steward to a suitable separate entity. Each new version␣
↪of the Agreement will be given a distinguishing version number. The Program␣
↪(including Contributions) may always be distributed subject to the version of the␣
↪Agreement under which it was received. In addition, after a new version of the␣
↪Agreement is published, Contributor may elect to distribute the Program (including␣
↪its Contributions) under the new version. Except as expressly stated in Sections␣
↪2(a) and 2(b) above, Recipient receives no rights or licenses to the intellectual␣
↪property of any Contributor under this Agreement, whether expressly, by implication,
↪ estoppel or otherwise. All rights in the Program not expressly granted under this␣
↪Agreement are reserved.

This Agreement is governed by the laws of the State of New York and the intellectual␣
↪property laws of the United States of America. No party to this Agreement will␣
↪bring a legal action under this Agreement more than one year after the cause of␣
↪action arose. Each party waives its rights to a jury trial in any resulting␣
↪litigation.
```

11.6 libcspm

11.7 LLVM

(continues on next page)

(continued from previous page)

```
      this list of conditions and the following disclaimers.

    * Redistributions in binary form must reproduce the above copyright notice,
      this list of conditions and the following disclaimers in the
      documentation and/or other materials provided with the distribution.

    * Neither the names of the LLVM Team, University of Illinois at
      Urbana-Champaign, nor the names of its contributors may be used to
      endorse or promote products derived from this Software without specific
      prior written permission.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS
FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.  IN NO EVENT SHALL THE
CONTRIBUTORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS WITH THE
SOFTWARE.

==============================================================================
Copyrights and Licenses for Third Party Software Distributed with LLVM:
==============================================================================
The LLVM software contains code written by third parties.  Such software will
have its own individual LICENSE.TXT file in the directory in which it appears.
This file will describe the copyrights, license, and restrictions which apply
to that code.

The disclaimer of warranty in the University of Illinois Open Source License
applies to all code in the LLVM Distribution, and nothing in any of the
other licenses gives permission to use the names of the LLVM Team or the
University of Illinois to endorse or promote products derived from this
Software.

The following pieces of software have additional or alternate copyrights,
licenses, and/or restrictions:

Program             Directory
-------             ---------
Autoconf            llvm/autoconf
                    llvm/projects/ModuleMaker/autoconf
                    llvm/projects/sample/autoconf
Google Test         llvm/utils/unittest/googletest
OpenBSD regex       llvm/lib/Support/{reg*, COPYRIGHT.regex}
pyyaml tests        llvm/test/YAMLParser/{*.data, LICENSE.TXT}
ARM contributions   llvm/lib/Target/ARM/LICENSE.TXT
```

11.8 lz4

```
LZ4 - Fast LZ compression algorithm
Copyright (C) 2011-2012, Yann Collet.
BSD 2-Clause License (http://www.opensource.org/licenses/bsd-license.php)

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are
```

(continues on next page)

(continued from previous page)

```
met:

   * Redistributions of source code must retain the above copyright
notice, this list of conditions and the following disclaimer.
   * Redistributions in binary form must reproduce the above
copyright notice, this list of conditions and the following disclaimer
in the documentation and/or other materials provided with the
distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

You can contact the author at :
- LZ4 homepage : http://fastcompression.blogspot.com/p/lz4.html
- LZ4 source repository : http://code.google.com/p/lz4/
```

11.9 popcount.h

```
New BSD License

Copyright (c) 2013, Kim Walisch.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

    * Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.
    * Redistributions in binary form must reproduce the above copyright
      notice, this list of conditions and the following disclaimer in the
      documentation and/or other materials provided with the distribution.
    * Neither the name of the author nor the names of its contributors may
      be used to endorse or promote products derived from this software
      without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL <COPYRIGHT HOLDER> BE LIABLE FOR ANY
DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

11.10 QT

(continues on next page)

(continued from previous page)

```
author's reputation will not be affected by problems that might be
introduced by others.

 Finally, software patents pose a constant threat to the existence of
any free program.  We wish to make sure that a company cannot
effectively restrict the users of a free program by obtaining a
restrictive license from a patent holder.  Therefore, we insist that
any patent license obtained for a version of the library must be
consistent with the full freedom of use specified in this license.

 Most GNU software, including some libraries, is covered by the
ordinary GNU General Public License.  This license, the GNU Lesser
General Public License, applies to certain designated libraries, and
is quite different from the ordinary General Public License.  We use
this license for certain libraries in order to permit linking those
libraries into non-free programs.

 When a program is linked with a library, whether statically or using
a shared library, the combination of the two is legally speaking a
combined work, a derivative of the original library.  The ordinary
General Public License therefore permits such linking only if the
entire combination fits its criteria of freedom.  The Lesser General
Public License permits more lax criteria for linking other code with
the library.

 We call this license the "Lesser" General Public License because it
does Less to protect the user's freedom than the ordinary General
Public License.  It also provides other free software developers Less
of an advantage over competing non-free programs.  These disadvantages
are the reason we use the ordinary General Public License for many
libraries.  However, the Lesser license provides advantages in certain
special circumstances.

 For example, on rare occasions, there may be a special need to
encourage the widest possible use of a certain library, so that it becomes
a de-facto standard.  To achieve this, non-free programs must be
allowed to use the library.  A more frequent case is that a free
library does the same job as widely used non-free libraries.  In this
case, there is little to gain by limiting the free library to free
software only, so we use the Lesser General Public License.

 In other cases, permission to use a particular library in non-free
programs enables a greater number of people to use a large body of
free software.  For example, permission to use the GNU C Library in
non-free programs enables many more people to use the whole GNU
operating system, as well as its variant, the GNU/Linux operating
system.

 Although the Lesser General Public License is Less protective of the
users' freedom, it does ensure that the user of a program that is
linked with the Library has the freedom and the wherewithal to run
that program using a modified version of the Library.

 The precise terms and conditions for copying, distribution and
modification follow.  Pay close attention to the difference between a
"work based on the library" and a "work that uses the library".  The
former contains code derived from the library, whereas the latter must
```

(continues on next page)

(continued from previous page)

```
be combined with the library in order to run.

                  GNU LESSER GENERAL PUBLIC LICENSE
   TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

  0. This License Agreement applies to any software library or other
program which contains a notice placed by the copyright holder or
other authorized party saying it may be distributed under the terms of
this Lesser General Public License (also called "this License").
Each licensee is addressed as "you".

  A "library" means a collection of software functions and/or data
prepared so as to be conveniently linked with application programs
(which use some of those functions and data) to form executables.

  The "Library", below, refers to any such software library or work
which has been distributed under these terms.  A "work based on the
Library" means either the Library or any derivative work under
copyright law: that is to say, a work containing the Library or a
portion of it, either verbatim or with modifications and/or translated
straightforwardly into another language.  (Hereinafter, translation is
included without limitation in the term "modification".)

  "Source code" for a work means the preferred form of the work for
making modifications to it.  For a library, complete source code means
all the source code for all modules it contains, plus any associated
interface definition files, plus the scripts used to control compilation
and installation of the library.

  Activities other than copying, distribution and modification are not
covered by this License; they are outside its scope.  The act of
running a program using the Library is not restricted, and output from
such a program is covered only if its contents constitute a work based
on the Library (independent of the use of the Library in a tool for
writing it).  Whether that is true depends on what the Library does
and what the program that uses the Library does.

  1. You may copy and distribute verbatim copies of the Library's
complete source code as you receive it, in any medium, provided that
you conspicuously and appropriately publish on each copy an
appropriate copyright notice and disclaimer of warranty; keep intact
all the notices that refer to this License and to the absence of any
warranty; and distribute a copy of this License along with the
Library.

  You may charge a fee for the physical act of transferring a copy,
and you may at your option offer warranty protection in exchange for a
fee.

  2. You may modify your copy or copies of the Library or any portion
of it, thus forming a work based on the Library, and copy and
distribute such modifications or work under the terms of Section 1
above, provided that you also meet all of these conditions:

    a) The modified work must itself be a software library.

    b) You must cause the files modified to carry prominent notices
```

(continues on next page)

(continued from previous page)

```
   stating that you changed the files and the date of any change.

   c) You must cause the whole of the work to be licensed at no
   charge to all third parties under the terms of this License.

   d) If a facility in the modified Library refers to a function or a
   table of data to be supplied by an application program that uses
   the facility, other than as an argument passed when the facility
   is invoked, then you must make a good faith effort to ensure that,
   in the event an application does not supply such function or
   table, the facility still operates, and performs whatever part of
   its purpose remains meaningful.

   (For example, a function in a library to compute square roots has
   a purpose that is entirely well-defined independent of the
   application.  Therefore, Subsection 2d requires that any
   application-supplied function or table used by this function must
   be optional: if the application does not supply it, the square
   root function must still compute square roots.)

These requirements apply to the modified work as a whole.  If
identifiable sections of that work are not derived from the Library,
and can be reasonably considered independent and separate works in
themselves, then this License, and its terms, do not apply to those
sections when you distribute them as separate works.  But when you
distribute the same sections as part of a whole which is a work based
on the Library, the distribution of the whole must be on the terms of
this License, whose permissions for other licensees extend to the
entire whole, and thus to each and every part regardless of who wrote
it.

Thus, it is not the intent of this section to claim rights or contest
your rights to work written entirely by you; rather, the intent is to
exercise the right to control the distribution of derivative or
collective works based on the Library.

In addition, mere aggregation of another work not based on the Library
with the Library (or with a work based on the Library) on a volume of
a storage or distribution medium does not bring the other work under
the scope of this License.

 3. You may opt to apply the terms of the ordinary GNU General Public
License instead of this License to a given copy of the Library.  To do
this, you must alter all the notices that refer to this License, so
that they refer to the ordinary GNU General Public License, version 2,
instead of to this License.  (If a newer version than version 2 of the
ordinary GNU General Public License has appeared, then you can specify
that version instead if you wish.)  Do not make any other change in
these notices.

 Once this change is made in a given copy, it is irreversible for
that copy, so the ordinary GNU General Public License applies to all
subsequent copies and derivative works made from that copy.

 This option is useful when you wish to copy part of the code of
the Library into a program that is not a library.
```

(continues on next page)

(continued from previous page)

```
 4. You may copy and distribute the Library (or a portion or
derivative of it, under Section 2) in object code or executable form
under the terms of Sections 1 and 2 above provided that you accompany
it with the complete corresponding machine-readable source code, which
must be distributed under the terms of Sections 1 and 2 above on a
medium customarily used for software interchange.

 If distribution of object code is made by offering access to copy
from a designated place, then offering equivalent access to copy the
source code from the same place satisfies the requirement to
distribute the source code, even though third parties are not
compelled to copy the source along with the object code.

 5. A program that contains no derivative of any portion of the
Library, but is designed to work with the Library by being compiled or
linked with it, is called a "work that uses the Library".  Such a
work, in isolation, is not a derivative work of the Library, and
therefore falls outside the scope of this License.

 However, linking a "work that uses the Library" with the Library
creates an executable that is a derivative of the Library (because it
contains portions of the Library), rather than a "work that uses the
library".  The executable is therefore covered by this License.
Section 6 states terms for distribution of such executables.

 When a "work that uses the Library" uses material from a header file
that is part of the Library, the object code for the work may be a
derivative work of the Library even though the source code is not.
Whether this is true is especially significant if the work can be
linked without the Library, or if the work is itself a library.  The
threshold for this to be true is not precisely defined by law.

 If such an object file uses only numerical parameters, data
structure layouts and accessors, and small macros and small inline
functions (ten lines or less in length), then the use of the object
file is unrestricted, regardless of whether it is legally a derivative
work.  (Executables containing this object code plus portions of the
Library will still fall under Section 6.)

 Otherwise, if the work is a derivative of the Library, you may
distribute the object code for the work under the terms of Section 6.
Any executables containing that work also fall under Section 6,
whether or not they are linked directly with the Library itself.

 6. As an exception to the Sections above, you may also combine or
link a "work that uses the Library" with the Library to produce a
work containing portions of the Library, and distribute that work
under terms of your choice, provided that the terms permit
modification of the work for the customer's own use and reverse
engineering for debugging such modifications.

 You must give prominent notice with each copy of the work that the
Library is used in it and that the Library and its use are covered by
this License.  You must supply a copy of this License.  If the work
during execution displays copyright notices, you must include the
copyright notice for the Library among them, as well as a reference
directing the user to the copy of this License.  Also, you must do one
```

(continues on next page)

(continued from previous page)

```
of these things:

   a) Accompany the work with the complete corresponding
   machine-readable source code for the Library including whatever
   changes were used in the work (which must be distributed under
   Sections 1 and 2 above); and, if the work is an executable linked
   with the Library, with the complete machine-readable "work that
   uses the Library", as object code and/or source code, so that the
   user can modify the Library and then relink to produce a modified
   executable containing the modified Library.  (It is understood
   that the user who changes the contents of definitions files in the
   Library will not necessarily be able to recompile the application
   to use the modified definitions.)

   b) Use a suitable shared library mechanism for linking with the
   Library.  A suitable mechanism is one that (1) uses at run time a
   copy of the library already present on the user's computer system,
   rather than copying library functions into the executable, and (2)
   will operate properly with a modified version of the library, if
   the user installs one, as long as the modified version is
   interface-compatible with the version that the work was made with.

   c) Accompany the work with a written offer, valid for at
   least three years, to give the same user the materials
   specified in Subsection 6a, above, for a charge no more
   than the cost of performing this distribution.

   d) If distribution of the work is made by offering access to copy
   from a designated place, offer equivalent access to copy the above
   specified materials from the same place.

   e) Verify that the user has already received a copy of these
   materials or that you have already sent this user a copy.

 For an executable, the required form of the "work that uses the
Library" must include any data and utility programs needed for
reproducing the executable from it.  However, as a special exception,
the materials to be distributed need not include anything that is
normally distributed (in either source or binary form) with the major
components (compiler, kernel, and so on) of the operating system on
which the executable runs, unless that component itself accompanies
the executable.

 It may happen that this requirement contradicts the license
restrictions of other proprietary libraries that do not normally
accompany the operating system.  Such a contradiction means you cannot
use both them and the Library together in an executable that you
distribute.

 7. You may place library facilities that are a work based on the
Library side-by-side in a single library together with other library
facilities not covered by this License, and distribute such a combined
library, provided that the separate distribution of the work based on
the Library and of the other library facilities is otherwise
permitted, and provided that you do these two things:

   a) Accompany the combined library with a copy of the same work
```

(continues on next page)

(continued from previous page)

```
   based on the Library, uncombined with any other library
   facilities.  This must be distributed under the terms of the
   Sections above.

   b) Give prominent notice with the combined library of the fact
   that part of it is a work based on the Library, and explaining
   where to find the accompanying uncombined form of the same work.

 8. You may not copy, modify, sublicense, link with, or distribute
0the Library except as expressly provided under this License.  Any
attempt otherwise to copy, modify, sublicense, link with, or
distribute the Library is void, and will automatically terminate your
rights under this License.  However, parties who have received copies,
or rights, from you under this License will not have their licenses
terminated so long as such parties remain in full compliance.

 9. You are not required to accept this License, since you have not
signed it.  However, nothing else grants you permission to modify or
distribute the Library or its derivative works.  These actions are
prohibited by law if you do not accept this License.  Therefore, by
modifying or distributing the Library (or any work based on the
Library), you indicate your acceptance of this License to do so, and
all its terms and conditions for copying, distributing or modifying
the Library or works based on it.

 10. Each time you redistribute the Library (or any work based on the
Library), the recipient automatically receives a license from the
original licensor to copy, distribute, link with or modify the Library
subject to these terms and conditions.  You may not impose any further
restrictions on the recipients' exercise of the rights granted herein.
You are not responsible for enforcing compliance by third parties with
this License.

 11. If, as a consequence of a court judgment or allegation of patent
infringement or for any other reason (not limited to patent issues),
conditions are imposed on you (whether by court order, agreement or
otherwise) that contradict the conditions of this License, they do not
excuse you from the conditions of this License.  If you cannot
distribute so as to satisfy simultaneously your obligations under this
License and any other pertinent obligations, then as a consequence you
may not distribute the Library at all.  For example, if a patent
license would not permit royalty-free redistribution of the Library by
all those who receive copies directly or indirectly through you, then
the only way you could satisfy both it and this License would be to
refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any
particular circumstance, the balance of the section is intended to apply,
and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any
patents or other property right claims or to contest validity of any
such claims; this section has the sole purpose of protecting the
integrity of the free software distribution system which is
implemented by public license practices.  Many people have made
generous contributions to the wide range of software distributed
through that system in reliance on consistent application of that
```

(continues on next page)

(continued from previous page)

```
system; it is up to the author/donor to decide if he or she is willing
to distribute software through any other system and a licensee cannot
impose that choice.

This section is intended to make thoroughly clear what is believed to
be a consequence of the rest of this License.

 12. If the distribution and/or use of the Library is restricted in
certain countries either by patents or by copyrighted interfaces, the
original copyright holder who places the Library under this License may add
an explicit geographical distribution limitation excluding those countries,
so that distribution is permitted only in or among countries not thus
excluded.  In such case, this License incorporates the limitation as if
written in the body of this License.

 13. The Free Software Foundation may publish revised and/or new
versions of the Lesser General Public License from time to time.
Such new versions will be similar in spirit to the present version,
but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number.  If the Library
specifies a version number of this License which applies to it and
"any later version", you have the option of following the terms and
conditions either of that version or of any later version published by
the Free Software Foundation.  If the Library does not specify a
license version number, you may choose any version ever published by
the Free Software Foundation.

 14. If you wish to incorporate parts of the Library into other free
programs whose distribution conditions are incompatible with these,
write to the author to ask for permission.  For software which is
copyrighted by the Free Software Foundation, write to the Free
Software Foundation; we sometimes make exceptions for this.  Our
decision will be guided by the two goals of preserving the free status
of all derivatives of our free software and of promoting the sharing
and reuse of software generally.

                           NO WARRANTY

 15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO
WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW.
EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR
OTHER PARTIES PROVIDE THE LIBRARY "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.  THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE
LIBRARY IS WITH YOU.  SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME
THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

 16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN
WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY
AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU
FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR
CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE
LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING
RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A
FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF
```

(continues on next page)

(continued from previous page)

```
SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH
DAMAGES.

                    END OF TERMS AND CONDITIONS

          How to Apply These Terms to Your New Libraries

 If you develop a new library, and you want it to be of the greatest
possible use to the public, we recommend making it free software that
everyone can redistribute and change.  You can do so by permitting
redistribution under these terms (or, alternatively, under the terms of the
ordinary General Public License).

 To apply these terms, attach the following notices to the library.  It is
safest to attach them to the start of each source file to most effectively
convey the exclusion of warranty; and each file should have at least the
"copyright" line and a pointer to where the full notice is found.

   <one line to give the library's name and a brief idea of what it does.>
   Copyright (C) <year>  <name of author>

   This library is free software; you can redistribute it and/or
   modify it under the terms of the GNU Lesser General Public
   License as published by the Free Software Foundation; either
   version 2.1 of the License, or (at your option) any later version.

   This library is distributed in the hope that it will be useful,
   but WITHOUT ANY WARRANTY; without even the implied warranty of
   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
   Lesser General Public License for more details.

   You should have received a copy of the GNU Lesser General Public
   License along with this library; if not, write to the Free Software
   Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301  USA

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your
school, if any, to sign a "copyright disclaimer" for the library, if
necessary.  Here is a sample; alter the names:

 Yoyodyne, Inc., hereby disclaims all copyright interest in the
 library `Frob' (a library for tweaking knobs) written by James Random Hacker.

 <signature of Ty Coon>, 1 April 1990
 Ty Coon, President of Vice

That's all there is to it!
```

11.11 zlib

```
Copyright (C) 1995-2013 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
```

(continues on next page)

(continued from previous page)

```
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

11.12 Haskell

11.12.1 GHC

```
The Glasgow Haskell Compiler License

Copyright 2002, The University Court of the University of Glasgow.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice,
this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright notice,
this list of conditions and the following disclaimer in the documentation
and/or other materials provided with the distribution.

- Neither name of the University nor the names of its contributors may be
used to endorse or promote products derived from this software without
specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE UNIVERSITY COURT OF THE UNIVERSITY OF
GLASGOW AND THE CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,
INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
UNIVERSITY COURT OF THE UNIVERSITY OF GLASGOW OR THE CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
DAMAGE.
```

11.12.2 array/base/containers/directory/ghc-prim/integer-gmp/old-locale/old-time/pretty/text

```
This library (libraries/base) is derived from code from several
sources:

  * Code from the GHC project which is largely (c) The University of
    Glasgow, and distributable under a BSD-style license (see below),

  * Code from the Haskell 98 Report which is (c) Simon Peyton Jones
    and freely redistributable (but see the full license for
    restrictions).

  * Code from the Haskell Foreign Function Interface specification,
    which is (c) Manuel M. T. Chakravarty and freely redistributable
    (but see the full license for restrictions).

The full text of these licenses is reproduced below.  All of the
licenses are BSD-style or compatible.

-----------------------------------------------------------------------------

The Glasgow Haskell Compiler License

Copyright 2004, The University Court of the University of Glasgow.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice,
this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright notice,
this list of conditions and the following disclaimer in the documentation
and/or other materials provided with the distribution.

- Neither name of the University nor the names of its contributors may be
used to endorse or promote products derived from this software without
specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE UNIVERSITY COURT OF THE UNIVERSITY OF
GLASGOW AND THE CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,
INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
UNIVERSITY COURT OF THE UNIVERSITY OF GLASGOW OR THE CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
DAMAGE.

-----------------------------------------------------------------------------

Code derived from the document "Report on the Programming Language
```

(continues on next page)

(continued from previous page)

```
Haskell 98", is distributed under the following license:

  Copyright (c) 2002 Simon Peyton Jones

  The authors intend this Report to belong to the entire Haskell
  community, and so we grant permission to copy and distribute it for
  any purpose, provided that it is reproduced in its entirety,
  including this Notice.  Modified versions of this Report may also be
  copied and distributed for any purpose, provided that the modified
  version is clearly presented as such, and that it does not claim to
  be a definition of the Haskell 98 Language.

-----------------------------------------------------------------------------

Code derived from the document "The Haskell 98 Foreign Function
Interface, An Addendum to the Haskell 98 Report" is distributed under
the following license:

  Copyright (c) 2002 Manuel M. T. Chakravarty

  The authors intend this Report to belong to the entire Haskell
  community, and so we grant permission to copy and distribute it for
  any purpose, provided that it is reproduced in its entirety,
  including this Notice.  Modified versions of this Report may also be
  copied and distributed for any purpose, provided that the modified
  version is clearly presented as such, and that it does not claim to
  be a definition of the Haskell 98 Foreign Function Interface.

-----------------------------------------------------------------------------
```

11.12.3 bytestring

```
Copyright (c) Don Stewart 2005-2009
          (c) Duncan Coutts 2006-2011
          (c) David Roundy 2003-2005
          (c) Simon Meier 2010-2011

All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the author nor the names of his contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHORS OR CONTRIBUTORS BE LIABLE
```

(continues on next page)

(continued from previous page)

11.12.4 deepseq

11.12.5 filepath

```
Copyright Neil Mitchell 2005-2007.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are
met:

    * Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.

    * Redistributions in binary form must reproduce the above
      copyright notice, this list of conditions and the following
      disclaimer in the documentation and/or other materials provided
      with the distribution.

    * Neither the name of Neil Mitchell nor the names of other
      contributors may be used to endorse or promote products derived
      from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

11.12.6 graph-wrapper

```
Copyright Max Bolingbroke 2006-2007.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are
met:

    * Redistributions of source code must retain the above copyright
      notice, this list of conditions and the following disclaimer.

    * Redistributions in binary form must reproduce the above
      copyright notice, this list of conditions and the following
      disclaimer in the documentation and/or other materials provided
      with the distribution.

    * Neither the name of Max Bolingbroke nor the names of other
      contributors may be used to endorse or promote products derived
      from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
```

(continues on next page)

(continued from previous page)

11.12.7 hashable

11.12.8 hashtables

(continues on next page)

(continued from previous page)

11.12.9 libgmp

(continues on next page)

(continued from previous page)

```
  The "Minimal Corresponding Source" for a Combined Work means the
Corresponding Source for the Combined Work, excluding any source code
for portions of the Combined Work that, considered in isolation, are
based on the Application, and not on the Linked Version.

  The "Corresponding Application Code" for a Combined Work means the
object code and/or source code for the Application, including any data
and utility programs needed for reproducing the Combined Work from the
Application, but excluding the System Libraries of the Combined Work.

  1. Exception to Section 3 of the GNU GPL.

  You may convey a covered work under sections 3 and 4 of this License
without being bound by section 3 of the GNU GPL.

  2. Conveying Modified Versions.

  If you modify a copy of the Library, and, in your modifications, a
facility refers to a function or data to be supplied by an Application
that uses the facility (other than as an argument passed when the
facility is invoked), then you may convey a copy of the modified
version:

   a) under this License, provided that you make a good faith effort to
   ensure that, in the event an Application does not supply the
   function or data, the facility still operates, and performs
   whatever part of its purpose remains meaningful, or

   b) under the GNU GPL, with none of the additional permissions of
   this License applicable to that copy.

  3. Object Code Incorporating Material from Library Header Files.

  The object code form of an Application may incorporate material from
a header file that is part of the Library.  You may convey such object
code under terms of your choice, provided that, if the incorporated
material is not limited to numerical parameters, data structure
layouts and accessors, or small macros, inline functions and templates
(ten or fewer lines in length), you do both of the following:

   a) Give prominent notice with each copy of the object code that the
   Library is used in it and that the Library and its use are
   covered by this License.

   b) Accompany the object code with a copy of the GNU GPL and this license
   document.

  4. Combined Works.

  You may convey a Combined Work under terms of your choice that,
taken together, effectively do not restrict modification of the
portions of the Library contained in the Combined Work and reverse
engineering for debugging such modifications, if you also do each of
the following:

   a) Give prominent notice with each copy of the Combined Work that
```

(continues on next page)

(continued from previous page)

```
   the Library is used in it and that the Library and its use are
   covered by this License.

   b) Accompany the Combined Work with a copy of the GNU GPL and this license
   document.

   c) For a Combined Work that displays copyright notices during
   execution, include the copyright notice for the Library among
   these notices, as well as a reference directing the user to the
   copies of the GNU GPL and this license document.

   d) Do one of the following:

       0) Convey the Minimal Corresponding Source under the terms of this
       License, and the Corresponding Application Code in a form
       suitable for, and under terms that permit, the user to
       recombine or relink the Application with a modified version of
       the Linked Version to produce a modified Combined Work, in the
       manner specified by section 6 of the GNU GPL for conveying
       Corresponding Source.

       1) Use a suitable shared library mechanism for linking with the
       Library.  A suitable mechanism is one that (a) uses at run time
       a copy of the Library already present on the user's computer
       system, and (b) will operate properly with a modified version
       of the Library that is interface-compatible with the Linked
       Version.

   e) Provide Installation Information, but only if you would otherwise
   be required to provide such information under section 6 of the
   GNU GPL, and only to the extent that such information is
   necessary to install and execute a modified version of the
   Combined Work produced by recombining or relinking the
   Application with a modified version of the Linked Version. (If
   you use option 4d0, the Installation Information must accompany
   the Minimal Corresponding Source and Corresponding Application
   Code. If you use option 4d1, you must provide the Installation
   Information in the manner specified by section 6 of the GNU GPL
   for conveying Corresponding Source.)

  5. Combined Libraries.

  You may place library facilities that are a work based on the
Library side by side in a single library together with other library
facilities that are not Applications and are not covered by this
License, and convey such a combined library under terms of your
choice, if you do both of the following:

   a) Accompany the combined library with a copy of the same work based
   on the Library, uncombined with any other library facilities,
   conveyed under the terms of this License.

   b) Give prominent notice with the combined library that part of it
   is a work based on the Library, and explaining where to find the
   accompanying uncombined form of the same work.

  6. Revised Versions of the GNU Lesser General Public License.
```

(continues on next page)

(continued from previous page)

```
  The Free Software Foundation may publish revised and/or new versions
of the GNU Lesser General Public License from time to time. Such new
versions will be similar in spirit to the present version, but may
differ in detail to address new problems or concerns.

  Each version is given a distinguishing version number. If the
Library as you received it specifies that a certain numbered version
of the GNU Lesser General Public License "or any later version"
applies to it, you have the option of following the terms and
conditions either of that published version or of any later version
published by the Free Software Foundation. If the Library as you
received it does not specify a version number of the GNU Lesser
General Public License, you may choose any version of the GNU Lesser
General Public License ever published by the Free Software Foundation.

  If the Library as you received it specifies that a proxy can decide
whether future versions of the GNU Lesser General Public License shall
apply, that proxy's public statement of acceptance of any version is
permanent authorization for you to choose that version for the
Library.
```

11.12.10 mtl

```
The Glasgow Haskell Compiler License

Copyright 2004, The University Court of the University of Glasgow.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice,
this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright notice,
this list of conditions and the following disclaimer in the documentation
and/or other materials provided with the distribution.

- Neither name of the University nor the names of its contributors may be
used to endorse or promote products derived from this software without
specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE UNIVERSITY COURT OF THE UNIVERSITY OF
GLASGOW AND THE CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,
INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
UNIVERSITY COURT OF THE UNIVERSITY OF GLASGOW OR THE CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
DAMAGE.
```

11.12.11 primitive

```
Copyright (c) 2008-2009, Roman Leshchinskiy
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice,
this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright notice,
this list of conditions and the following disclaimer in the documentation
and/or other materials provided with the distribution.

- Neither name of the University nor the names of its contributors may be
used to endorse or promote products derived from this software without
specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE UNIVERSITY COURT OF THE UNIVERSITY OF
GLASGOW AND THE CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,
INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
UNIVERSITY COURT OF THE UNIVERSITY OF GLASGOW OR THE CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
DAMAGE.
```

11.12.12 transformers

```
The Glasgow Haskell Compiler License

Copyright 2004, The University Court of the University of Glasgow.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice,
this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright notice,
this list of conditions and the following disclaimer in the documentation
and/or other materials provided with the distribution.

- Neither name of the University nor the names of its contributors may be
used to endorse or promote products derived from this software without
specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE UNIVERSITY COURT OF THE UNIVERSITY OF
GLASGOW AND THE CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,
```

(continues on next page)

(continued from previous page)

```
INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
UNIVERSITY COURT OF THE UNIVERSITY OF GLASGOW OR THE CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
DAMAGE.
```

11.12.13 unix

```
The Glasgow Haskell Compiler License

Copyright 2004, The University Court of the University of Glasgow.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice,
this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright notice,
this list of conditions and the following disclaimer in the documentation
and/or other materials provided with the distribution.

- Neither name of the University nor the names of its contributors may be
used to endorse or promote products derived from this software without
specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE UNIVERSITY COURT OF THE UNIVERSITY OF
GLASGOW AND THE CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,
INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
UNIVERSITY COURT OF THE UNIVERSITY OF GLASGOW OR THE CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH
DAMAGE.
```

11.12.14 value-supply

```
Copyright (c) 2006 Iavor S. Diatchki

Permission is hereby granted, free of charge, to any person obtaining a copy of this␣
↪software and associated documentation files (the "Software"), to deal in the␣
↪Software without restriction, including without limitation the rights to use, copy,␣
↪modify, merge, publish, distribute, sublicense, and/or sell copies of the Software,␣
↪and to permit persons to whom the Software is furnished to do so, subject to the␣
↪following conditions:
```

(continues on next page)

(continued from previous page)

11.12.15 vector

BIBLIOGRAPHY

[ALOR12] Philip Armstrong, Gavin Lowe, Joel Ouaknine, and A. W. Roscoe. Model checking Timed CSP. HOWARD. Easychair, pub. 2012.

[GRABR14] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and A.W. Roscoe. FDR3 — A Modern Refinement Checker for CSP. Tools and Algorithms for the Construction and Analysis of Systems 2014. LNCS vol. 8413.

[GMRWZ03] Michael Goldsmith, Nick Moffat, A. W. Roscoe, Tim Whitworth, and Irfan Zakiuddin Watchdog Transformations for Property-Oriented Model-Checking. FME 2003: Formal Methods. LNCS vol. 2805.

[Ros98] A. W. Roscoe. The Theory and Practice of Concurrency. Prentice Hall. 1998.

[Ros10] A. W. Roscoe. Understanding Concurrent Systems. Springer. 2010.

INDEX

Symbols

A

B

C

D

U

V

W

Y