NAME

torx-intro − introduction to the Cote de Resyste testing tool torx

DESCRIPTION

This page briefly discusses TorX terminology that might confuse readers coming from a different background. In addition, it gives an overview of the commands offered by torx, together with its utilities, interfaces, etc.

TERMINOLOGY

CHANNEL

Here we discuss a concept that might cause confusion.

The concept **channel** is used in TorX in two places: in the promela specification language that is used as input for the promela primer, and elswhere in TorX, e.g. in interfaces and configuration files of TorX tool components. It is important to understand that the *only* place in TorX where **channel** is used with the meaning that it has in promela, is in the promela specifications. *Everywhere else in TorX* the concept **channel** is interpreted as in the **MIOCO** extension of the **IOCO** testing theory, where a channel is a (group of) point(s) to interact (interface) with the implementation. A channel is unidirectional. This means in particular that multiple promela channels can belong to a single TorX (MIOCO) channel. Currently, when we use the IOCO testing theory, in all cases we use exactely two (MIOCO) channels, one for input (in) and one for output (out). A theoretical definition can be found on page 34 of Lex Heerink's PhD thesis "Ins and Outs in Refusal Testing".

LTS

The Explorer component in TorX gives access to a Labelled Transition System (LTS) representation of the specification (model) fed into it. In the specification, the LTS may be present explicitly (as is the case for the Aldebaran (.aut) files) in which case it will be finite, or implicitly (as is the case for, for example, LOTOS), in which case it (the number of states and/or transitions) may be infinite.

The Primer uses the LTS offered by an Explorer and partitions the transitions into inputs and outputs (stimuli and observations), adds quiescent transitions, and (lazily) determinizes it. The result of this can be seen again as a LTS, which can be explored using **primexp**(1).

INTEGRATION WITH OTHER TOOL(KIT)S

CADP

There is a ''bi-directional'' connection with the CADP toolkit via the open/caesar interface. Via **mkprimer**(1) TorX can be a *user* of the open/caesar interface to be able to use CADP ''explorers'', and it can also via **torx_open**(1) be a *provider* of the open/caesar interface to allow CADP tools (simulators, state space generators, etc.) to use the TorX ''explorers'', or even to explore the LTS offered by TorX ''primers'' (by using **primexp**(1) in combination with **torx_open**(1)).

mucrl

TorX can explore mucrl specifications using the program mucrl from the mucrl toolkit and the programs tbf2lpe and lpe2torx from the mcrl2 toolkit.

mcrl2

TorX can explore mcrl2 specifications using the programs mcrl2 and lpe2torx from the mcrl2 toolkit.

ltsa

TorX can explore fsp specifications using the **ltsaexp**(1) explorer that builds on the fsp ''explorer'' that is present in the LTSA tool.

GraphViz

The graphviz toolkit is used for visualization (using graphviz program dot) and animation (using graphiz tcl extension tcldot) of graphs, automata etc.

LIST OF COMMANDS

The commands are grouped below as follows: for each of the components (modules) in the TorX architecture (driver, primer, explorer, adapter) there is a corresponding command group. The other commands consist of utilities of various kinds, most notably visualization and file format conversion.

User Interfaces

Name	Description
pui(1)	simple primer user interface
xtorx(1)	graphical user interface for **torx**(1)

User Interface Utilities

Name	Description
torx-logclient(1)	connect torx log monitor command to torx
torx-querypr(1)	query torx problem report database
torx-sendpr(1)	submit torx problem report
xtorx-showmsc(1)	show a TorX run log as Message Sequence Chart
xtorx-showspec(1)	show the specification (source) of a primer or mutant

Program Interface For External Tools

Name	Description
torx_open(1)	offer open/caesar API access to (LTS of) TorX explorer program and via **primexp**(1) to (LTS of) TorX primer program

Driver

Name	Description
torx(1)	execute test on the fly

Explorer

Name	Description
autexp(1)	explore Aldebaran (.aut) automaton files
jararaca(1)	explore traces generated from regular expressions
ltsaexp(1)	use ltsa as **torx-explorer**(5) for the language fsp
smileexp(1)	use smile as symbolic **torx-explorer**(5) for LOTOS
primexp(1)	provide torx-explorer interface to torx primer

Primer

Name	Description
primer(1)	compute test primitives using explorer
intersector(1)	combine multiple torx primers

Primer Creation

Name	Description
mkprimer(1)	generate a primer
mkprimer-aut(1)	generate a AUT primer that uses **autexp**(1)
mkprimer-cadp(1)	generate a LOTOS, BCG, FC2 or AUT primer using CADP
mkprimer-jararaca(1)	generate a TP or JARARACA primer that uses **jararaca**(1)
mkprimer-ltsa(1)	generate an FSP primer that uses **ltsaexp**(1)
mkprimer-mcrl(1)	generate a mCRL primer
mkprimer-mcrl2(1)	generate a mCRL2 primer
mkprimer-trojka(1)	generate a promela primer using trojka
preprocmkprimer(1)	preprocess input before invoking **mkprimer**(1)
cppmkprimer(1)	preprocess input with **cpp**(1) before invoking **mkprimer**(1)

m4mkprimer(1)	preprocess input with **m4**(1) before invoking **mkprimer**(1)

Adapters

Name	Description
adaptor(1)	default TorX program to interface to the SUT
adaptlog(1)	TorX program to interface to a **torx-log**(4) log file used as SUT
adaptsim(1)	TorX program to interface to a **torx-primer**(5) used as SUT

Adapter Utilities

Name	Description
tcp(1)	connection program for tcp
udp(1)	connection program for udp
hexcontext(1)	hex encode/decode stdio of (IUT) program
unhexify(1)	translate from hexadecimal to ascii

Other TorX Components

Name	Description
instantiator(1)	instantiate free variables for torx
iochooser(1)	suggest by probabilities to stimulate or observe
partitioner(1)	weight-based test primitive selection for primer

Visualization

Name	Description
anifsm(1)	animate and edit graph in dot format and write Aldebaran (.aut) automaton
aniwait(1)	animate progressbar
jararacy(1)	animate **jararaca**(1) trace using **lefty**(1)
mctrl(1)	animation progress scrollbar
mscviewer(1)	view Message Sequence Chart in window
Bmsc(1)	shell command to load Message Sequence Chart file(s) into running **mscviewer**(1)

Visualization Utilities

Name	Description
tmcs(1)	tcp multicast service program

Format Conversion

Name	Description
aut2fsmview(1)	translate Aldebaran (.aut) to FSMView input
jararacy2anifsm(1)	translate from **jararacy**(1) to **anifsm**(1) input format
log2anifsm(1)	extract info from **torx-log**(4) file for animation with **anifsm**(1)
log2aniwait(1)	extract information for **aniwait**(1) from **torx-log**(4) file
log2aut(1)	extract states and transitions from **torx-log**(4) file for **autexp**(1) and **anifsm**(1)
log2jararacy(1)	extract states and transitions from **torx-log**(4) file for **jararacy**(1) and **anifsm**(1)
log2mctrl(1)	extract step numbers from **torx-log**(4) file for animation with

	mctrl(1)
log2msc(1)	extract Message Sequence Chart from **torx-log**(4) file
log2primer(1)	generate **torx-primer**(5) commands from **torx-log**(4) file

Various

Name	Description
autsimplify(1)	simplify automaton Aldebaran (.aut) file
campaign(1)	a *very experimental* tool (and language) to describe a test campaign and populate a directory structure with Makefiles and configuration files
torx-mans(1)	list TorX manual page file names
torx-root(1)	report torx installation directory
torx-hostname(1)	print hostname taken from network database

LIST OF INTERFACES

Name	Description
mkprimer(5)	(**perl**) API to add support for a specification language or toolkit to **mkprimer**(1)
torx-adaptor(5)	(interface to) en/decoder and connector to system-under-test for **torx**(1)
torx-explorer(5)	(interface to) explore a labelled transition system for **torx**(1)
torx-instantiator(5)	(interface to) instantiator program for **torx**(1)
torx-primer(5)	(interface to) primer (specification) program for **torx**(1)
xtorx-extension(n)	(**tcl**) API to specify *Primers* and/or *Mutants* menu for **xtorx**(1)

LIST OF FILE FORMATS

Name	Description
torx-config(4)	configuration file for **torx**(1)
torx-log(4)	test run log file generated by **torx**(1)

SEE ALSO

Lex Heerink, *Ins and Outs in Refusal Testing*, PhD thesis, University of Twente, The Netherlands, 1998. ISBN 90-365-1128-3

CONTACT

By Email: <torx_support@cs.utwente.nl>
On the Web: <URL:http://www.purl.org/net/torx/>

VERSION

This manual page documents version 3.9.0 of torx.

NAME

mscviewer − view a Message Sequence Chart

SYNOPSIS

mscviewer [-r] [-m *mcastid* **] [** *files ...* **]**
Bmsc [-r] [-m *mcastid* **] [** *files ...* **]**
Bmsc -exit

DESCRIPTION

The **mscviewer** program reads MSC's from *files*, or from standard input if no files are given, and displays it to the user, step by step. Each MSC is displayed in a separate window. Instead of waiting for the whole MSC to be available, it will immediately start displaying what it has read, and update the display as soon as it has been able to read more of the MSC.

Bmsc is a shell-level command that causes a running **mscviewer** to load the named MSC files, or to display its standard input. The connection between **Bmsc** and a running **mscviewer** will not be closed until all *files* (or the complete standard input) of the **Bmsc** command have been processed by **mscviewer**, in order to allow the running **mscviewer** to report possible error messages (e.g. about syntax errors) about the files that it processes via the standard error of the **Bmsc** command that sent the files to it. If **Bmsc** cannot find a running **mscviewer**, it will start a new one. To display the new MSC file(s), **mscviewer** will reuse windows that contain a complete MSC and have the **Reuse** toggle activated. If more windows are needed, they are created.

In general, it is probably best to only use the **Bmsc** command, and let it start **mscviewer** when necessary. However, one should be aware of the fact that when a **Bmsc** command is given when no **mscviewer** is currently running, the **Bmsc** will ''become'' a **mscviewer** command, which is ''long-running'' and will only exit when all its windows are closed or the **Quit** button is pressed (or a **Bmsc -exit** command is given). In contrast, a **Bmsc** command given when a **mscviewer** is already running will exit as soon as its files or standard input are processed by the running **mscviewer**.

The **-r** command line option of both **mscviewer** and **Bmsc** will activate the **Reuse** toggle button for the windows that will contain the MSC's given on the same command line or via standard input.

When **Bmsc** is started with only command line option **-m** *mcastid*, or when environment variable **TORXMCASTID** was set, the MSC viewer tries to connect to the address given in the *mcastid* and to use the resulting connection as a remote control connection to synchronise displaying a particular step in the MSC viewer. Whenever the user does something in the user interface that selects a different step in the MSC, its step number is written to the remote control connection. Whenever a step number can be read from the remote control connection, the corresponding step is displayed in the MSC viewer.

When **Bmsc** is started with only one command line parameter: **-exit**, the running **mscviewer** will clean up and exit.

The MSC file should be in *event oriented* textual representation. **mscviewer** indicates both ''normal'' end-of-msc and ''abnormal'' end-of-input without having seen end-of-msc. The ''normal'' end-of-msc is visualized by drawing horizontal bars at the end of every instance in the MSC. The ''abnormal'' end-of-input is visualized by drawing at the end of each instance of the MSC a stippled/dotted contininuation of the instance, and ending that with stippled/dotted horizontal bars.

BUTTONS

At the bottom of the MSC viewer there are several buttons. The **Save as** button opens a dialog box that allows saving of the MSC in postscript form (by choosing or entering a file name ending in a .ps suffix) and in textual form (by choosing or entering any other file name).
The **Font** down and up arrow buttons decrement resp. increment the font size. When a font size change makes this necessary, labels are moved to the right to keep them visible.
The **Highlight** toggle button enables and disables highlighting (default: enabled). Independent of this button, the *step number* of the MSC item under the mouse is shown in the **Step** field. Step numbers start at 1, and are assigened when the *second* part (target) of a message is seen. Step number 0 is special: it used to refer to the instance headers. When highlighting is enabled, the item under the mouse is highlighted by

drawing a box arround it and making the arrow slightly bigger. Also, when a new item is added to the MSC, it is highlighed. To highlight the item for a known *step*, enter the step number in the **Step** entry field, and hit the return key. The MSC window automatically scrolls to make the highlighted item visible. If a step number is present in the **Step** field, the down and up arrow buttons can be used to decrement resp. increment the step number, to move the highlight up resp. down in the MSC.
The **Reuse** toggle button indicates that its window may be reused for a new MSC, when end-of-input has been seen for the MSC currently displayed in it. (default value: unset, except when overridden by a **-r** command line option of **mscviewer** or **Bmsc**).
The **Close** button closes the MSC window, and, if this was the last remaining window, exits the progam.
The **Quit** button closes all MSC windows and exits the progam.

SEE ALSO

torx-intro(1), **xtorx-showmsc**(1), **log2msc**(1), **torx-logclient**(1), **jararacy**(1), **tmcs**(1),
Ekkart Rudolph, Peter Graubmann and Jens Grabowski: *Tutorial on Message Sequence Charts*, Computer Networks and ISDN Systems, Volume 28, Issue 12, June 1996, Pages 1629-1641

FILES

/tmp/mscviewer-*$USER-$DISPLAY*
file to communicate tcp port number on which **mscviewer** listens for **Bmsc** to connect

/tmp/mscviewer-*$USER-$DISPLAY***.pid**
the file containing a list of process identifiers (one per line) of **mscviewer** and its subprocesses

NOTE

The **Bmsc** command was named (and designed) after the **B** shell-level command of the **sam**(1) editor.

BUGS

The current implementation expects each ‘‘statement’’ of the MSC in event oriented textual representation to be on a separate line. The output of **log2msc**(1) complies to this limitation.

The ‘‘endinstance’’ statements in the MSC are ignored; the ‘‘endsmsc’’ statement is used to close all instances.

Only a limited subset of the MSC language is implemented. Valid input is assumed; only very limited checking is done.

The syntax recognized for the MSC language is inferred from the tutorial mentioned above, but not checked with a more formal syntax description. In particular, **mscviewer** expects double quotes (") to be present for MSC items containing whitespace -- whether this is consistent with the MSC standard has not been checked.

When **mscviewer** is started, it checks if other instances of it are running. If so, they are killed. This was added to clean up run-away processes.

When **mscviewer** is given multiple files that are to be processed simultaneously, it has a tendency to process the files one after the other, in reverse order, instead of procesing them in parallel, step by step.

It is counter-intuitive that the **Step** *up* arrow button moves the highlight *down* (because the up button increments the step number, and the steps are numbered increasing from top to bottom).

NAME

adaptlog − torx program to use a torx logfile as implementation

SYNOPSIS

adaptlog

DESCRIPTION

This adaptor program implements the Driver-Adapter interface, as discussed in **torx-adaptor**(5), to use a TorX log file as discussed in **torx-log**(4), as implementation. The log file should be configured as the argument for the **IUT** configuration entry discussed in **torx-config**(4), i.e. the **IUT** configuration line should look like

IUT my-path-to-my-logfile.log

BUGS

The reuse (overloading) of the **IUT** configuration keyword is a crock, but adding a new keyword to **torx-config**(4) did not seem a really more attractive alternative.

SEE ALSO

torx-intro(1), **torx-config**(4), **torx-log**(4), **torx**(1)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of **torx**.

NAME

adaptor − default torx program to interface to the SUT

SYNOPSIS

ADAPTOR

DESCRIPTION

This adaptor program implements the Driver-Adapter interface, as discussed in **torx-adaptor**(5). It expects that the user has implemented encoding and decoding routines in Tcl (Tool Command Language), that can be accessed as discussed in the adaptor-specific parts of **torx-config**(4).

BUGS

Generally it is easier to write your own adapter than it is to configure this one.

SEE ALSO

torx-intro(1), **torx-config**(4), **torx**(1)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of **torx**.

NAME

adaptsim − torx program to interface to a TorX primer used as SUT

SYNOPSIS

adaptsim

DESCRIPTION

This adaptor program implements the Driver-Adapter interface, as discussed in **torx-adaptor**(5), to use a TorX primer program as implementation. It expects that the IUT program that it connects to implements (a subset of) the TorX Primer-Driver interface. The IUT program should be configured as **IUT** according to **torx-config**(4).

SEE ALSO

torx-intro(1), **torx-config**(4), **torx**(1)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of **torx**.

NAME

anifsm − animate, construct or edit graphs in dot format

SYNOPSIS

anifsm [−r] [−m *mcastid* **] [−t** *title* **] [−k** *key* **] [** *dotfiles* **]**
anifsm [−r] [−m *mcastid* **] [−t** *title* **] [−k** *key* **] −**
anifsmsrv
anifsm −exit

DESCRIPTION

anifsm uses **tcldot**(1) to animate, construct or edit graphs in **dot** format. In TorX it is used to animate the automaton (RFSM) file(s) generated by **jararaca**(1), **autexp**(1), or any other explorer, and to on-the-fly construct the automaton of the test run. The automaton represented by the graph can be written out to file in Aldebaran (.aut) format. This allows **anifsm** to be used as a graphical editor to construct simple automata in Aldebaran (.aut) format.

After start up, **anifsm** creates a window (with the given *title*) for each file in *dotfiles* (or just a single window if no *dotfiles* was given, or if *dotfiles* consists only of the special name ''−'') in which it draws the automaton for that file and then for the last (or only) window waits for commands (for animation or layout) on standard input. The animation and layout commands are discussed below in COMMANDS. On end of file on standard input, or when the user removes the window (or presses the **Quit** button which tells **anifsmsrv** to stop running) **anifsm** exits.
Note: the special treatment of ''−'' is deprecated and may disappear is future versions.

In each **anifsm** window the graph can be edited (constructed, changed) using the mouse. The left mouse button is used to create nodes and edges; the right mouse button is used to delete them, to edit their attributes, and to post a pop-up menu which includes entries to save the graph to file in **dot**, **Aldebaran (.aut)**, and **Postscript** format. For further details see EDITING below. Details of the transformation to Aldebaran (.aut) format are discussed below in AUTOMATON.

The middle mouse button can be used to scroll the canvas in its window by moving the mouse with the middle button pressed. When the middle button is clicked without moving the mouse the canvas of all clones of that window is scrolled to show the positition at which the mouse was clicked. When the Control key is pressed while the mouse is moved with the middle button pressed the canvases of all clones of the window are continuously scrolled to show the item under mouse.

Actually, **anifsm** is a shell-level command that uses a running **anifsmsrv** to load the named dot file, and animate it using animation commands on standard input. The connection between **anifsm** and a running **anifsmsrv** will not be closed until the complete standard input of the **anifsm** command has been processed by **anifsmsrv**. If **anifsm** cannot find a running **anifsmsrv**, it will start a new one. To display the new dot file, **anifsm** will reuse windows, but only those that contain a completed animation, and have the **Reuse** toggle activated. To choose which window to reuse, **anifsm** uses the **−k** *key* command line option. If there are reusable windows with the same *key*, one of those will be used. Otherwise, if **anifsm** was invoked without **−k** *key* option, it will look for reusable windows with a non-empty key. If **anifsm** was invoked with a non-empty **−k** *key* option, it will look for reusable windows with an empty key. If none of the above is succesful, a new window will be created.
In general, it should not be necessary to start **anifsmsrv** by hand.

The **−r** command line option of **anifsm** will activate the **Reuse** toggle button for the window that will contain the dotfile given on the same command line.

The **−k** *key* command line option of **anifsm** will associate *key* with the window in which the given dotfile is animated. The *key* will be displayed to the right of the **Reuse** button. The *key* is used to guide the reuse of windows in which the animation is finished, as discussed above.

To make a running **anifsmsrv** go away invoke **anifsm** with the **−exit** command line option.

The animation in the window will follow the animation commands read from standard input. The animation can be stepped through manually using the **Step** up and down arrow buttons (as discussed below in

BUTTONS).

In addition, the animation can be remotely controlled. If the **−m** *mcastid* command line option is given, or environment variable **TORXMCASTID** was set, **anifsm** will attempt to make a remote control connection to the tcp address in *mcastid*. If it succeeds, it will then interpret lines of text read from the remote control connection consisting of a single number as commands to show the corresponding step in the animation. Additionally, whenever the user uses mouse button and/or navigation commands to show a different step, its step number is written to the remote control connection. The remote control connection allows multiple viewers to show the same test step.

COMMANDS

Each animation or dot layout command consists of a single line of text. The animation and layout commands can appear interspersed, see EXAMPLES below. The display is updated after execution of each individual command (unless the **Render** button is disabled, in which case the display is not updated for dot layout commands).

The animation commands are expected to be generated using **log2jararacy**(1) and **jararacy2anifsm**(1), e.g. using a unix command as

log2jararacy < *logfile* | **jararacy2anifsm** | **anifsm** *dotfile*

or

tail −f *logfile* | **log2jararacy** | **jararacy2anifsm** | **anifsm** *dotfile*

Make sure that the *logfile* contains a run of the automaton present in *dotfile*.

A mix of animation and layout (graph operation) commands is expected to be generated by **log2anifsm**(1) e.g. using a unix command as

log2anifsm < *logfile* | **anifsm −**

or

tail −f *logfile* | **log2anifsm** | **anifsm −**

Of course, layout and animation commands can also be generated by other programs, or even be written by hand.

LAYOUT COMMANDS

The dot layout (graph operation) commands start with the word **dot**, followed by the command (either **subgraph**, **node**, **edge**, **delnode**, or **deledge**), followed by the argument to the command, followed by optional attributes. The items in a dot layout command are separated by whitespace. For each optional attribute its name and its value are given, separated by whitespace. The general form is:

dot *command argument aname avalue aname avalue ...*

Known layout commands and their arguments are:

dot subgraph *subgraphname aname avalue aname avalue ...*

Create a subgraph named *subgraphname*. Each group of *aname* and *avalue* defines an attribute of the subgraph. The usual dot subgraph attributes can be specified. An attribute with name **subgraph** is treated special, to allow definition of nested subgraphs. The *subgraphname* can be used in subsequent subgraph or node commands, as value of an attribute named **subgraph** to add the a new subgraph or node to the subgraph specified earlier. See EXAMPLES below.

dot node *nodename aname avalue aname avalue ...*

Create a node named *nodename*. Each group of *aname* and *avalue* defines an attribute of the node. The usual dot node attributes can be specified. An attribute with name **subgraph** is treated special: it indicates that the node should be created in the subgraph with the name given in the value of the attribute. This subgraph should have been defined earlier. (Apart from the special treatment of the **subgraph** attribute) this commands corresponds to a dot file line of

nodename **[** *aname=avalue*, *aname=avalue*, ... **]**

The *nodename* can be used in subsequent **dot edge** dot layout commands. It is not necessary to define all nodes using this command: if no attributes need to be given, nodes

can be implicitly defined in the **dot edge** commands. The *nodename* will also be used as the label of the node in the animation, unless a **label** attribute is specified among the *aname* and *avalue*. In this respect it is very much like a node definition in an ordinary dot input file.

dot edge *edgelist aname avalue aname avalue ...*
Create one or more edges (and, implicitly nodes, for those nodes listed in *edgelist* that do not yet exist) as specified by *edgelist* which consists of a list of node names separated by −> (without any whitspace). So *edgelist* is of the form *src*−>*dst* or *n1*−>*n2*−>*n3* etc. Such a **dot edge** command specifies an edge from node *src* to node *dst*, or from *n1* to *n2* to *n3*, and each group of *aname* and *avalue* defines an attribute of the edge. It corresponds to a dot file line of
src −> *dst* **[** *aname*=*avalue*, *aname*=*avalue*, ... **]**
The usual dot edge attributes can be specified.

dot delnode *nodename*
Delete the node or nodes specified by *nodename*, together with their (incoming or outgoing) edges, from the graph.

dot deledge *edge* Delete the edge or edges specified by *edge* (a list of nodenames separated by −> or the value of a **name** attribute specified for an edge) from the graph.

COLOR ANIMATION COMMANDS

The animation commands consist of alternating commands and arguments:
command arguments command arguments ...
Known commands and their arguments are:

−**c** *color*	where *color* should be a color known by tcl/tk.
−**n** *nodes*	where *nodes* consists of a whitespace separated list of node identifiers.
−**e** *edges*	where *edges* consists of a whitespace separated list of edge identifiers.
−− *word*	do not try to interpret *word* as a command, but use it literally.

The node identifiers should be present in the dotfile or given in **dot node** or **dot edge** commands. The edge identifiers should be given in the dotfile as the value of a **name** attribute of an edge, as for example **e42** is given in
src −> *dst* **[label**=*action*, **name**=e42 , ...**];**
Alternatively, an edge identifier can be of the form
src−>*dst*
(note: no whitespace between *src*, −> and *dst*) where *src* and *dst* are node identifiers. Note, however, that if *src* and *dst* are linked by multiple edges, an arbitrary one will be chosen! It is much safer to rely on **name** attributes in the dotfile.
During animation, the given states and edges will be colored as indicated by the −**c** *color* command preceding it (reading from left to right). The initial color is red. An initial −**n** command is implied and may be omitted. Nodes and edges that are not mentioned in a command will have their initial color, as specified in the dotfile. If a node or edge is mentioned multiple times on a single command line, it will be colored according to the color specified for its last (rightmost) occurence.

BUTTONS

At the bottom of an anifsm window there are several buttons. The **Zoom** up and down arrow buttons zoom out resp. in. When zooming, the font size is adjusted accordingly. When the fontsize becomes too small to be usable, only the nodes and edges are drawn and the node and edge labels are omitted. When, as a consequence of zooming in, the fontsize increases again sufficiently, the node and edge labels are shown again.
The **Fit** button zooms the animation to fit in the window. If the **Auto** toggle button is enabled, the animation is automatically zoomed to fit the window when the window is resized.
If the **Render** toggle button is enabled (which is the default) dot layout commands read from standard input have immediate effect. Otherwise, the layout displayed will not be updated until the **Render** toggle button

is enabled again.
The *step number* of the animation step in the trace is shown in the **Step** field. Step numbers start at 0, for the initial state. To visualize the animation step for a known *step*, enter the step number in the **Step** entry field, and hit the return key. If a step number is present in the **Step** field, the down and up arrow buttons can be used to step backwards resp. forwards in the animation.
The **Reuse** toggle button indicates that its window may be reused for a new dotfile, when end-of-input has been seen for the dotfile currently displayed in it. While an animation is in progress (so, when end-of-input has not yet been seen) the **Reuse** button is disabled. (default value: unset, except when overridden by a **−r** command line option of **anifsm**).
To the right of the **Reuse** button, a menu button displays the current *key* value. The *key* value is used to guide the reuse of the window. Pressing the key menu button pops up a menu that offers the choice between all ‘‘known’’ keys. While an animation is in progress (so, when end-of-input has not yet been seen) the key menu button is disabled. (default value: empty, except when overridden by a **−k** *key* command line option of **anifsm).**
The **Clone** button creates a new anifsm window, showing the same animated dot file. As described above the canvas of the clone(s) can be made to scroll simultaneously to show the same item.
The **Close** button closes the window, and, if this was the last remaining anifsm window, exits the **anifsmsrv** program.
The **Quit** button closes all anifsm windows and exits the **anifsmsrv** program.

EDITING

The mouse can be used to edit the graph in an **anifsm** window. The left mouse button is used to create nodes and edges, the right mouse button is used to delete them, to edit their attributes, and to post a pop-up menu.

Clicking the left mouse button on the background of the canvas creates a new node.

Pressing the left mouse button on (in) a node and (while keeping the mouse button pressed) moving the mouse slightly starts the creation of an edge, as indicated by the red arrow which then appears. If the left mouse button is then released with the mouse cursor on (in) a node, an edge is created from the originating node to the destination node. If these nodes are the same a self-loop is created. If the mouse was released while the mouse cursor was not in a node, no edge is created (this can be used to cancel the creation of an edge). Just clicking the left mouse button on a node without moving the mouse at all does nothing (to avoid having to remove lots of unwanted self-loops).

Pressing the right mouse button on a node or edge pops up an attribute edit box, which allows editing of node or edge attributes, and deletion of the node or edge. The top part of the edit box contains attribute names (on the left) with their current values (on the right). The values that can be changed appear in entry fields. To change a value, edit it in the entry field, and press the return key to ’commit’ the change. To add an attribute not yet present, enter its name in the empty entry field on the left (under the ’known’ attribute names), and its value in its corresponding entry field on the right, and press the return key.
At the bottom of the edit box there are two buttons: **Delete**, which deletes the node or edge from the graph, and **Dismiss**, which makes the edit box disappear.

Pressing the right mouse button on a the background of the canvas pops up a menu that contains commands to create a new (inititially empty) window, to open (read) a dot file, to show some information, to connect to an mcast session, to write the graph in the window in dot, Aldebaran (.aut), or postscript format, and to pop up an edit box to edit global graph, node or edge attributes.

AUTOMATON

The graph in the window can be interpreted as an automaton. Special node attributes are used to indicate the start state, and to indicate those graph nodes that are not part of the automaton (those graph nodes will not be present in the automaton written in Aldebaran (.aut) format). Initial values of these attributes are set when the dot file is read, or, if no dot file was given, when the graph is constructed.

autstart when set to 1, indicates that this node is to be the start state of the (Aldebaran, .aut) automaton represented by the graph. At most one node in the graph can have this attribute with a non-zero and/or non-empty value.

autexclude when set to 1, indicates that this node should not appear in the (Aldebaran, .aut) automaton. When reading an automaton or graph from file, nodes of which either the node name or the value of the **label** attribute starts with an underscore will have this attribute set to 1.

When a graph is read or constructed, the start state will be determined as follows. Initially, the first node created in a graph will be the start state. Then, the nodes are inspected and where applicable **autexclude** attributes will be set. Finally the edges are inspected. If there is an edge from an aut-excluded node to a non-aut-excluded node, the non-aut-excluded will be the start state of the automaton. If there are multiple such edges, the 'last' one 'wins'.

When the Aldebaran (.aut) file is written, the transition names are taken from the label attributes of the edges. The state names are determined as follows. If a graph node has a label attribute, it is used to determine the state name, otherwise the node name is used. If all such names (of all non-autexcluded graph nodes) consist of the same string prefix followed by a number, the numbers are used as state numbers in the Aldebaran file. Otherwise the complete names are used (which make them just be numbers).

ATTRIBUTES

A number of node and edge attributes have direct effect on the appearance of the graph in **anifsm** (even though they (mostly) have no effect on the automaton). The definitive reference for these is the **dot (1)** manual page; we only list a number of them here for convenience.

In addition to the attribute names understood by dot there are a few node and edge attribute names that **dot** does not care about, but that are special for **anifsm**.

Attributes that have no value yet (that have the empty value) appear with **{}** as value in the edit box.

DOT NODE ATTRIBUTES

label the text that appears in a node. The value **{\N}** is special: it indicates that the node name should be used as label.

color the color of the node

fillcolor the fill color of the node, if its **style** is set to **filled** (if **fillcolor** is not set the value of **color** will be used)

style for example, filled

shape the shape of the node.

DOT EDGE ATTRIBUTES

label the text that appears with an edge.

color the color of the edge and its accompanying text

ANIFSM NODE ATTRIBUTES

autstart when set to 1, indicates that this node is to be the start state of the (Aldebaran, .aut) automaton represented by the graph. At most one node in the graph can have this attribute with a non-zero and/or non-empty value.

autexclude when set to 1, indicates that this node should not appear in the (Aldebaran, .aut) automaton.

subgraph gives the name of the subgraph to which the node belongs.

ANIFSM EDGE ATTRIBUTES

name identifies the edge. This name can be used, for example, during animation to hightlight the edge. The same name may be assigned to multiple edges (which all will be highlighted when the name is used in an animation command).

EXAMPLES

Valid commands are:

S0
−n S0

−c red −n S0
−c #f00 −n S0

to color node S0 red; the commands are equivalent, the first uses the defaults. The last shows that in addition to color names also the #rgb color specifications of tcl/tk can be used.

−c green S0

to color node S0 green;

−c green S0 S1 −e e0 −c blue e1 −n S2 S3 S4 −e e2

to color nodes S0 and S1 and edge e0 green and nodes S2, S3 and S4 and edges e1 and e2 blue;

−e e0

to color edge e0 red.

Example of mix of dot layout and animation commands (note that we do not have to use **dot node** commands if we do not need to specify attributes for the nodes, and we do not have multiple nodes with the same name). We specify edges between nodes **a**, **b**, and **c**, with **name** attributes that we use in subsequent animation commands, and for the edge from **a** to **c** we specify a ''backwards'' direction, equivalent to **dir=back** in a dot file. We then specify some additional edges, without name attribute, so we use the *src*−>*dst* notation to refer to them in the subsequent animation commands.

```
dot edge a−>b name e0
dot edge b−>c name e1
dot edge a−>c name e2 dir back
−e e0
−e e1
−e e2
dot edge a−>d
dot edge d−>c
−e a−>d
−e d−>c
−e e2
```

Example of the creation of subgraphs. With the **subgraph** definition and **subgraph** attribute in the definition of both nodes **a** and **d** both nodes will be at the top, having the same rank. Without the subgraph definition node **d** would be next to node **b**.

```
dot subgraph g1 rank same
dot node a subgraph g1
dot node b
dot node c
dot node d subgraph g1
dot edge a−>b
dot edge b−>c
dot edge d−>c
```

The three dot edge lines above can be combined into two:

```
dot edge a−>b−>c
dot edge d−>c
```

DIAGNOSTICS

Error messages and navigation diagnostics appear on standard error.

BUGS

The environment variable **TORX_ROOT** is not supported.

Because the animation commands are read from standard input, it is not possible to read the *dotfile* from standard input. However, the (new) dot layout commands that can be given on standard input compensate for that to a certain extent.

The window does not automatically scroll to follow the colored states.

After an syntax error has been encountered in an input dotfile, **tcldot**(1) (at least the version in GraphViz 1.8.5) seems to be unable to recover sufficiently to be able to read more (syntactly correct) dotfiles.

After the revision of the syntax of the commands accepted on standard input to make it more general and take out the TorX specific features, the language accepted by **anifsm** differs from the one accepted by **jararacy**(1). The difference between the two is bridged by **jararacy2anifsm**(1). However, the fact that we have this difference breaks the fall-back to **jararacy**(1) that used to be present in **anifsm**, because if this fall-back would be used, **jararacy**(1) would be given the revised commands which it will not understand. As a consequence, the fall-back has been removed: if **anifsm** can not find tcl package Tcldot it will just give up.

The usage of **autstart** and **autexclude** attributes to indicate automaton features of the graph is clumsy.

When the attribute edit box is popped up, it tries to position itself under the mouse cursor, in an attempt to reduce the necessary mouse movements. We added this in the hope that it would be benificial, but it works not as benificial as intended: the fact that the box is moving all the time is very annoying. To be fixed.

There is no indication that a graph that was read in from a dot file has been altered using mouse commands.

There are no distinct view and edit modes (editing is always enabled). There is not enough experience (yet) with **anifsm** to tell whether such modes are actually needed, though.

SEE ALSO

torx-intro(1), **jararaca**(1), **jararacy2anifsm**(1), **log2jararacy**(1), **dot**(1), **doted**(1), **tcldot**(1), **jararacy**(1), **torx-logclient**(1), **tmcs**(1), **anifsm**(1), **aniwait**(1), **mscviewer**(1), **environ**(5)

ACKNOWLEDGEMENTS

Parts of **anifsm** (in particular the zooming and graph editing code) have been lifted and adapted from:
doted - dot graph editor - John Ellson (ellson@graphviz.org)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of torx.

NAME

anifsm − animate, construct or edit graphs in dot format

SYNOPSIS

anifsm [−r] [−m *mcastid* **] [−t** *title* **] [−k** *key* **] [** *dotfiles* **]**
anifsm [−r] [−m *mcastid* **] [−t** *title* **] [−k** *key* **] −**
anifsmsrv
anifsm −exit

DESCRIPTION

anifsm uses **tcldot**(1) to animate, construct or edit graphs in **dot** format. In TorX it is used to animate the automaton (RFSM) file(s) generated by **jararaca**(1), **autexp**(1), or any other explorer, and to on-the-fly construct the automaton of the test run. The automaton represented by the graph can be written out to file in Aldebaran (.aut) format. This allows **anifsm** to be used as a graphical editor to construct simple automata in Aldebaran (.aut) format.

After start up, **anifsm** creates a window (with the given *title*) for each file in *dotfiles* (or just a single window if no *dotfiles* was given, or if *dotfiles* consists only of the special name ''−'') in which it draws the automaton for that file and then for the last (or only) window waits for commands (for animation or layout) on standard input. The animation and layout commands are discussed below in COMMANDS. On end of file on standard input, or when the user removes the window (or presses the **Quit** button which tells **anifsmsrv** to stop running) **anifsm** exits.
Note: the special treatment of ''−'' is deprecated and may disappear is future versions.

In each **anifsm** window the graph can be edited (constructed, changed) using the mouse. The left mouse button is used to create nodes and edges; the right mouse button is used to delete them, to edit their attributes, and to post a pop-up menu which includes entries to save the graph to file in **dot**, **Aldebaran (.aut)**, and **Postscript** format. For further details see EDITING below. Details of the transformation to Aldebaran (.aut) format are discussed below in AUTOMATON.

The middle mouse button can be used to scroll the canvas in its window by moving the mouse with the middle button pressed. When the middle button is clicked without moving the mouse the canvas of all clones of that window is scrolled to show the positition at which the mouse was clicked. When the Control key is pressed while the mouse is moved with the middle button pressed the canvases of all clones of the window are continuously scrolled to show the item under mouse.

Actually, **anifsm** is a shell-level command that uses a running **anifsmsrv** to load the named dot file, and animate it using animation commands on standard input. The connection between **anifsm** and a running **anifsmsrv** will not be closed until the complete standard input of the **anifsm** command has been processed by **anifsmsrv**. If **anifsm** cannot find a running **anifsmsrv**, it will start a new one. To display the new dot file, **anifsm** will reuse windows, but only those that contain a completed animation, and have the **Reuse** toggle activated. To choose which window to reuse, **anifsm** uses the **−k** *key* command line option. If there are reusable windows with the same *key*, one of those will be used. Otherwise, if **anifsm** was invoked without **−k** *key* option, it will look for reusable windows with a non-empty key. If **anifsm** was invoked with a non-empty **−k** *key* option, it will look for reusable windows with an empty key. If none of the above is succesful, a new window will be created.
In general, it should not be necessary to start **anifsmsrv** by hand.

The **−r** command line option of **anifsm** will activate the **Reuse** toggle button for the window that will contain the dotfile given on the same command line.

The **−k** *key* command line option of **anifsm** will associate *key* with the window in which the given dotfile is animated. The *key* will be displayed to the right of the **Reuse** button. The *key* is used to guide the reuse of windows in which the animation is finished, as discussed above.

To make a running **anifsmsrv** go away invoke **anifsm** with the **−exit** command line option.

The animation in the window will follow the animation commands read from standard input. The animation can be stepped through manually using the **Step** up and down arrow buttons (as discussed below in

BUTTONS).

In addition, the animation can be remotely controlled. If the **−m** *mcastid* command line option is given, or environment variable **TORXMCASTID** was set, **anifsm** will attempt to make a remote control connection to the tcp address in *mcastid*. If it succeeds, it will then interpret lines of text read from the remote control connection consisting of a single number as commands to show the corresponding step in the animation. Additionally, whenever the user uses mouse button and/or navigation commands to show a different step, its step number is written to the remote control connection. The remote control connection allows multiple viewers to show the same test step.

COMMANDS

Each animation or dot layout command consists of a single line of text. The animation and layout commands can appear interspersed, see EXAMPLES below. The display is updated after execution of each individual command (unless the **Render** button is disabled, in which case the display is not updated for dot layout commands).

The animation commands are expected to be generated using **log2jararacy**(1) and **jararacy2anifsm**(1), e.g. using a unix command as

log2jararacy < *logfile* **| jararacy2anifsm | anifsm** *dotfile*

or

tail −f *logfile* **| log2jararacy | jararacy2anifsm | anifsm** *dotfile*

Make sure that the *logfile* contains a run of the automaton present in *dotfile***.**

A mix of animation and layout (graph operation) commands is expected to be generated by **log2anifsm**(1) e.g. using a unix command as

log2anifsm < *logfile* **| anifsm −**

or

tail −f *logfile* **| log2anifsm | anifsm −**

Of course, layout and animation commands can also be generated by other programs, or even be written by hand.

LAYOUT COMMANDS

The dot layout (graph operation) commands start with the word **dot**, followed by the command (either **subgraph**, **node**, **edge**, **delnode**, or **deledge**), followed by the argument to the command, followed by optional attributes. The items in a dot layout command are separated by whitespace. For each optional attribute its name and its value are given, separated by whitespace. The general form is:

dot *command argument aname avalue aname avalue ...*

Known layout commands and their arguments are:

dot subgraph *subgraphname aname avalue aname avalue ...*

Create a subgraph named *subgraphname*. Each group of *aname* and *avalue* defines an attribute of the subgraph. The usual dot subgraph attributes can be specified. An attribute with name **subgraph** is treated special, to allow definition of nested subgraphs. The *subgraphname* can be used in subsequent subgraph or node commands, as value of an attribute named **subgraph** to add the a new subgraph or node to the subgraph specified earlier. See EXAMPLES below.

dot node *nodename aname avalue aname avalue ...*

Create a node named *nodename*. Each group of *aname* and *avalue* defines an attribute of the node. The usual dot node attributes can be specified. An attribute with name **subgraph** is treated special: it indicates that the node should be created in the subgraph with the name given in the value of the attribute. This subgraph should have been defined earlier. (Apart from the special treatment of the **subgraph** attribute) this commands corresponds to a dot file line of

nodename **[** *aname=avalue***,** *aname=avalue***,** *...* **]**

The *nodename* can be used in subsequent **dot edge** dot layout commands. It is not necessary to define all nodes using this command: if no attributes need to be given, nodes

can be implicitly defined in the **dot edge** commands. The *nodename* will also be used as the label of the node in the animation, unless a **label** attribute is specified among the *aname* and *avalue*. In this respect it is very much like a node definition in an ordinary dot input file.

dot edge *edgelist aname avalue aname avalue ...*
Create one or more edges (and, implicitly nodes, for those nodes listed in *edgelist* that do not yet exist) as specified by *edgelist* which consists of a list of node names separated by −> (without any whitspace). So *edgelist* is of the form *src*−>*dst* or *n1*−>*n2*−>*n3* etc. Such a **dot edge** command specifies an edge from node *src* to node *dst*, or from *n1* to *n2* to *n3*, and each group of *aname* and *avalue* defines an attribute of the edge. It corresponds to a dot file line of
src −> *dst* **[** *aname*=*avalue*, *aname*=*avalue*, ... **]**
The usual dot edge attributes can be specified.

dot delnode *nodename*
Delete the node or nodes specified by *nodename*, together with their (incoming or outgoing) edges, from the graph.

dot deledge *edge* Delete the edge or edges specified by *edge* (a list of nodenames separated by −> or the value of a **name** attribute specified for an edge) from the graph.

COLOR ANIMATION COMMANDS

The animation commands consist of alternating commands and arguments:
command arguments command arguments ...
Known commands and their arguments are:

−**c** *color*	where *color* should be a color known by tcl/tk.
−**n** *nodes*	where *nodes* consists of a whitespace separated list of node identifiers.
−**e** *edges*	where *edges* consists of a whitespace separated list of edge identifiers.
−− *word*	do not try to interpret *word* as a command, but use it literally.

The node identifiers should be present in the dotfile or given in **dot node** or **dot edge** commands. The edge identifiers should be given in the dotfile as the value of a **name** attribute of an edge, as for example **e42** is given in
src −> *dst* **[label**=*action*, **name**=e42 , ...**];**
Alternatively, an edge identifier can be of the form
src−>*dst*
(note: no whitespace between *src*, −> and *dst*) where *src* and *dst* are node identifiers. Note, however, that if *src* and *dst* are linked by multiple edges, an arbitrary one will be chosen! It is much safer to rely on **name** attributes in the dotfile.
During animation, the given states and edges will be colored as indicated by the −**c** *color* command preceding it (reading from left to right). The initial color is red. An initial −**n** command is implied and may be omitted. Nodes and edges that are not mentioned in a command will have their initial color, as specified in the dotfile. If a node or edge is mentioned multiple times on a single command line, it will be colored according to the color specified for its last (rightmost) occurence.

BUTTONS

At the bottom of an anifsm window there are several buttons. The **Zoom** up and down arrow buttons zoom out resp. in. When zooming, the font size is adjusted accordingly. When the fontsize becomes too small to be usable, only the nodes and edges are drawn and the node and edge labels are omitted. When, as a consequence of zooming in, the fontsize increases again sufficiently, the node and edge labels are shown again.
The **Fit** button zooms the animation to fit in the window. If the **Auto** toggle button is enabled, the animation is automatically zoomed to fit the window when the window is resized.
If the **Render** toggle button is enabled (which is the default) dot layout commands read from standard input have immediate effect. Otherwise, the layout displayed will not be updated until the **Render** toggle button

is enabled again.
The *step number* of the animation step in the trace is shown in the **Step** field. Step numbers start at 0, for the initial state. To visualize the animation step for a known *step*, enter the step number in the **Step** entry field, and hit the return key. If a step number is present in the **Step** field, the down and up arrow buttons can be used to step backwards resp. forwards in the animation.
The **Reuse** toggle button indicates that its window may be reused for a new dotfile, when end-of-input has been seen for the dotfile currently displayed in it. While an animation is in progress (so, when end-of-input has not yet been seen) the **Reuse** button is disabled. (default value: unset, except when overridden by a **−r** command line option of **anifsm**).
To the right of the **Reuse** button, a menu button displays the current *key* value. The *key* value is used to guide the reuse of the window. Pressing the key menu button pops up a menu that offers the choice between all ‘‘known’’ keys. While an animation is in progress (so, when end-of-input has not yet been seen) the key menu button is disabled. (default value: empty, except when overridden by a **−k** *key* command line option of **anifsm).**
The **Clone** button creates a new anifsm window, showing the same animated dot file. As described above the canvas of the clone(s) can be made to scroll simultaneously to show the same item.
The **Close** button closes the window, and, if this was the last remaining anifsm window, exits the **anifsmsrv** program.
The **Quit** button closes all anifsm windows and exits the **anifsmsrv** program.

EDITING

The mouse can be used to edit the graph in an **anifsm** window. The left mouse button is used to create nodes and edges, the right mouse button is used to delete them, to edit their attributes, and to post a pop-up menu.

Clicking the left mouse button on the background of the canvas creates a new node.

Pressing the left mouse button on (in) a node and (while keeping the mouse button pressed) moving the mouse slightly starts the creation of an edge, as indicated by the red arrow which then appears. If the left mouse button is then released with the mouse cursor on (in) a node, an edge is created from the originating node to the destination node. If these nodes are the same a self-loop is created. If the mouse was released while the mouse cursor was not in a node, no edge is created (this can be used to cancel the creation of an edge). Just clicking the left mouse button on a node without moving the mouse at all does nothing (to avoid having to remove lots of unwanted self-loops).

Pressing the right mouse button on a node or edge pops up an attribute edit box, which allows editing of node or edge attributes, and deletion of the node or edge. The top part of the edit box contains attribute names (on the left) with their current values (on the right). The values that can be changed appear in entry fields. To change a value, edit it in the entry field, and press the return key to ’commit’ the change. To add an attribute not yet present, enter its name in the empty entry field on the left (under the ’known’ attribute names), and its value in its corresponding entry field on the right, and press the return key.
At the bottom of the edit box there are two buttons: **Delete**, which deletes the node or edge from the graph, and **Dismiss**, which makes the edit box disappear.

Pressing the right mouse button on a the background of the canvas pops up a menu that contains commands to create a new (inititially empty) window, to open (read) a dot file, to show some information, to connect to an mcast session, to write the graph in the window in dot, Aldebaran (.aut), or postscript format, and to pop up an edit box to edit global graph, node or edge attributes.

AUTOMATON

The graph in the window can be interpreted as an automaton. Special node attributes are used to indicate the start state, and to indicate those graph nodes that are not part of the automaton (those graph nodes will not be present in the automaton written in Aldebaran (.aut) format). Initial values of these attributes are set when the dot file is read, or, if no dot file was given, when the graph is constructed.

autstart when set to 1, indicates that this node is to be the start state of the (Aldebaran, .aut) automaton represented by the graph. At most one node in the graph can have this attribute with a non-zero and/or non-empty value.

autexclude when set to 1, indicates that this node should not appear in the (Aldebaran, .aut) automaton. When reading an automaton or graph from file, nodes of which either the node name or the value of the **label** attribute starts with an underscore will have this attribute set to 1.

When a graph is read or constructed, the start state will be determined as follows. Initially, the first node created in a graph will be the start state. Then, the nodes are inspected and where applicable **autexclude** attributes will be set. Finally the edges are inspected. If there is an edge from an aut-excluded node to a non-aut-excluded node, the non-aut-excluded will be the start state of the automaton. If there are multiple such edges, the 'last' one 'wins'.

When the Aldebaran (.aut) file is written, the transition names are taken from the label attributes of the edges. The state names are determined as follows. If a graph node has a label attribute, it is used to determine the state name, otherwise the node name is used. If all such names (of all non-autexcluded graph nodes) consist of the same string prefix followed by a number, the numbers are used as state numbers in the Aldebaran file. Otherwise the complete names are used (which make them just be numbers).

ATTRIBUTES

A number of node and edge attributes have direct effect on the appearance of the graph in **anifsm** (even though they (mostly) have no effect on the automaton). The definitive reference for these is the **dot (1)** manual page; we only list a number of them here for convenience.

In addition to the attribute names understood by dot there are a few node and edge attribute names that **dot** does not care about, but that are special for **anifsm**.

Attributes that have no value yet (that have the empty value) appear with **{}** as value in the edit box.

DOT NODE ATTRIBUTES

label the text that appears in a node. The value **{\N}** is special: it indicates that the node name should be used as label.

color the color of the node

fillcolor the fill color of the node, if its **style** is set to **filled** (if **fillcolor** is not set the value of **color** will be used)

style for example, filled

shape the shape of the node.

DOT EDGE ATTRIBUTES

label the text that appears with an edge.

color the color of the edge and its accompanying text

ANIFSM NODE ATTRIBUTES

autstart when set to 1, indicates that this node is to be the start state of the (Aldebaran, .aut) automaton represented by the graph. At most one node in the graph can have this attribute with a non-zero and/or non-empty value.

autexclude when set to 1, indicates that this node should not appear in the (Aldebaran, .aut) automaton.

subgraph gives the name of the subgraph to which the node belongs.

ANIFSM EDGE ATTRIBUTES

name identifies the edge. This name can be used, for example, during animation to hightlight the edge. The same name may be assigned to multiple edges (which all will be highlighted when the name is used in an animation command).

EXAMPLES

Valid commands are:

S0
−n S0

−c red −n S0
−c #f00 −n S0

to color node S0 red; the commands are equivalent, the first uses the defaults. The last shows that in addition to color names also the #rgb color specifications of tcl/tk can be used.

−c green S0

to color node S0 green;

−c green S0 S1 −e e0 −c blue e1 −n S2 S3 S4 −e e2

to color nodes S0 and S1 and edge e0 green and nodes S2, S3 and S4 and edges e1 and e2 blue;

−e e0

to color edge e0 red.

Example of mix of dot layout and animation commands (note that we do not have to use **dot node** commands if we do not need to specify attributes for the nodes, and we do not have multiple nodes with the same name). We specify edges between nodes **a**, **b**, and **c**, with **name** attributes that we use in subsequent animation commands, and for the edge from **a** to **c** we specify a ''backwards'' direction, equivalent to **dir=back** in a dot file. We then specify some additional edges, without name attribute, so we use the *src*−>*dst* notation to refer to them in the subsequent animation commands.

```
dot edge a−>b name e0
dot edge b−>c name e1
dot edge a−>c name e2 dir back
−e e0
−e e1
−e e2
dot edge a−>d
dot edge d−>c
−e a−>d
−e d−>c
−e e2
```

Example of the creation of subgraphs. With the **subgraph** definition and **subgraph** attribute in the definition of both nodes **a** and **d** both nodes will be at the top, having the same rank. Without the subgraph definition node **d** would be next to node **b**.

```
dot subgraph g1 rank same
dot node a subgraph g1
dot node b
dot node c
dot node d subgraph g1
dot edge a−>b
dot edge b−>c
dot edge d−>c
```

The three dot edge lines above can be combined into two:

```
dot edge a−>b−>c
dot edge d−>c
```

DIAGNOSTICS

Error messages and navigation diagnostics appear on standard error.

BUGS

The environment variable **TORX_ROOT** is not supported.

Because the animation commands are read from standard input, it is not possible to read the *dotfile* from standard input. However, the (new) dot layout commands that can be given on standard input compensate for that to a certain extent.

The window does not automatically scroll to follow the colored states.

After an syntax error has been encountered in an input dotfile, **tcldot**(1) (at least the version in GraphViz 1.8.5) seems to be unable to recover sufficiently to be able to read more (syntactly correct) dotfiles.

After the revision of the syntax of the commands accepted on standard input to make it more general and take out the TorX specific features, the language accepted by **anifsm** differs from the one accepted by **jararacy**(1). The difference between the two is bridged by **jararacy2anifsm**(1). However, the fact that we have this difference breaks the fall-back to **jararacy**(1) that used to be present in **anifsm**, because if this fall-back would be used, **jararacy**(1) would be given the revised commands which it will not understand. As a consequence, the fall-back has been removed: if **anifsm** can not find tcl package Tcldot it will just give up.

The usage of **autstart** and **autexclude** attributes to indicate automaton features of the graph is clumsy.

When the attribute edit box is popped up, it tries to position itself under the mouse cursor, in an attempt to reduce the necessary mouse movements. We added this in the hope that it would be benificial, but it works not as benificial as intended: the fact that the box is moving all the time is very annoying. To be fixed.

There is no indication that a graph that was read in from a dot file has been altered using mouse commands.

There are no distinct view and edit modes (editing is always enabled). There is not enough experience (yet) with **anifsm** to tell whether such modes are actually needed, though.

SEE ALSO

torx-intro(1), **jararaca**(1), **jararacy2anifsm**(1), **log2jararacy**(1), **dot**(1), **doted**(1), **tcldot**(1), **jararacy**(1), **torx-logclient**(1), **tmcs**(1), **anifsm**(1), **aniwait**(1), **mscviewer**(1), **environ**(5)

ACKNOWLEDGEMENTS

Parts of **anifsm** (in particular the zooming and graph editing code) have been lifted and adapted from:
doted - dot graph editor - John Ellson (ellson@graphviz.org)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of torx.

NAME

aniwait − animate progressbar

SYNOPSIS

aniwait [-r] [-m *mcastid* **] [-t** *title* **]**
aniwaitsrv
aniwait -exit

DESCRIPTION

aniwait ‘animates’ a progressbar. After start up, **aniwait** creates a window (with the given *title*) containing a progressbar, and waits for animation commands on standard input. On end of file on standard input, **aniwait** waits for the user to remove the window (or press the **Quit** button), after which it exits.

Actually, **aniwait** is a shell-level command that uses a running **aniwaitsrv** to create or reuse a progressbar window, and animate it using animation commands on standard input of **aniwait.** The connection between **aniwait** and a running **aniwaitsrv** will not be closed until the complete standard input of the **aniwait** command has been processed by **aniwaitsrv**. If **aniwait** cannot find a running **aniwaitsrv**, it will start a new one. In general, it should not be necessary to start **aniwaitsrv** by hand. However, if startup time of **aniwait** is an issue, it may be advantageous to start **aniwaitsrv** (by hand) in advance, because a starting **aniwaitsrv** may spend some time to check if another **aniwaitsrv** is already running.

To display a new progressbar, **aniwaitsrv** will reuse windows that contain a completed animation and have the **Reuse** toggle activated. If more windows are needed, they are created.

The **-r** command line option of **aniwait** will activate the **Reuse** toggle button for the aniwait window.

The animation commands are expected to be generated using **log2aniwait**(1), e.g. using a unix command as

log2aniwait < *logfile* **| aniwait**

or

tail -f *logfile* **| log2aniwait | aniwait**

Each animation command consists of a single line of text, of the following form:

wait *count*
freeze [*remains* **]**
stop [*remains* **]**

where *count* and *remains* are floating point numbers. The **wait** command starts a countdown of the given *count* number of seconds. The **freeze** and **stop** commands stop the countdown, and add a ‘step’ to the trace of progress times. **freeze** and **stop** interpret the optional *remains* as the time remaining from the *count* from the **countdown** command; if no *remains* is given, the real-time system clock is used. An additional **freeze** or **stop** command without preceding **wait** command has no effect and is ignored. The difference between **freeze** and **stop** is in the color of the progressbar: **freeze** does not change the color, but only ‘freezes’ the animation, whereas **stop** changes the color of the progressbar to blue.

The animation in the window will follow the animation commands read from standard input. The animation can be be done manually using the left and middle mouse button, and/or with the **Step** up and down arrow buttons (as discussed below).

In addition, the animation can be remotely controlled. If the **-m** *mcastid* command line option is given, or environment variable **TORXMCASTID** was set, **aniwait** will attempt to make a remote control connection to the tcp address in *mcastid*. If it succeeds, it will then interpret lines of text read from the remote control connection consisting of a single number as commands to show the corresponding step in the animation. Additionally, whenever the user uses mouse button and/or navigation commands to show a different step, its step number is written to the remote control connection. The remote control connection allows multiple viewers to show the same test step.

The left mouse button and the right mouse button can be used to ‘‘navigate’’ in the animation: the left mouse button will show the ‘‘next’’ step in the animation, and the right mouse button will show the ‘‘previous’’ step in the animation.

To stop a running **aniwaitsrv**, invoke **aniwait** with the **-exit** command line option.

BUTTONS

At the bottom of an aniwait window there are several buttons. The *step number* of the animation step in the trace is shown in the **Step** field. Step numbers start at 0, for the initial state. To visualize the animation step for a known *step*, enter the step number in the **Step** entry field, and hit the return key. If a step number is present in the **Step** field, the down and up arrow buttons can be used to step backwards resp. forwards in the animation.

The **Reuse** toggle button indicates that its window may be reused for a new animation, when end-of-input has been seen for the animation currently displayed in it. While an animation is in progress (so, when end-of-input has not yet been seen) the **Reuse** button is disabled. (default value: unset, except when overridden by a **-r** command line option of **aniwait**).

The **Close** button closes the window, and, if this was the last remaining aniwait window, exits the progam.

The **Quit** button closes all aniwait windows and exits the progam.

DIAGNOSTICS

Error messages and navigation diagnostics appear on standard error.

BUGS

The environment variable **TORX_ROOT** is not supported.

SEE ALSO

torx-intro(1), **log2aniwait**(1), **torx-logclient**(1), **tmcs**(1), **jararacy**(1), **anifsm**(1), **mscviewer**(1), **environ**(5)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of torx.

NAME

aniwait − animate progressbar

SYNOPSIS

aniwait [-r] [-m *mcastid* **] [-t** *title* **]**
aniwaitsrv
aniwait -exit

DESCRIPTION

aniwait ‘animates’ a progressbar. After start up, **aniwait** creates a window (with the given *title*) containing a progressbar, and waits for animation commands on standard input. On end of file on standard input, **aniwait** waits for the user to remove the window (or press the **Quit** button), after which it exits.

Actually, **aniwait** is a shell-level command that uses a running **aniwaitsrv** to create or reuse a progressbar window, and animate it using animation commands on standard input of **aniwait.** The connection between **aniwait** and a running **aniwaitsrv** will not be closed until the complete standard input of the **aniwait** command has been processed by **aniwaitsrv**. If **aniwait** cannot find a running **aniwaitsrv**, it will start a new one. In general, it should not be necessary to start **aniwaitsrv** by hand. However, if startup time of **aniwait** is an issue, it may be advantageous to start **aniwaitsrv** (by hand) in advance, because a starting **aniwaitsrv** may spend some time to check if another **aniwaitsrv** is already running.

To display a new progressbar, **aniwaitsrv** will reuse windows that contain a completed animation and have the **Reuse** toggle activated. If more windows are needed, they are created.

The **-r** command line option of **aniwait** will activate the **Reuse** toggle button for the aniwait window.

The animation commands are expected to be generated using **log2aniwait**(1), e.g. using a unix command as

log2aniwait < *logfile* **| aniwait**

or

tail -f *logfile* **| log2aniwait | aniwait**

Each animation command consists of a single line of text, of the following form:

wait *count*
freeze [*remains* **]**
stop [*remains* **]**

where *count* and *remains* are floating point numbers. The **wait** command starts a countdown of the given *count* number of seconds. The **freeze** and **stop** commands stop the countdown, and add a ‘step’ to the trace of progress times. **freeze** and **stop** interpret the optional *remains* as the time remaining from the *count* from the **countdown** command; if no *remains* is given, the real-time system clock is used. An additional **freeze** or **stop** command without preceding **wait** command has no effect and is ignored. The difference between **freeze** and **stop** is in the color of the progressbar: **freeze** does not change the color, but only ‘freezes’ the animation, whereas **stop** changes the color of the progressbar to blue.

The animation in the window will follow the animation commands read from standard input. The animation can be be done manually using the left and middle mouse button, and/or with the **Step** up and down arrow buttons (as discussed below).

In addition, the animation can be remotely controlled. If the **-m** *mcastid* command line option is given, or environment variable **TORXMCASTID** was set, **aniwait** will attempt to make a remote control connection to the tcp address in *mcastid*. If it succeeds, it will then interpret lines of text read from the remote control connection consisting of a single number as commands to show the corresponding step in the animation. Additionally, whenever the user uses mouse button and/or navigation commands to show a different step, its step number is written to the remote control connection. The remote control connection allows multiple viewers to show the same test step.

The left mouse button and the right mouse button can be used to ‘‘navigate’’ in the animation: the left mouse button will show the ‘‘next’’ step in the animation, and the right mouse button will show the ‘‘previous’’ step in the animation.

To stop a running **aniwaitsrv**, invoke **aniwait** with the **-exit** command line option.

BUTTONS

At the bottom of an aniwait window there are several buttons. The *step number* of the animation step in the trace is shown in the **Step** field. Step numbers start at 0, for the initial state. To visualize the animation step for a known *step*, enter the step number in the **Step** entry field, and hit the return key. If a step number is present in the **Step** field, the down and up arrow buttons can be used to step backwards resp. forwards in the animation.

The **Reuse** toggle button indicates that its window may be reused for a new animation, when end-of-input has been seen for the animation currently displayed in it. While an animation is in progress (so, when end-of-input has not yet been seen) the **Reuse** button is disabled. (default value: unset, except when overridden by a **-r** command line option of **aniwait**).

The **Close** button closes the window, and, if this was the last remaining aniwait window, exits the progam.

The **Quit** button closes all aniwait windows and exits the progam.

DIAGNOSTICS

Error messages and navigation diagnostics appear on standard error.

BUGS

The environment variable **TORX_ROOT** is not supported.

SEE ALSO

torx-intro(1), **log2aniwait**(1), **torx-logclient**(1), **tmcs**(1), **jararacy**(1), **anifsm**(1), **mscviewer**(1), **environ**(5)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of torx.

NAME

aut2fsmview − translate Aldebaran (.aut) to FSMView input

SYNOPSIS

aut2fsmview

DESCRIPTION

aut2fsmview reads an Aldebaran (.aut) file from standard input and writes corresponding input for FSMView on standard output. FSMView is a tool for interactive visualization of state transition systems.

The state information in the generated FSMView input only contains fan-in and fan-out.

If we use this to visualize a test run (so the Aldebaran (.aut) file is generated from a **torx-log**(4) file using **log2aut**(1)) then it could be interesting to generate the FSMView input directly from the **torx-log**(4) file and include more information from it like statistics about the state space exploration.

SEE ALSO

torx-intro(1), **autexp**(1), **log2aut**(1), **torx-log**(4)

FSMView home page: http://www.win.tue.nl/~fvham/fsm/ (papers, FSMView download for windows and linux)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of torx.

NAME

autexp − explore Aldebaran (.aut) automaton files

SYNOPSIS

autexp [-d | -m] *aut-file*

DESCRIPTION

autexp implements an explorer for simple automatons in the Aldebaran (.aut) file format from the Caesar/Aldebaran Development Package (CADP). It offers the TorX **torx-explorer**(5) interface on standard input and output.

When invoked with the **-d** flag, **autexp** writes a **dot**(1) representation of the automaton to standard output, and exits. The dot node names correspond to the state numbers of the automaton, and with each dot edge (depicting a transition of the automaton) a dot attribute **name** is associated, which is given a unique value.

When invoked with the **-m** flag, **autexp** writes the connection matrix of the automaton to standard output, and exits.

The **dot**(1) output can be used for animation, because **autexp** encodes the dot state and edge (transition) names in the state identifiers that it uses in the **torx-explorer**(5) interface. These identifiers can be extracted from a **torx-log**(5) file using **log2jararacy**(1) such that a trace of a run of **autexp** can be animated using **anifsm**(1) and **jararacy**(1).
This is the main advantage of using **autexp** over using the equivalent Aldebaran (.aut) file explorer availabe via the CADP package (see **mkprimer-cadp**(1)).

BUGS

The environment variable **TORX_ROOT** is not supported.

SEE ALSO

torx-intro(1), **anifsm**(1), **dot**(1), **log2jararacy**(1), **jararacy**(1), **jararacy2anifsm**(1), **mkprimer-cadp**(1), **torx-explorer**(5), **environ**(5)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of torx.

NAME

autsimplify − simplify automaton Aldebaran (.aut) file

SYNOPSIS

autsimplify

DESCRIPTION

autsimplify reads an Aldebaran (.aut) format file from standard input, simplifies the automaton as described below, and writes the resulting automaton in Aldebaran (.aut) format on standard output.

autsimplify is an experiment at reducing the size of an automaton without changing it structure, by reducing chains of transitions by a single transition with a single label obtained by concatenating all individual labels. The hope is that this makes it easier for dot (or for **anifsm**(1)) to compute a layout for the graph.
Initial experiments seem to suggest that the effect may be limited. It may help to remove the label text from the result of **autsimplify**, such that only the structure of the graph remains, but even that may not be sufficient to allow dot to efficiently compute the layout of a big graph.

BUGS

autsimplify is just an experiment, it needs more experimentation.

SEE ALSO

torx-intro(1), **autexp**(1), **anifsm**(1)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of torx.

NAME

campaign − generate and populate TorX test campaign directory structure

SYNOPSIS

campaign *configfile*

DESCRIPTION

campaign is an *experimental* command that generates and populates a TorX test campaign directory structure. The *configfile* contains one or more TorX test run configurations, i.e. configurations specifying the specification, the (access to the) implementation, the TorX tool components to use, their parameters, seeds for random number generaters, etc. The configuration file is structured as a list of blocks of *name=value* pairs. Generally, for each test run that should be executed, the configuration file contains such a block for each instance of a TorX tool concept that ''participates'' in the test run. Examples of TorX tool concepts are the TorX tool components, the Implementation Under Test (IUT), the connections to the IUT, like the Points of Control and Observation (PCOs), and Implementation Access Points (IAPs). Each block has a type and a (unique) name, which together form the first name=value *type=name* pair of each block.

The names are used to ''link'' blocks, in the form of a DAG (directed acyclic graph). The ''root'' of the DAG is formed by a block of type **campaign**. The ''links'' are formed by name-value pairs in a block that refer to the names of other blocks, instead of specifying configuration parameters.

A particular ''link'' is made by the **base** field: it inherits the definition named in the field's value, provided that this definition is of the same type as the block that contains the **base** field.

To reduce the size of the configuration, a very primitive variable mechanism can be used to ''simulate'' parameterized blocks, and a primitive ''foreach'' construct can be used to create multiple instances of a configuration (with different variable instantiations).

TEST CAMPAIGNS

Here we discuss test campaigns, the things that can be described in the campaign configuration language. Part of this section also tries to clarify the design of the campaign language.

STRUCTURE

We see test campaigns as hierarchies constisting of three ''levels''.

At the top we have the ''campaign'', that consists of a set of ''executions''. Each ''execution'' consists of a single experiment, where all parameters under control of the test operator are fixed. So, the configuration of an ''execution'' describes the test architecture, the tools, the parameters of the tools etc. The only remaining ''parameter'' is the non-determinism of the implementation. To cope with that, we allow each ''execution'' to be run more than once, which gives us that each ''execution'' is (has?) a set of ''runs''.

COMPONENTS

The ingredients of a test campaign is formed by the execution architecture (what components do we have, and how are they connected) and its parameters.

The execution architecture is effectively identical to the test architecture (how are tester and iut connected), with the addition of information about the internal structure of the tester (what components are used, how are they connected).

In the test architecture we have the concepts ''Implementation Under Test'' (IUT), ''Implementation Access Point'' (IAP), ''Point of Control and Observation'' (PCO), ''Test Context'' and ''System Under Test''. In our view, the Adapter can only reach, access, communicate with, the IUT via the Test Context. The PCOs form the connection between the Adapter and the Test Context, and the IAPs form the connection between IUT and the Test Context. For the execution architecture we are not interested in the Test Context and the SUT, but we are interested in all the other concepts. In addition, we are interested in the concepts from the tool architecture: explorer, primer, combinator, test purpose, (batch) test, instantiator, driver, and adapter.

The IUT has one or more IAP's, and there are one or more PCO's. A PCO may coincide with an IAP, or there may be a test-context separating them, in which case there may be a many-to-many mapping between IAP's and PCO's. We currently assume that in the execution architecture we (may) have: one IUT, one adapter, one driver, zero or more explorers, zero or more primers, zero or more test purposes, zero or more

combinators, zero or more batch tests, zero or more instantiators.

CONNECTIONS

The IUT is connected (only) to the adapter, directly over IAP's or indirectly via PCO's and IAP's. The adapter is directly connected with the driver. The driver is has also connected either a primer, or a combinator, or a batch-test-primer, or an instantiator. The primer and batch-test-primer may have an explorer connected, or the explorer may be integrated in them. The combinator can have primers, test purposes and other combinators connected. Somehow, we should indicate for each combinator that is connected to another combinator how it should be treated (as a primer or as a test-purpose?). An indirect way to do that would be to associate with each combinator a role (the role that it wants to play for the driver or other combinator that it is connected to) or more indirectly, to specify for each combinator the operation that it is supposed to perform on its inputs, which then includes the ''interpretation'' that should be given to these inputs (trivial for primer and test-purpose inputs, more interesting for a combinator input).

GENERATED DIRECTOR STRUCTURE

campaign generates from its configuration a multi-level directory structure. The top-level one contains the generated top-level Makefile, and (optionally via intermediate sub-directories, as specified in the **dir** field of each **experiment** specified in the test campaign) a number of ''experiment'' directories, one for each experiment specified in the configuration as part of the test campaign. Each of the ''experiment'' directories contains a generated Makefile for the experiment, and generated configuration files for the TorX tool components. The top-level Makefile recursively invokes the Makefiles in the ''experiment'' directories to execute the specified test campaign.

CONFIGURATION LANGUAGE

We describe the language by the block types, and give for each type the recognized *name=value* fields (also known as attributes).

Not all fields of all blocks have to be given *(we should mark the optional ones)*. The fields that may occur more than once in a block have a suffix ''∗'' in the list below.

Note that the special field **base** may appear in every kind of block. **base=***value* inherits the definition named in *value*, provided that that definition is of the same type as the block that contains the **base** field. Effectively, a block definition that starts in the following way:

type=name
base=*anothername*
...

will try to ''in-place insert'' the contents of block *type=anothername* at the start of block *type=name*. Examples of the usage of **base** can be found in the EXAMPLES section below.

The block descriptions are in alphabetical order, as are the field descriptions in each block.

adapter

The value of the **adapter** name is just a name that is used to refer to the **adapter** definition from other configuration entries.

codingdir
: the directory containing the coding library that will be used during test execution. In particular, this directory should contain implementations for the functions named in the **multiplexer**, **encoder** and **decoder** fields of the pco's.

exec name of the program to execute when the adapter has to be started

execdir directory in which the program named in the **exec** field has to be run

execparams∗
: arguments for the program named in the **exec** field. There may be multiple **execparams** fields: we need one for each argument.

pco∗ reference to definition elsewhere in the configuration

address The value of the **address** name is used as a reference from within other blocks, and additionally it

is (can be) used as a PIXIT parameter in en/decoding rules and/or connectors.

value actual address. For network addressing we use the (plan9 derived) syntax giving (exclamation-mark separated) *network*, *node* and *port* as *network*!*node*!*port*. Currently known networks are **pipe** (just a single pipe, no node or port needed -- used when the implementation is started by the driver), and **tcp** and **udp**, where the nodename ''∗'' refers to the local host. A port number may be omitted, which means that it can be chosen by the Operating System (or by the tool).

campaign

The value of the **campaign** name indicates the name of the campaign, which is not used. The **campaign** block is the ''root'' of the hierarchy of blocks in the configuration.

comment
to be used for documentation, not used by the tools

dir the root of the campaing directory. All execution directories should be inside the campaign directory. The value of the **dir** field is available through the $campaign variable.

experiment∗
reference to definition elsewhere in the configuration: an experiment that is part of the campaign

makefile
name of the top-level Makefile generated by **campaign**.

mkinclude
name of Makefile that is to be included in the top-level Makefile generated by **campaign**.

channel

The value of the **channel** name indicates the name of the channel.

iokind the type (kind) of the channel, which must be either **input** or **output**

pco∗ reference to definition elsewhere in the configuration: a pco''connected'' to this channel

sevent the event representing ''suspension'' or ''quiescense'' (usally this will be **Delta**) *not implemented yet*

timeout time out value for the channel (only for channels with **iokind**=**output**)

combinator

The value of the **combinator** name is just a name that is used to refer to the **combinator** definition from other configuration entries.

combinator∗
reference to definition elsewhere in the configuration

config name of the configuration file that has to be generated by **campaign** for the combinator described in this block.

exec name of the program to execute when the primer has to be started

execdir directory in which the program named in the **exec** field has to be run

execparams∗
arguments for the program named in the **exec** field. There may be multiple **execparams** fields: we need one for each argument.

gen the program that can build (or generate) the combinator

genparams∗
an argument for the program named in the **gen** field. There may be multiple **genparams** fields: we need one for each argument.

partitioner∗

reference to definition elsewhere in the configuration

primer∗
reference to definition elsewhere in the configuration

test∗ reference to definition elsewhere in the configuration *not implemented yet!*

tp∗ reference to definition elsewhere in the configuration *not implemented yet!*

driver The value of the **driver** name is just a name that is used to refer to the **driver** definition from other configuration entries. (usually the value will be torx)

exec name of the program to execute when the driver has to be started

execparams∗
arguments for the program named in the **exec** field. There may be multiple **execparams** fields: we need one for each argument.

post program that has to be run after the driver has finished *(not yet implemented)*

pre program that has to be run before the driver is started *(not yet implemented)*

experiment
The value of the **experiment** name is just a name that is used to refer to the experiment definition from other configuration entries. The value of the **experiment** field is available in this block (and in blocks refered by it) through the $experiment variable.

adapter
reference to definition elsewhere in the configuration

combinator
reference to definition elsewhere in the configuration

config name of the configuration file that has to be generated by **campaign** for the experiment described in this block. Usually this will be in the directory indicated in the **dir** field.

dir the directory in which the execution should take place

driver reference to definition elsewhere in the configuration

driverparams∗
additional flags for the driver program

impl reference to definition elsewhere in the configuration

log name of file in which to store the execution log (which includes the execution trace)

makefile
name of the Makefile that has to be generated by **campaign** for the experiment described in this block. Usually this will be in the directory indicated in the **dir** field.

maxdepth
the maximum number of steps that will be executed in a test run for this experiment

msg name of file in which to store the (stderr) messages produced by the various components during execution

mkinclude
name of Makefile that is to be included in the Makefile generated by **campaign**.

mutant name of the implementation mutant tu run. This is used to define the **MUTANT** entry in the generated configuration file.

partitioner
reference to definition elsewhere in the configuration

post program that has to be run at the end of the test execution run (What are the default parameters for this program?)

postparams∗
additional arguments for the program given in the **post** field

pre program that has to be run at the start of the test execution run (What are the default parameters for this program?)

preparams∗
additional arguments for the program given in the **pre** field

primer reference to definition elsewhere in the configuration

runs number of execution runs that will be executed for this experiment

seed the seed parameter to be used during test execution

foreach The **foreach** blocks define (typed) variables that can be used to create multiple instantiations of a configuration. The **foreach** definition can be ''activated'' in a **product** block by adding the appropriate **foreach** entry to it. The value of the **foreach** name is just a name that is used to refer to the **foreach** definition from other configuration entries.

name the name of the variable

type the type of the variable. Usually this will be something like *block***.***field*

value∗ one of the values of the variable over which will be iterated

iap The value of the **iap** name indicates the name of the iap. It will be referred to from **impl** blocks.

address reference to definition elsewhere in the configuration. Currently we assume a single address for each iap.

impl The value of the **impl** name is just a name that is used to refer to the **impl** definition from other configuration entries.

configgen
program that is able to generate a configuration file for the implementation, based on the configuration file together with parameters that are only known at run-time (e.g. port numbers chosen dynamically). *(not yet implemented)*

configgenparams∗
arguments for the program named in the **configgen** field. There may be multiple **configgenparams** fields: we need one for each argument. *(not yet implemented)*

exec name of the program to execute when the implementation has to be started

execcontext
program that is used as a filter between the implementation and the adapter. Such a filter can be used e.g. to translate between binary i/o done by the implementation and a hex encoding of it that is more pleasant for the adapter.

execcontextparams∗
arguments for the program named in the **execcontext** field. There may be multiple **execcontextparams** fields: we need one for each argument.

execdir directory in which the program named in the **exec** field has to be run

execparams∗
arguments for the program named in the **exec** field. There may be multiple **execparams** fields: we need one for each argument.

iap∗ reference to definition elsewhere in the configuration. This may contain information that is needed by the adapter, like port addresses at the implementation side of the test context.

post program that has to be run after the implemention has finished

postparams∗

additional arguments for the program given in the **post** field

pre program that has to be run before the implemention is started

preparams*
additional arguments for the program given in the **pre** field

partitioner
The value of the **partitioner** name is just a name that is used to refer to the **partitioner** definition from other configuration entries.

config name of the configuration file that has to be generated by **campaign** for the partitioner described in this block.

exec name of the program to execute when the partitioner has to be started

execdir directory in which the program named in the **exec** field has to be run

execparams*
arguments for the program named in the **exec** field. There may be multiple **execparams** fields: we need one for each argument.

gen the program that can build (or generate) the combinator

genparams*
an argument for the program named in the **gen** field. *not implemented yet*

partfile the location of the partition configuration file (that associates the weights with the actions)

pco The value of the **pco** name indicates the name of the pco. It will be referred to from **channel** and **adapter** blocks (for the channels, we probably should check that there at most two references to each pco, one from an **input** channel and one from an **output** one).

address reference to definition elsewhere in the configuration. Currently we assume a single address for each pco.

decoder
name of the decoding function that is used to decode values that are received via this pco. This function must be present in the library indicated by the **codingdir** field of the **adapter**. In the future we will not need this function, but instead use patterns over the event (if necessary enhanced with predicates).

encoder
name of the encoding function that is used to encode values that are sent over this pco. This function must be present in the library indicated by the **codingdir** field of the **adapter**. In the future we will not need this function, but instead use patterns over the event (if necessary enhanced with predicates).

ievent a pattern over the events of the specification, that is used to partition those events in input and output pco's. This pattern indicates an input event. For backwords compatibility we also allow the pattern to consist of just a single gate name, together with the specification of a **multiplexer** function that will partition events on the same gate.

multiplexer
name of the function that is used to map an event to a pco. This function must be present in the library indicated by the **codingdir** field of the **adapter**. In the future we will not need this function, but instead use patterns over the event (if necessary enhanced with predicates).

oevent a pattern over the events of the specification, that is used to partition those events in input and output pco's. This pattern indicates an output event. For backwords compatibility we also allow the pattern to consist of just a single gate name, together with the specification of a **multiplexer** function that will partition events on the same gate.

regexp the value is exported to the decoding function, where it may be used to segment stream-like data received from the SUT

primer The value of the **primer** name is just a name that is used to refer to the **primer** definition from other configuration entries.

channel∗
reference to definition elsewhere in the configuration. The channel definitions define the channels, the subset of the labels that they represent, and whether it is input or output.

exec name of the program to execute when the primer has to be started

execdir directory in which the program named in the **exec** field has to be run

execparams∗
arguments for the program named in the **exec** field. There may be multiple **execparams** fields: we need one for each argument.

gen the program that can build (or generate) the primer

genparams∗
an argument for the program named in the **gen** field. There may be multiple **genparams** fields: we need one for each argument.

spec reference to definition elsewhere in the configuration

product
The **product** blocks define multiple instantiations of a given template. The instantiations are generated as the cross product of the values of the product variables, as defined in **foreach** fields. The value of the **product** name is just a name that is used to refer to the **product** definition from other configuration entries.

foreach∗
the variable definitions

prefix the prefix of the names of the resulting instantiations. Their names will consist of the prefix, followed by for each foreach clause a hyphen followed by the value of the variable. So, ‘‘product=lotosmutants’’ in the example below, generates names like ‘‘lotos-mutants-000-3’’ (first the prefix, followed by a hyphen and a mutant value, followed by a hyphen and a seed value).

template
a reference to the block that should be instantiated. It should be of the type given in the **type** field.

type the type of the result, which should be identical to the type of the given template. This will be something like *block*

spec The value of the **spec** name is just a name that is used to refer to the **spec** definition from other configuration entries.

auxfile∗
the location of an auxiliary specification file. Currently they are used for user-supplied ADT implementation files (with .t and .f file name suffixes) that may be needed by CADP (via **mkprimer**(1)) to generate a Primer program from a LOTOS specification.

dialect (optionally) describes tool dialect (e.g. to distinguish between LOTOS specs for lite and for CADP) (so far only used for documentation, not used by the **campaign** tool)

file the location of the (main) specification file. (Note: in general a single specification could consist of several files. We can probably handle that by requesting that all files of a specification appear in the same directory, which then can be named here, and use the **gen** field to deal with it).

language
the specification language (only used for documentation, not used by the **campaign** tool)

var The **var** blocks define (typed) variables that can be used to parameterise a configuration. The **var** definition can be ''activated'' in an arbitrary block by adding the appropriate **var** entry to it. The value of the **var** name is just a name that is used to refer to the **var** definition from other configuration entries.

name the name of the variable

type the type of the variable Usually this will be something like *block***.***field*

value∗ the value of the variable. If the *field* given in the **type** field may appear more than once in its block, there may be multiple value fields for the variable definition.

EXAMPLES

FIRST EXAMPLE

```
#=============================================
spec=confprot01l
    file=$campaign/specs/confprot01.lot
    language=LOTOS  dialect=lite
spec=confprot01c
    file=$campaign/specs/confprot01.caesar.lot
    auxfile=$campaign/specs/confprot01.caesar.t
    auxfile=$campaign/specs/confprot01.caesar.f
    language=LOTOS  dialect=cadp
spec=confprot01p
    file=$campaign/specs/conf-solo.trojka
    language=PROMELA

primer=pl
    spec=confprot01c
    gen=mkprimer
    genparams=
    exec=$campaign/specs/confprot01.caesar
    execdir=$campaign/specs
    execparams=
    channel=in
    channel=out
primer=pp
    spec=confprot01p
    gen=mkprimer
    genparams=
    exec=$campaign/specs/conf-solo.sh
    execdir=$campaign/specs
    execparams=
    channel=in
    channel=out

impl=jan
    pre=
    post=
    exec=$campaign/impls/confprot.sh
    execparams=-c
    # execparams=$campaign/executions/$experiment/cfg.txt
    execparams=$campaign/cfg.txt
```

```
        execparams=-DEBUG
        execparams=-1
        execparams=-CSAP
        execparams=-MUTANT
        execparams=v-mutant
        execcontext=hexcontext
        execcontextparams=--
        configgen=
        configgenparams=-o
        configgenparams=$campaign/executions/$experiment/cfg.txt
        iap=up
        iap=low

driver=torx
        configgen=
        exec=torx
        execparams=--log
        execparams=$(log)
        execparams=--seed
        execparams=$(seed)
        execparams=--depth
        execparams=$(maxdepth)
        execparams=$(config)
        pre=
        post=

address=aup     name=v-add0   value=pipe
address=alow1   name=v-add1   value=udp!*!1075
address=alow2   name=v-add2   value=udp!*!1076
address=alow3   name=v-add3   value=udp!*!1077
address=alow4   name=v-add4   value=udp!*!1078

adapter=a       codingdir=v-coding
                    pco=up1  pco=low2 pco=low3 pco=low4

channel=in      iokind=input  pco=up1  pco=low2 pco=low3 pco=low4
channel=out     iokind=output pco=up1  pco=low2 pco=low3 pco=low4  timeout=2

pco=upbase      encoder=enCodingOfCFsp decoder=CFsp_nl2CFsp   regexp={RECVHEX[^0+0
pco=lowbase     encoder=enCodingOfUdp  decoder=udp_nl2udpsp   regexp={RECVHEX[^0+0
            multiplexer=pcoOfUdp
pco=up1         base=upbase   address=aup       ievent=v-iev0 oevent=v-oev0
pco=low2        base=lowbase  address=alow2     ievent=v-iev2 oevent=v-oev2
pco=low3        base=lowbase  address=alow3     ievent=v-iev3 oevent=v-oev3
pco=low4        base=lowbase  address=alow4     ievent=v-iev4 oevent=v-oev4

iap=up          address=aup
iap=low         address=alow1

var=l-u-iev0  name=v-iev0   type=pco.ievent   value=cfsap_in!*!*
var=l-u-oev0  name=v-oev0   type=pco.oevent   value=cfsap_out!*
                                value=cfsap_out!*!*
var=l-l-iev2  name=v-iev2   type=pco.ievent   value=udp_in!udp1!udp_req(udp2,*)
```

```
                                value=udp_in!udp2!*
var=l-l-oev2  name=v-oev2   type=pco.oevent   value=udp_out!udp1!udp_ind(udp2,*)
                                value=udp_out!udp2!*
var=l-l-iev3  name=v-iev3   type=pco.ievent   value=udp_in!udp1!udp_req(udp3,*)
                                value=udp_in!udp3!*
var=l-l-oev3  name=v-oev3   type=pco.oevent   value=udp_out!udp1!udp_ind(udp3,*)
                                value=udp_out!udp3!*
var=l-l-iev4  name=v-iev4   type=pco.ievent   value=udp_in!udp1!udp_req(udp4,*)
                                value=udp_in!udp4!*
var=l-l-oev4  name=v-oev4   type=pco.oevent   value=udp_out!udp1!udp_ind(udp4,*)
                                  value=udp_out!udp4!*

var=p-u-iev0  name=v-iev0   type=pco.ievent   value=from_upper
var=p-u-oev0  name=v-oev0   type=pco.oevent   value=to_upper
var=p-l-iev2  name=v-iev2   type=pco.ievent   value=from_lower
var=p-l-oev2  name=v-oev2   type=pco.oevent   value=to_lower
var=p-l-iev3  name=v-iev3   type=pco.ievent   value=from_lower
var=p-l-oev3  name=v-oev3   type=pco.oevent   value=to_lower
var=p-l-iev4  name=v-iev4   type=pco.ievent   value=from_lower
var=p-l-oev4  name=v-oev4   type=pco.oevent   value=to_lower

var=l-u-add0  name=v-add0   type=address.name value=cf1
var=l-l-add1  name=v-add1   type=address.name value=udp1
var=l-l-add2  name=v-add2   type=address.name value=udp2
var=l-l-add3  name=v-add3   type=address.name value=udp3
var=l-l-add4  name=v-add4   type=address.name value=udp4

var=p-u-add0  name=v-add0   type=address.name value=cf1
var=p-l-add1  name=v-add1   type=address.name value=1
var=p-l-add2  name=v-add2   type=address.name value=0
var=p-l-add3  name=v-add3   type=address.name value=2
var=p-l-add4  name=v-add4   type=address.name value=4

var=l-coding  name=v-coding type=adapter.codingdir
            value=$campaign/coding/LOTOS
var=p-coding  name=v-coding type=adapter.codingdir
            value=$campaign/coding/PROMELA

experiment=defaults
      msg=msg
      log=log
      dir=$campaign/executions/$experiment
      makefile=$campaign/executions/$experiment/torx.mk
      config=$campaign/executions/$experiment/torx.if
      pre=:
      post=:
      driver=torx
experiment=templatedefaults
      base=defaults
      runs=2
      seed=v-seed
      maxdepth=30
      adapter=a
```

```
        impl=jan
    experiment=lotos
        base=templatedefaults
        primer=pl       var=l-coding   var=l-l-add1
        var=l-u-iev0   var=l-u-oev0   var=l-u-add0
        var=l-l-iev2   var=l-l-oev2   var=l-l-add2
        var=l-l-iev3   var=l-l-oev3   var=l-l-add3
        var=l-l-iev4   var=l-l-oev4   var=l-l-add4
    experiment=promela
        base=templatedefaults
        primer=pp       var=p-coding   var=p-l-add1
        var=p-u-iev0   var=p-u-oev0   var=p-u-add0
        var=p-l-iev2   var=p-l-oev2   var=p-l-add2
        var=p-l-iev3   var=p-l-oev3   var=p-l-add3
        var=p-l-iev4   var=p-l-oev4   var=p-l-add4

    foreach=seed
        type=experiment.seed
        name=v-seed
        value=1    value=2    value=3    value=4    value=5
    foreach=mutants
        type=impl.execparams
        name=v-mutant
        value=000   value=100   value=111   value=214   value=247
        value=276   value=289   value=293   value=294   value=332
        value=345   value=348   value=358   value=384   value=398
        value=444   value=462   value=467   value=548   value=666
        value=687   value=738   value=749   value=782   value=836
        value=856   value=945

    product=lotosmutants
        type=experiment
        foreach=mutants
        foreach=seed
        prefix=lotos-mutants
        template=lotos
    product=promelamutants
        type=experiment
        foreach=mutants
        foreach=seed
        prefix=promela-mutants
        template=promela

    campaign=one
        dir=/home/fmg/belinfan/src/cdr/utest_old_release/Examples/CampaignTemplate
        makefile=$campaign/Makefile
        product=lotosmutants
        product=promelamutants
    #============================================
```

SECOND EXAMPLE

```
    # ============================================

    driver=torx
```

```
        configgen=
        exec=torx
        execparams=--log
        execparams=$(log)
        execparams=--seed
        execparams=$(seed)
        execparams=--depth
        execparams=$(maxdepth)
        execparams=$(config)
        pre=
        post=

    spec=LOTOS
        file=$campaign/specs/LOTOS/cf-pe-sut.caesar.lot
        auxfile=$campaign/specs/LOTOS/cf-pe-sut.caesar.t
        auxfile=$campaign/specs/LOTOS/cf-pe-sut.caesar.f
        language=LOTOS
        preproc=
        dialect=cadp

    primer=pl
        spec=LOTOS
        gen=mkprimer
        genparams=
        exec=$campaign/specs/LOTOS/cf-pe-sut.caesar
        execdir=$campaign/specs/LOTOS
        execparams=
        channel=in
        channel=out

    impl=janbase
        pre=
        post=
        exec=$campaign/impls/confprotv3c/confprot.sh
        gen=make
        execdir=$campaign/impls/confprotv3c
        genparams=confprot
        execparams=-a
        execparams=pythagoras:1075
        execparams=-a
        execparams=pythagoras:1076
        execparams=-a
        execparams=pythagoras:1077
        execcontext=hexcontext
        execcontextparams=--
        iap=up
        iap=low

    foreach=mutants
        type=impl.execparams
        type=experiment.mutant
        name=v-nr
        value=001
```

```
        value=002
        value=003
        value=055
        value=056
        value=057
        value=058
        value=059
        value=099

    foreach=seeds
        type=experiment.seed
        name=var-s
        value=789
        value=161
        value=78
        value=102
        value=360
        value=301
        value=24
        value=197
        value=694
        value=278

    foreach=maxdepths
        type=experiment.maxdepth
        name=v-mdepth
        value=25
        value=50
        value=75
        value=100
        value=125
        value=150
        value=175
        value=200
        value=250
        value=300

    #     value=400
    #     value=500
    #     value=700
    #     value=750
    #     value=1000
    #     value=2000
    #     value=4000
    #     value=8000
    #     value=50000
    #     value=100000

    adapter=a
        codingdir=var-coding
        pco=up1
        pco=low2
        pco=low3
```

```
pco=upbase
        encoder=enCodingOfCFsp
        decoder=CFsp_nl2CFsp
        regexp={RECVHEX[^0+0

pco=lowbase
        encoder=enCodingOfUdp
        decoder=udp_nl2udpsp
        multiplexer=pcoOfUdp
        regexp={RECVHEX[^0+0

pco=up1
        base=upbase
        address=up
        ievent=v-iev0
        oevent=v-oev0

pco=low2
        base=lowbase
        address=low2
        ievent=v-iev2
        oevent=v-oev2

pco=low3
        base=lowbase
        address=low3
        ievent=v-iev3
        oevent=v-oev3

iap=up
        address=up

iap=low
        address=low1

channel=in
        iokind=input
        pco=up1
        pco=low2
        pco=low3

channel=out
        iokind=output
        pco=up1
        pco=low2
        pco=low3
        timeout=5
        sevent=Delta

address=up
        name=var-address0
        value=pipe
```

```
address=low1
        name=var-address1
        value=udp!*!1075

address=low2
        name=var-address2
        value=udp!*!1076

address=low3
        name=var-address3
        value=udp!*!1077

var=l-u-iev0
        name=v-iev0
        type=pco.ievent
        value=cfsap_in!*!*

var=l-u-oev0
        name=v-oev0
        type=pco.oevent
        value=cfsap_out!*
        value=cfsap_out!*!*

var=l-l-iev2
        name=v-iev2
        type=pco.ievent
        value=udp_in!udp1!udp_req(udp2,*)
        value=udp_in!udp2!*

var=l-l-oev2
        name=v-oev2
        type=pco.oevent
        value=udp_out!udp1!udp_ind(udp2,*)
        value=udp_out!udp2!*

var=l-l-iev3
        name=v-iev3
        type=pco.ievent
        value=udp_in!udp1!udp_req(udp3,*)
        value=udp_in!udp3!*

var=l-l-oev3
        name=v-oev3
        type=pco.oevent
        value=udp_out!udp1!udp_ind(udp3,*)
        value=udp_out!udp3!*

var=l-u-address
        name=var-address0
        type=address.name
        value=cf1

var=l-l-address1
```

```
        name=var-address1
        type=address.name
        value=udp1

    var=l-l-address2
        name=var-address2
        type=address.name
        value=udp2

    var=l-l-address3
        name=var-address3
        type=address.name
        value=udp3

    var=l-coding
        name=var-coding
        type=adapter.codingdir
        value=$campaign/coding/LOTOS

    experiment=defaults
        msg=msg
        log=log
        dir=$campaign/experiment/$experiment
        makefile=$campaign/experiment/$experiment/torx.mk
        mkinclude=experiment.incl
        config=$campaign/experiment/$experiment/torx.if
        runs=var-runs
        seed=var-s
        maxdepth=v-mdepth
        var=l-u-iev0
        var=l-u-oev0
        var=l-l-iev2
        var=l-l-oev2
        var=l-l-iev3
        var=l-l-oev3
        var=l-u-address
        var=l-l-address1
        var=l-l-address2
        var=l-l-address3
        var=l-coding
        primer=pl
        adapter=a
        impl=janbase
        pre=:
        post=:
        driver=torx

    # ================================================================

    experiment=001
        base=defaults
        runs=1
        maxdepth=1000
```

```
product=expr
        type=experiment
        foreach=seeds
        prefix=expr
        template=001

# ============================================================

campaign=main
        dir=/home/fmg/feenstra/jf/campaign/confprot
        makefile=$campaign/Makefile
        experiment=product=expr

# ============================================
```

BUGS

The campaign configuration language, and its tool support, are, at best, an interesting prototype, that still needs a number of iterations. Too much detail can and must be specified, the variable mechanism could be improved.

In general, it will be easier to write a shell script to invoke **torx**(1) in the way described in **torx**(1) than it is to use **campaign**.

The main problem seems to be that we picked a limited syntax and stayed with it, even though it became increasingly painful to add to it the features that (we think) we need.

SEE ALSO

torx-intro(1), **environ**(5)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of torx.

NAME

cppmkprimer − preprocess input with cpp before invoking mkprimer

SYNOPSIS

cppmkprimer [*preproc-args ...* **] .***newsuffix specification***.***suffix*

DESCRIPTION

cppmkprimer invokes the preprocessor **cpp**(1) on input file *specification***.***suffix* with the given *preproc-args* to generate the file *specification***.***newsuffix* on which then **mkprimer**(1) is invoked.

m4mkprimer is a simple wrapper around **preprocmkprimer**(1).

SEE ALSO

torx-intro(1), **mkprimer**(1), **cpp**(1), **m4mkprimer**(1), **preprocmkprimer**(1)

BUGS

It is not possible to specify command line arguments for **mkprimer**(1).

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of **mkprimer**.

NAME

hexcontext − run program in hexadecimal context

SYNOPSIS

hexcontext [-debug [*nr* **]] [-port** *portnr* **] [-[no]printdata] [-[no]printdatahex]** *program args ...*

DESCRIPTION

hexcontext starts the given *program* with given *args*, keeping pipes between itself and the standard input, standard output and standard error of the started *program*. If this succeeds, it waits for commands on standard input and output of the *program* that arrives on the pipes, until end of file is detected on the standard input of *hexcontext* and the standard output and standard error of *program*, after which *hexcontext* will wait for *program* to exit, and then exits itself. The recognized commands are discussed below. When output of the program arrives on the pipes, *hexcontext* outputs on standard output a line of the form

RECV *pipe data*

if printing of data is enabled, and/or, if printing of data in hexadecimal form is enabled, a line of the form

RECVHEX *pipe datahex*

In these lines *pipe* will be **stdout** if the data was received from the standard output of *program*, and **stderr** if the data was received from the standard error of *program*, and *data* and *datahex* are the contents of the message, as received resp. in hexadecimal form. By default, output in hexadecimal format is enabled, and output in ''normal'' format is disabled. This can be changed using the command line options **-[no]printdatahex** and **-[no]printdata** and with corresponding commands, as discussed below.

The **-debug [***nr***]** option opens a hardcoded pseudo terminal (pty) on which debugging information is printed. The amount of information printed depends on the numeric debug mode given. For more information, use the source.

COMMANDS

The following commands can be given on the standard input of **hexcontext**. The command keyword (printed in capitals in this section) is recognized regardless of case (uppercase, lowercase, mixed).

SENDHEX *datahex*
send the data (given as hexadecimal string) to the standard input of *program*.

PRINTDATA
enable printing of data ''as received'', in the form of **RECV** lines

NOPRINTDATA
disable printing of data ''as received'', in the form of **RECV** lines

PRINTDATAHEX
enable printing of data in hexadecimal form, in the form of **RECVHEX** lines

NOPRINTDATAHEX
disable printing of data in hexadecimal form, in the form of **RECVHEX** lines

DEBUG [*nr***]**
set debugging mode. Debugging mode 0 disables debugging, for the other modes, see the source.

NODEBUG
disable debugging

SEE ALSO

torx-intro(1), **tcp**(1), **udp**(1), **unhexify**(1)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of **torx**.

NAME

instantiator − instantiate free variables for torx

SYNOPSIS

instantiator [-f *configFile* **] [-s** *seed* **]**

DESCRIPTION

instantiator is an experimental TorX component that is meant as a filter between Driver and Primer: it reads (on standard input) the messages that are sent from Driver to Primer, and instatiates variables in these messages, and writes the resulting messages to standard output. The ‘‘instantiation’’ is just syntactical substitution. Which variables have to be instantiated, from which domains, can be expressed in configuration file, which can be named using the **-f** *configFile* flag. If no **-f** *configFile* flag is given, **instantiator** tries to read file ‘‘instconfig.txt’’. The instatiation values are chosen randomly from the domains given in the configuration file; the seed of the random number generator can be set using the **-s** *seed* flag. If necessary, **instantiator** will repeatedly (recursively) try to apply instantiation rules, until no changes occur. This can be used to instantiate a variable with an expression that contains new variables that then will also be instantiated, etc.

USE IN TORX

We usually use the **instantiator** in the following way. We make a small wrapper shell script of the following form:

```
#! /bin/sh
instantiator -s 0 -f  instconfig.txt | primer-program "$@"
```

where primer-program is the actual primer program file that has to be invoked. So, all standard input send to the primer-program first passes through the **instantiator**. In the **torx-config**(4) configuration file (or in other places where we have to give the primer program file) we now give the wrapper script instead of the ‘‘real’’ primer program.

CONFIGURATION FILE FORMAT

The configuration file consists a number of substitution entries of the form:

```
type : position : prefix : regexp : domain
```

Empty lines, lines containing only whitespace and comment lines (lines starting with optional whitespace followed by #) are ignored. Continuation lines are allowed: these should end with a \ character; this character is replaced by a space character when the lines are joined. leading and trailing whitespace in the lines, and in the fields (type, position, prefix, regexp, domain) is ignored.

type is the type of the variable, as given in the expression that we have to instantiate. In general, variables in TorX have the form **var_***type***[$***i***]** with **var_** a fixed prefix, and **$***i* an optional numerical suffix to make the variables unique. Note: variables of the type **mtype** are treated special: as soon as a value is found for it, it is substituted, so the found value can be used as ‘‘context’’ in the regular expression **regexp** field.

position

is a non-negative (usually also non-zero) number that refers to the part of the event (action) that has to which the rule applies, where we take the **!** character as a separator in the action, as in **pos***0***!pos***1***!pos***2* etc. The special value ‘‘∗’’ makes the rule applicable to any position.

prefix is a regular expression that refers to the command in the TorX Primer-Driver interface (see **torx-primer**(5)) for which the rule applies. This will usually be the command **C_INPUT** (for torx version 2.∗) or the command **C_GETINPUT** (for torx version 3.∗). Note that it will only look at the **event** field of this command (this is hardcoded in **instantiator**; maybe this should be configurable too). Note: all lines belonging to the same instantiation group should have the same prefix (because, as soon as an instantiation is possible using one of the prefixes, the rules of the other prefixes will not be tried).

regexp is a regular expression that specifies the ‘‘context’’ of the rule. For a rule to be applicable, the **regexp** has to be matched.

domain specifies (enumerates) the values that can be chosen from. Currently there are two syntaxes

allowed. The first is a set-like notation, consisting of an enumeration of values, enclosed between { and }, separated by commas, where the values themselves do not contain commas. No whitespace is allowed between { and }. The disadvantage of this format is that the values may not contain commas. Example: {val1,val2,val3,val4}

The second format does not have this restriction. Also this contains of an enumeration of values, now enclosed between (and), separated by whitespace, where the values themselves may not contain whitespace. Example: (e1(e2,e3) b1 c1 d1(e3,f3(g,h)))

EXAMPLES

Conference Protocol in Promela

Here we have to instantiate events of the form

from_upper!LEAVE!var_byte!var_byte

from_lower!PDU_JOIN!var_byte!var_byte!var_byte!var_byte

All (most) variables here are of type ''byte'', even though semantically these ''bytes'' represent different things. That is why we need the **position** field here, to distinguish the different semantical domains.

```
# ================================
# empty lines and comment lines are ignored.
# comment lines are lines that start with
# (optional whitespace followed by) a hash sign (#).
byte:2:C_INPUT:from_upper!JOIN!.*!.*:{1,2,3,4}
byte:3:C_INPUT:from_upper!JOIN!.*!.*:{1,2}

# NOTE: _both_ lines of this regexp group have same prefix;
# if one line has C_INPUT and the other C_INPU(S?) then
# only one instantiation will be done even if an input line
# contains two var_byte fields...
byte:2:C_INPUT(S?):from_upper!LEAVE!.*!.*:{1,2,3,4}
byte:3:C_INPUT(S?):from_upper!LEAVE!.*!.*:{1,2}

byte:2:C_INPUT:from_upper!DREQ!.*!.*:{1,2,3,4}
byte:3:C_INPUT:from_upper!DREQ!.*!.*:{1}

byte:2:C_INPUT:from_lower!PDU_JOIN!.*!.*!.*!.*:{1,2,3,4}
byte:3:C_INPUT:from_lower!PDU_JOIN!.*!.*!.*!.*:{1,2}
byte:4:C_INPUT:from_lower!PDU_JOIN!.*!.*!.*!.*:{0,2}
byte:5:C_INPUT:from_lower!PDU_JOIN!.*!.*!.*!.*:{1}
# ================================
```

Conference Protocol in Promela

Another more elaborate example for the same events. Here we first replace the ''byte'' variables with new variable names with a more expressive type name, which we then instantiate.

```
# ================================
byte:2:C_INPUT:from_upper!JOIN!.*!.*:{var_usertitle}
byte:3:C_INPUT:from_upper!JOIN!.*!.*:{var_conferenceid}

byte:2:C_INPUT:from_upper!LEAVE!.*!.*:{var_usertitle}
byte:3:C_INPUT:from_upper!LEAVE!.*!.*:{var_conferenceid}

byte:2:C_INPUT:from_upper!DREQ!.*!.*:{var_len}
byte:3:C_INPUT:from_upper!DREQ!.*!.*:{var_data}

byte:2:C_INPUT:from_lower!PDU_JOIN!.*!.*!.*!.*:{var_usertitle}
byte:3:C_INPUT:from_lower!PDU_JOIN!.*!.*!.*!.*:{var_conferenceid}
byte:4:C_INPUT:from_lower!PDU_JOIN!.*!.*!.*!.*:{var_udpaddr_src}
```

```
byte:5:C_INPUT:from_lower!PDU_JOIN!.*!.*!.*!.*:{var_udpaddr_dst}

byte:2:C_INPUT:from_lower!PDU_ANSWER!.*!.*!.*!.*:{var_usertitle}
byte:3:C_INPUT:from_lower!PDU_ANSWER!.*!.*!.*!.*:{var_conferenceid}
byte:4:C_INPUT:from_lower!PDU_ANSWER!.*!.*!.*!.*:{var_udpaddr_src}
byte:5:C_INPUT:from_lower!PDU_ANSWER!.*!.*!.*!.*:{var_udpaddr_dst}

byte:2:C_INPUT:from_lower!PDU_LEAVE!.*!.*!.*!.*:{var_usertitle}
byte:3:C_INPUT:from_lower!PDU_LEAVE!.*!.*!.*!.*:{var_conferenceid}
byte:4:C_INPUT:from_lower!PDU_LEAVE!.*!.*!.*!.*:{var_udpaddr_src}
byte:5:C_INPUT:from_lower!PDU_LEAVE!.*!.*!.*!.*:{var_udpaddr_dst}

byte:2:C_INPUT:from_lower!PDU_DATA!.*!.*!.*!.*:{var_len}
byte:3:C_INPUT:from_lower!PDU_DATA!.*!.*!.*!.*:{var_data}
byte:4:C_INPUT:from_lower!PDU_DATA!.*!.*!.*!.*:{var_udpaddr_src}
byte:5:C_INPUT:from_lower!PDU_DATA!.*!.*!.*!.*:{var_udpaddr_dst}

mtype:1:C_INPUT:from_lower!.*!.*!.*!.*!.*:{PDU_JOIN,PDU_ANSWER,PDU_LEAVE,PDU_DATA}
mtype:1:C_INPUT:from_upper!.*!.*!.*:{JOIN,LEAVE,DREQ}

usertitle:*:C_INPUT:from_upper!.*!.*!.*:{101,102,103,104}
conferenceid:*:C_INPUT:from_upper!.*!.*!.*:{51,52}
udpaddr_src:*:C_INPUT:from_upper!.*!.*!.*:{0,2}
udpaddr_dst:*:C_INPUT:from_upper!.*!.*!.*:{1}
len:*:C_INPUT:from_upper!.*!.*!.*:{21}
data:*:C_INPUT:from_upper!.*!.*!.*:{31,32,33,34}

usertitle:*:C_INPUT:from_lower!.*!.*!.*!.*!.*:{101,102,103,104}
conferenceid:*:C_INPUT:from_lower!.*!.*!.*!.*!.*:{51,52}
udpaddr_src:*:C_INPUT:from_lower!.*!.*!.*!.*!.*:{0,2}
udpaddr_dst:*:C_INPUT:from_lower!.*!.*!.*!.*!.*:{1}
len:*:C_INPUT:from_lower!.*!.*!.*!.*!.*:{21}
data:*:C_INPUT:from_lower!.*!.*!.*!.*!.*:{31,32,33,34}

unknown:*:C_INPUT:from_upper!.*!.*!.*:( join(var_usertitle,var_conferenceid) \
                        answer(var_usertitle,var_conferenceid) \
                        leave(var_usertitle,var_conferenceid) \
                        data(var_len,var_data) )
# ================================
```

Conference Protocol in LOTOS, Symbolic

Here the variables already have a clear type, so we don't have to look at the position instantiate. We do use the ''repeated rule application'' feature to construct ''complex'' values, by repeatedly instantiating variables with expressions that contain new variables, that are then instantiated, etc.

```
# ================================
DataField     :*:C_INPUT:.*:{m1,m2,m3,m4}
DataFieldLen  :*:C_INPUT:.*:{l}
UserTitle     :*:C_INPUT:.*:{ut_A,ut_B,ut_C,ut_D}
ConfIdent     :*:C_INPUT:.*:{ci_one}
# ConfIdent     :*:C_INPUT:.*:{ci_one,ci_two}
UDPAddress_dst:*:C_INPUT:.*:{udp1}
UDPAddress_src:*:C_INPUT:.*:{udp2,udp3}

CFsp          :*:C_INPUT:.*:( datareq(var_DataField)          \
```

```
                        join(var_UserTitle,var_ConfIdent)    \
                        leave                                )

        UDPsp         :*:C_INPUT:.*:( udp_datareq(var_UDPAddress_dst,var_PDU) )
        #UDPsp        :*:C_INPUT:.*:( udp_dataind(var_UDPAddress_dst,var_PDU) )

        PDU           :*:C_INPUT:.*:( PDU_J(var_UserTitle,var_ConfIdent)     \
        #                             PDU_A(var_UserTitle,var_ConfIdent)     \
        #                             PDU_L(var_UserTitle,var_ConfIdent)     \
                                      PDU_D(var_DataFieldLen,var_DataField)  )
        # ================================
```

SEE ALSO

torx-intro(1), **torx-primer**(5)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of **torx**.

NAME

intersector − combine multiple torx primers

SYNOPSIS

intersector [*options ...* **]** *configuration-file ...*

DESCRIPTION

intersector is an experimental program to integrate the menu's received from multiple **torx-primer**(5). It presents itself as a single Primer to the Driver. One way to see it, is as a kind of multiplexer. So, on the one hand, it interfaces with each primer, using the primer-driver interface, playing the role of the driver. On the other hand, it interfaces with the driver, playing the role of the primer.

It is called **intersector** because it (sort of) computes the (mathematical) intersection of the menu's that it receives.

In addition, for execution of test cases generated with the tool TGV **intersector** can be configured to interpret action (event) labels as verdicts: the presence of a particular action in the menu then means that a particular verdict has been given.

intersector takes the command line options as given in **torx-primer**(5). Most of these it just passes on to the Primers that it invokes. On start-up, the **intersector** reads its *configuration-file* which is similar to the **torx-config**(4) configuration file, and the configuration file of the **partitioner**(1). It then starts the **torx-primer**(5) that are specified in its configuration file, and asks them for their inputs and outputs, after which it waits for Primer-Driver interface commands on its standard input.

In the interaction with a test-purpose explorer/primer program, the **intersector** assumes that the test-purpose explorer/primer program knows about suspension (quiescense) actions: in **C_OUTPUT** interface commands it will use **suspension=1** where appropriate. Note that this has been changed in TorX version 3.2; in earlier versions of TorX, it would always send **suspension=0** in the interactions with a test-purpose explorer/primer program.

OPTIONS

intersector supports the following commandline options, which are all just passed to the **torx-primer**(5) that it invokes.

-s *number*
: the seed for the random number generator

-i *gates1,gate2,gate3,...*
: the list of input gates. Note there are no spaces between the gates!

-o *gates1,gate2,gate3,...*
: the list of output gates. Note there are no spaces between the gates!

-S *algorithm*
: the algorithm which can be **ioco**, **traces** or **simulation**.

-d *delta-event-tag*
: the *delta-event-tag* is used for quiescense in the interface.

CONFIGURATION FILE

The configuration file consists of a number of single-line entries as in **torx-config**(4). Several entries have a field *id*. An *id* is just an arbitrary name, that is intended to group together entries that describe information about the same Primer: these entries should contain the same value for *id*.

SPEC *id filename* **[** *args* **]**
: The filename of explorer/primer program *id*, and its (optional) arguments. The explorer/primer program will be started from the directory given with the **RUNDIR** entry for *id*. Note that the default value for **RUNDIR** is *not* the current working directory!

TEST *id filename* **[** *args* **]**
: The filename of test-case explorer/primer program *id*, and its (optional) arguments, for execution a

already generated test case. The explorer/primer program will be started from the directory given with the **RUNDIR** entry for *id*. Note that the default value for **RUNDIR** is *not* the current working directory!

GUIDE *id filename* **[** *args* **]**
The filename of test-purpose explorer/primer program *id*, and its (optional) arguments. The explorer/primer program will be started from the directory given with the **RUNDIR** entry for *id*. Note that the default value for **RUNDIR** is *not* the current working directory!

RUNDIR *id directory*
The directory from which the explorer/primer program of *id* will be started. Default value: the directory containing the explorer/primer program as specified in the **SPEC**, **TEST**, or **GUIDE** entry for *id.*

LABEL-DELTA *id label ...*
The action (label) that represents quiescense (suspension) for sub-primer *id*. This value should be parseable as a LOTOS event. Default value: Delta

LABEL-HIT *id label ...*
The action (label) that represents ‘‘hitting’’ the test-purpose for sub-primer *id*.

LABEL-MISS *id label ...*
The action (label) that represents ‘‘missing’’ the test-purpose for sub-primer *id*.

LABEL-PASS *id label ...*
The action (label) that represents ‘‘passing’’ the test for sub-primer *id*.

SEED *number*
specifies the seed for the random number generator, and is also passed down the the invoked Primer. Note: it is better to *not* specify this in the configuration file, but to just use the value given with the **--seed** flag.

CHOOSEINPUTS *number*
Indicate whether or not the **intersector** should select inputs from the menu, if the user does not choose. This is needed if an *iochooser* is used to choose values for ‘‘symbolic’’ events in the *Promela* specification. Allowed values: 0 (false), 1 (true). Default value: 0

SPECTIMEOUT *number*
Specify how long to wait for the spec to respond. This value should not be configured by the user. Default value: -1 (indicating: infinity)

LOGFILE

intersector combines the STATS adn STATEID of the primers that it invokes. Each STATS and STATEID line of a primer is prefixed with four space-separated words, followed by a space, resulting in something like:

id *id* **role** *role* stats line from primer

with *id* the id used for the primer in the **intersector** configuration file, and *role* the role of that primer, i.e. one of **spec**, **guide**, or **test**.

EXAMPLE

The following example starts a primer together with a test purpose. The ‘‘hit’’ label is set to ‘‘epsilon’’, which is the right value for **jararaca**(1) (when invoked with the right flag).

```
#==========================================
SPEC spec ../LOTOS/primer.sh
GUIDE tp ../TP/primer.sh
LABEL-HIT tp epsilon
LABEL-DELTA tp Delta
#==========================================
```

SEE ALSO

torx-intro(1), **torx-primer**(5), **partitioner**(1), **torx-log**(4)

BUGS

Th implementation is built reusing parts of already existing programs, and thus may contain some ‘‘dead’’ code.

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of **torx**.

NAME

iochooser − suggest by probabilities to stimulate or observe

SYNOPSIS

iochooser [-s *seed* **] [-debug] [** *bound***:***value* **[:...]]**

DESCRIPTION

iochooser can be used as a filter between the TorX Driver and a TorX Primer, to use probabilities in the decision between stimulating and observing. The **iochooser** program intercepts the **C_IOKIND** commands send from Driver to Primer, and, when such a command does not have a ‘‘suggestion’’ already (which means that the Driver leaves the decision to the Primer), the **iochooser** adds a ‘‘suggestion’’, using probabilities.

OPTIONS

The following command line options are supported:

-s *seed* specify the seed for the random number generator (default value: 0)

-debug generate debugging output

*bound***:***value***:...** specify the possible suggestion values, and their probabilities. This argument is a single string of colon (:) separated fields. There should always be an even number of fields. Two subsequent fields specify a bound and a suggestion value. The **bound** values should be between 0.0 and 1.0, and should increase from left to right. **iochooser** uses this argument as follows: When a suggestion value has to be generated, **iochooser** generates a random number between 0.0 and 1.0. It then compares the generated number with the bounds, from left to right, and uses the **value** field of the first **bound** in the argument that is greater or equal to the generated number. (default value: 0.5:iokind=input:1.0:iokind=output)

EXAMPLE

Below we show a **sh**(1) shell script that demonstrates how **iochooser** can be put as filter between the Driver and a Primer. The shell script should be specified as explorer/primer program in a **torx-config**(4) configuration (i.e. as value of a **SPEC** line). It assumes that the probability of doing an input, and the ‘‘real’’ explorer/primer program are specified as **SPECFLAGS** value in the **torx-config**(4) configuration file.

```
#============================================
#!/bin/sh
## We assume that we specify the probability and the primer program
## as values of SPECFLAGS in the torx-config(4) configuration file,
## as in:
##
##     SPECFLAGS 0.3 /my/path/to/my/real/primer
##
## which means they will be the last values in the argument list
## given to this script.
## We store those two values in variables PROB resp. PRIMER,
## and then strip them from the list of arguments with which
## we invoke the primer.
## NOTE there should be (hopefully is) a better way to do the
## command line argument dance below.

# use: number of arguments we consume here
use=2
if [ $# -lt $use ]
then
        echo "usage: script [ primer-args... ] prob primer" 1>&2
        exit 1
```

```
fi

## construct command (cmd) to re-set the positional parameters
## to the list of n that we want to pass to the primer, like:
##       set "$1"  "$2" ... "$n"
## and set PROB and PRIMER
cmd=set
n=‘expr $# - $use‘
i=1
while [ $i -le $n ]
do
      cmd="$cmd \"\$$i\""
      i=‘expr $i + 1‘
done
eval PROB=\$$i
i=‘expr $i + 1‘
eval PRIMER=\$$i
i=‘expr $i + 1‘

## only eval the command to re-set the positional parameters
## if there are positional parameters to be set
## otherwise, unset the positional parameters using shift
## (old bourne shells do not allow an argument to shift)
if [ $# -gt $use ]
then
      eval $cmd
else
      i=1
      while [ $i -le $use ]
      do
            shift
            i=‘expr $i + 1‘
      done
fi

## ready to start the real work
## xtorx will make sure that iochooser is in its PATH
if [ -n "$PROB" ]
then
      bounds_values="${PROB}:iokind=input:1.0:iokind=output"
      iochooser -s 0 $bounds_values | $PRIMER "$@"
else
      $PRIMER "$@"
fi
#===============================================
```

SEE ALSO

torx-intro(1), **torx**(1), **torx-primer**(5), **torx-config**(4), **sh**(1)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of torx.

NAME

jararaca − explore traces generated from regular expressions

SYNOPSIS

jararaca [*options ...* **] [** *file* **]**
jararaca -h

DESCRIPTION

jararaca implements a (non-symbolic) explorer for regular expressions, given in the **jararaca** input language. It reads the regular expressions from *file*, if given, or otherwise from standard input, and builds an automaton for it, that generates the set of traces that can be produced from the regular expression. The automaton is then extended with an epsilon selfloop with (label, event) string "epsilon" on the accept state. **Jararaca** then offers the TorX explorer-primer interface on standard input and output, to ‘‘lazily’’ explore this automaton, and hence, the set of traces. **Jararaca** uses the (label, event) strings defined for its atomic actions, instead of just using the action identifiers themselves.

OPTIONS

The following command line options are supported:

-h print the version number and an overview of the command line options, and exit.

-a use the action identifiers themselves instead of the (label, event) strings defined for them

-d print RFSM in **dot**(1) format to standard output and exit

-e run the explorer-primer interface on standard input and output (this is the default)

-l *eps* add epsilon loop with (label, event) string *eps* to accept state. When the TorX combinator sees a special action in the menu of one of its explorer/primers (by default **epsilon**) it knows that **jararaca** has reached the accept state. (by default a selfloop with (label, event) string **epsilon** is added on the accept state)

-L do not add epsilon self loop on the accept state

-p print parsetree in **dot**(1) format to file *dest***.pt.dot** where *dest* is either the name of the input file, or the string **stdin** if no input file was given and **jararaca** read its input from standard input.

-r print RFSM in **dot**(1) format to file *dest***.rfsm.dot** where *dest* is either the name of the input file, or the string **stdin** if no input file was given and **jararaca** read its input from standard input.

-s output strings instead of action id’s

-v verbose mode

INPUT FORMAT

The input file contains 4 sections, each of which starts with a special keyword that should appear at the start of a line. The sections have to appear in order, and the keyword the starts a next section at the same time closes the previous one. All identifiers should be defined before they can be used. C style comments are allowed (comments start with /∗ and end with ∗/ and can not be nested)

The first section, which starts with the keyword **%description** on a single line, contains a non-formal description of the test purpose, its author, goal, date of writing, etc.

The second section, which starts with the keyword **%declare** on a single line, contains declarations of the actions that are used in the test purpose. An action declaration has the form

action *aid* **"***text-string***" ;**

This defines an action identifier *aid*, together with a ‘‘verbose’’ string representation *text-string* of it. The ‘‘verbose’’ string representation is used unless the **-a** option of **jararaca** is used. If the **-a** option is used, the strings may be left empty (i.e. consisting just of ""). Action declarations can be grouped in named sets, as follows:

set *sid* **{**

action *aid1* "*text-string1*" **,**
action *aid2* "*text-string2*" **,**
...
};

This defines the action identifiers *aid1*, *aid2*, etc. with their string representations *text-string1* and *text-string2* as belonging to the set named *sid*. The action identifiers and the set names can be used in the third and fourth sections, which contain the regular expressions.

The third section, which starts with the keyword **%define** on a single line, contains named regular expressions, as

eid **=** *regular-expression* **;**

This makes that a named *regular-expression* can be used as sub-expression in the regular expressions that follow it (in the third section and in the fourth section), by referring to its name *eid.*

The fourth (and last) section, which starts with the keyword **%objective** on a single line, contains the regular expression for which the automaton should be build, and which should be explored.

REGULAR EXPRESSIONS

The semantics of the regular expressions are defined via a mapping function **T** that maps regular expressions on a set of sequences of actions (i.e. as a set of traces). In the definition we use **.** as concatenation operation on sequences of actions. Below we list the valid regular expressions, and their meaning.

Atomic regular expressions are:

aid — the action defined with action identifier *aid* in the **%declare** section
T(*aid*) = { *aid* }

sid — the action set defined with set identifier *sid* in the **%declare** section, which is interpreted as the choice over the elements *a1*, *a2*, ..., *an* if *sid* was defined as set **{***a1***,** *a2***, ...,** *an***}**
This is equivalent to regular expression **(***a1***|***a2***| ...|***an***)**
T(*sid*) = { *a* | exists <*sid,A*> and <*a,s*> in A }

eid — the regular expression defined with regular expression identifier *eid* in the **%define** section, which is interpreted as **(***e***)** if *e* is the regular expression that is assigned to *eid*
T(*eid*) = { *s* | exists <*eid,re*> and *s* in T(*re*) }

Regular expressions can be built recursively using the following operators, where *e*, *e1*, and *e2* are regular expressions:

*e1***.***e2* — this gives the regular expression formed by concatenating regular expressions *e1* and *e2*
T(*e1***.***e2*) = { *s1.s2* | *s1* in T(*e1*) and *s2* in T(*e2*) }

*e1***|***e2* — this gives the regular expression formed by choosing regular expressions *e1* or *e2*
T(*e1***|***e2*) = T(*e1*) union T(*e2*)

*e1***>***e2* — this gives the regular expression formed by determinstically concatenating regular expressions *e1* and *e2*. This is done by removing the actions that appear at the ''head'' of *e2* from the ''head'' of *e1*, and then non-deterministically doing *e2*, or doing *e1* (minus the removed actions) followed by *e2*

*e***[***m***..***n***]** — this general repetion operator gives the regular expression formed by doing *e* at least *m* and at most *n* times. *m* and *n* should be greater than or equal to zero, and *n* should be greater than or equal to *m*. The special value **infinity** is also allowed for *m* and *n*.
T(*e***[***m***..***n***]**) = Union(*m*<=*i*<=*n*) T(*e***[***i***]**) where
T(*e***[***0***]**) = { EmptyString }
T(*e***[***1***]**) = T(*e*)
T(*e***[***2***]**) = { *s1.s2* | *s1* in T(*e*) and *s2* in T(*e*) }
...
T(*e***[***i***]**) = { *s1*. ... .*si* | *s1* in T(*e*) and ... and *si* in T(*e*) }
...

*e** this gives the regular expression formed by doing *e* zero or more times.
This is equivalent to regular expressions *e***[0..]** and *e***[0..infinity]**

e+ this gives the regular expression formed by doing *e* one or more times
This is equivalent to regular expressions *e***[1..]** and *e***[1..infinity]**

*e***?** this gives the regular expression formed by doing *e* zero or one times
This is equivalent to regular expression *e***[0..1]**

*e***[***n***]** this gives the regular expression formed by doing *e* exactely *n* times.
n should be greater than or equal to zero.
This is equivalent to regular expression *e***[***n***..***n***]**

*e***[infinity]** this gives the regular expression formed by doing *e* infinitely many times.
This is equivalent to regular expression *e***.***e***[infinity]** or *e***[infinity..infinity]**

(*e***)** this gives the regular expression *e*.
This is used for grouping, to avoid ambiguous expressions.
jararaca enforces the use of parentheses where necessary.

EXAMPLES

Conference Protocol Requirements

```
%description

 Testpurpose 1-9
     d.d. 12 december 2001
     author: Rene de Vries

     Goal: Test purposes for requirement 1-9 of conference protocol

%declare
     set LU {
          action o_PduJoin_P1 "udp_out!UDP2!UDP_DATAIND(UDP1,PDU_J(UT_A,CI_ONE))",
          action o_PduJoin_P2 "udp_out!UDP3!UDP_DATAIND(UDP1,PDU_J(UT_A,CI_ONE))",
          action o_PduAnswer_P1_C1 "udp_out!UDP2!UDP_DATAIND(UDP1,PDU_A(UT_A,CI_ONE))",
          action o_PduAnswer_P1_C2 "udp_out!UDP2!UDP_DATAIND(UDP1,PDU_A(UT_A,CI_TWO))",
          action o_PduAnswer_P2_C1 "udp_out!UDP3!UDP_DATAIND(UDP1,PDU_A(UT_A,CI_ONE))",
          action o_PduAnswer_P2_C2 "udp_out!UDP3!UDP_DATAIND(UDP1,PDU_A(UT_A,CI_TWO))",
          action o_PduData_P1 "udp_out!UDP2!UDP_DATAIND(UDP1,PDU_D(L_1,M1))",
          action o_PduData_P2 "udp_out!UDP3!UDP_DATAIND(UDP1,PDU_D(L_1,M1))",
          action o_PduLeave_P1_C1 "udp_out!UDP2!UDP_DATAIND(UDP1,PDU_L(UT_A,CI_ONE))",
          action o_PduLeave_P1_C2 "udp_out!UDP2!UDP_DATAIND(UDP1,PDU_L(UT_A,CI_TWO))",
          action o_PduLeave_P2_C1 "udp_out!UDP3!UDP_DATAIND(UDP1,PDU_L(UT_A,CI_ONE))",
          action o_PduLeave_P2_C2 "udp_out!UDP3!UDP_DATAIND(UDP1,PDU_L(UT_A,CI_TWO))",
          action o_SpDataInd "cfsap_out!CF1!DATAIND(UT_A,M1)"
     };

     setLI{
          action i_PduJoin_C1 "udp_in!UDP2!UDP_DATAREQ(UDP1,PDU_J(UT_A,CI_ONE))",
          action i_PduJoin_C2 "udp_in!UDP2!UDP_DATAREQ(UDP1,PDU_J(UT_A,CI_TWO))",
          action i_PduAnswer_P1_C1 "udp_in!UDP2!UDP_DATAREQ(UDP1,PDU_A(UT_A,CI_ONE))",
          action i_PduAnswer_P1_C2 "udp_in!UDP2!UDP_DATAREQ(UDP1,PDU_A(UT_A,CI_TWO))",
          action i_PduAnswer_P2_C1 "udp_in!UDP3!UDP_DATAREQ(UDP1,PDU_A(UT_A,CI_ONE))",
          action i_PduAnswer_P2_C2 "udp_in!UDP2!UDP_DATAREQ(UDP1,PDU_A(UT_A,CI_TWO))",
          action i_PduData_P1 "udp_in!UDP2!UDP_DATAREQ(UDP1,PDU_D(L_1,M1))",
          action i_PduData_P2 "udp_in!UDP3!UDP_DATAREQ(UDP1,PDU_D(L_1,M1))",
          action i_PduLeave "udp_in!UDP2!UDP_DATAREQ(UDP1,PDU_L(UT_A,CI_ONE))",
```

```
        action i_SpJoin_C1 "cfsap_in!CF1!JOIN(UT_A,CI_ONE)",
        action i_SpDataReq "cfsap_in!CF1!DATAREQ(M1)",
        action i_SpLeave "cfsap_in!CF1!LEAVE"
    };

    action delta "Delta";

%define

    /* general strategies */
    LUD = LU|delta;
    eager = LU*.delta;

    /* rewriting/combinations */
    i_PduData = i_PduData_P1 | i_PduData_P2;
    SpJoinC1 = i_SpJoin_C1.LUD*;
    JoinedConf = SpJoinC1.i_PduAnswer_P1_C1.i_PduAnswer_P2_C1;

    o_PduJoin = o_PduJoin_P1 | o_PduJoin_P2;
    o_PduAnswer_P1 = o_PduAnswer_P1_C1 | o_PduAnswer_P1_C2;
    o_PduAnswer_P2 = o_PduAnswer_P2_C1 | o_PduAnswer_P2_C2;
    o_PduAnswer = o_PduAnswer_P1 | o_PduAnswer_P2;

    /* modeled requirements as test purpose */

    /* we assume 2 potential conference partners */
    Req1 = i_SpJoin_C1.(LUD*>o_PduJoin)[2];

    Req2 = i_SpJoin_C1.LUD*.i_PduJoin_C1.LUD*>o_PduAnswer;
    Req3 = i_SpJoin_C1.LUD*.i_PduJoin_C2.eager;
    Req4 = SpJoinC1.i_PduAnswer_P1_C1.i_PduAnswer_P2_C2.
        i_SpDataReq.i_PduData_P1.eager;
    Req5 = JoinedConf.i_SpDataReq.eager;
    Req6 = JoinedConf.i_PduData_P1.i_PduData_P2.eager;
    Req7 = JoinedConf.i_SpLeave.i_SpJoin_C1.eager.i_SpDataReq.eager;
    Req8 = JoinedConf.i_PduLeave.i_SpDataReq.eager;
    Req9 = SpJoinC1.i_PduData_P1.eager;


%objective
     Req1/*  */
     /* Req2 */
     /* Req3 */
     /* Req4 */
     /* Req5 */
     /* Req6 */
     /* Req7 */
     /* Req8 */
     /* Req9 */
     /*LUD */
```

Using a Preprocessor

Sometimes when we specify multiple requirements in a single **jararaca**(1) input file, it is useful to then use a simple shell script to select one of these requirements with a command line option or an environment

variable (instead of editing the file to uncomment the selected requirement). The **sh**(1) (bourne shell) script below demonstrates how we invoke **primer**(1) as **torx-primer**(5) with **jararaca**(1) as **torx-explorer**(5), where the input file for **jararaca**(1) is preprocessed with **cpp**(1) using environment variable **TPFLAG**. The **%objective** section of the above script would then be replaced by just:

```
%objective
    TP
```

and **TPFLAG** could then be used to define **TP**, for example to something like

```
    -DTP=Req1
```

to select the first requirement.

```
#!/bin/sh
tpfile=confprot.tp

tmptpfile=/tmp/torx$$.tp
cfg=explorer-primer-config.txt

cpp -C -E -P $TPFLAG $tpfile > $tmptpfile

primer -f $cfg "$@" jararaca  $tmptpfile
rm -f $tmptpfile
```

BUGS

The environment variable **TORX_ROOT** is not supported.

Because the TorX explorer-primer interface ‘‘works’’ on standard input and standard output, it is not possible to read the regular expression from standard input *and* run the TorX explorer-primer interface or write the **dot** file to standard output and run the TorX explorer-primer interface.

SEE ALSO

torx-intro(1), **torx-explorer**(5), **torx-primer**(5), **primer**(1), **intersector**(1), **cpp**(1), **dot**(1), **sh**(1), **environ**(5)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of torx.

NAME

jararacy − animate jararaca trace using lefty

SYNOPSIS

jararacy [-m *mcastid* **] [-t** *title* **] [-r] [-k** *key* **]** *dotfile*

DESCRIPTION

jararacy uses **lefty**(1) to animate the automaton (RFSM) generated by **jararaca**(1) in *dotfile* (which should be in **dot**(1) format). After start up, **jararacy** creates a window (with the given *title*) in which it draws the automaton, and then waits for animation commands on standard input. On end of file on standard input, **jararacy** waits for the user to remove the window, after which it exits. The animation commands are expected to be generated using **log2jararacy**(1), e.g. using a unix command as

log2jararacy < *logfile* **| jararacy** *dotfile*

or

tail -f *logfile* **| log2jararacy | jararacy** *dotfile*

Make sure that the *logfile* contains a run of the automaton present in *dotfile*.

Each animation command consists of a single line of text, of the following form:

[init *initinfo* **] [trans** *transinfo* **]**

where *initinfo* and *transinfo* are of the following form

[node] *states* **[edge** *edges* **]**

where *states* and *edges* consist of whitespace separated lists of state/location respectively edge identifiers. Both the **init** *initinfo* and the **trans** *transinfo* parts of the line may be omittted. If **edge** *edges* is omitted from from an *initinfo* or *transinfo*, also the **node** keyword may be omitted.

During animation, the states and edges in the **init** section will be colored red, and those in the **trans** section will be colored orange. The idea is that the *initstates* are those states directly reached by an observable action, and the *transstates* are those states that can be reached from the corresponding *initstates* via internal (invisible, tau) actions.

By default, the window will scroll to follow the colored states. This setting can be toggled with the ''track node'' command in the menu under the right mouse button.

By default, the animation in the window will follow the animation commands read from standard input. If this is disabled, the animation has to be done manually using the left and middle mouse button, and/or with the navigation commands in the menu under the right mouse button (as discussed below). This setting can be toggled with the ''mode'' command in the menu under the right mouse button.

In addition, the animation can be remotely controlled. If the **-m** *mcastid* command line option is given, or environment variable **TORXMCASTID** was set, **jararacy** will attempt to make a remote control connection to the tcp address in *mcastid*. If it succeeds, it will then interpret lines of text read from the remote control connection consisting of a single number as commands to show the corresponding step in the animation. Additionally, whenever the user uses mouse button and/or navigation commands to show a different step, its step number is written to the remote control connection. The remote control connection allows multiple viewers to show the same test step.

The left mouse button and the middle mouse button can be used to ''navigate'' in the animation: the left mouse button will show the ''next'' step in the animation, and the middle mouse button will show the ''previous'' step in the animation. The menu under the right mouse button contains navigation commands ''play fwd'' and ''play bwd'' to play the animation forward resp. backward, and ''|<<--'' and ''-->>|'' to go to the start resp. end of the animation.

Other commands, e.g. to zoom, to open a birdseye view, etc. can be found in the menu under the right mouse button. For these, and for the other commands in the window, please see **dotty**(1).

The **-r** and **-k** *key* command line options are only present for compatibility with **anifsm**(1): they are ignored.

DIAGNOSTICS

Error messages and navigation diagnostics appear on standard error. The navigation diagnostics can be

enabled and disabled with the ‘‘verbose’’ command in the menu under the right mouse button.

BUGS

The animation of ‘‘play fwd’’ and ‘‘play bwd’’ is too fast.

The environment variable **TORX_ROOT** is not supported.

Because the animation commands are read from standard input, it is not possible to read the *dotfile* from standard input.

To overcome problems with the use of reverse video in **lefty**(1), **jararacy** uses **xrdb**(1) to set the following X Windows resource:

LEFTY.reverseVideo: false

SEE ALSO

torx-intro(1), **jararaca**(1), **log2jararacy**(1), **dot**(1), **dotty**(1), **lefty**(1), **torx-logclient**(1), **tmcs**(1), **ani-fsm**(1), **aniwait**(1), **mscviewer**(1), **environ**(5)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of torx.

NAME

jararacy2anifsm − translate from jararacy to anifsm input format

SYNOPSIS

jararacy2anifsm

DESCRIPTION

jararacy2anifsm translates from our ‘‘old’’ **jararacy**(1) format to the new **anifsm**(1) format. It reads from standard input and writes to standard output.

Each **jararacy**(1) animation command consists of a single line of text, of the following form:

[init *initinfo* **] [trans** *transinfo* **]**

where *initinfo* and *transinfo* are of the following form

[node] *states* **[edge** *edges* **]**

where *states* and *edges* consist of whitespace separated lists of state/location respectively edge identifiers. Both the **init** *initinfo* and the **trans** *transinfo* parts of the line may be omittted. If **edge** *edges* is omitted from from an *initinfo* or *transinfo*, also the **node** keyword may be omitted. The edge identifiers should be given in the dotfile as the value of a **name** attribute of an edge, as in

src->*dst* **[label=***action***, name=**e42 , ...**];**

Alternatively, an edge identifier can be of the form

src->*dst*

where *src* and *dst* are states. Note, however, that if *src* and *dst* are linked by multiple edges, an arbitrary one will be chosen! It is much safer to rely on **name** attributes in the dotfile.

During animation, the states and edges in the **init** section will be colored red, and those in the **trans** section will be colored darker red. The idea is that the *initstates* are those states directly reached by an observable action (in *initedges*), and the *transstates* are those states that can be reached from the corresponding *initstates* via internal (invisible, tau) actions (in *transedges*).

The target format is documented in **anifsm**(1).

SEE ALSO

torx-intro(1), **log2jararacy**(1), **jararacy**(1), **anifsm**(1), **environ**(5)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of torx.

NAME

log2anifsm − translate TorX log to anifsm input

SYNOPSIS

log2anifsm [-p *pattern* **] [** *logfile* **]**

DESCRIPTION

log2anifsm reads a **torx-log**(4) from *logfile* or from standard input when no *logfile* is given and writes corresponding input for **anifsm**(1) on standard output, to visualize/animate the automaton of the test run. From the given logfile it takes **super** states from **STATEID** lines matching the given patterns, and actions from **ABSTRACT** lines. Multiple **-p** *pattern* command line options may be given.

If there are states from multiple torx tool components present in the logfile, the **-p** *pattern* command line option may be used to extract only the state information of a single tool component.

SEE ALSO

torx-intro(1), **anifsm**(1), **log2aut**(1), **log2jararacy**(1), **torx-log**(4)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of torx.

NAME

log2aniwait − extract aniwait animation commands from torx log

SYNOPSIS

log2aniwait [*logfile* **]**

DESCRIPTION

log2aniwait extracts state information from a **TorX torx-log**(4) logfile. It takes the information from **ANIWAIT** lines in the given logfile, and outputs them on standard output in the **aniwait**(1) format. If no *logfile* is given, **log2aniwait** reads from standard input.

SEE ALSO

torx-intro(1), **aniwait**(1), **torx-log**(4)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of torx.

NAME

log2aut − translate TorX log to Aldebaran (.aut)

SYNOPSIS

log2aut [*options ...* **]**

DESCRIPTION

log2aut reads a **torx-log**(4) from standard input and writes an automaton in Aldebaran (.aut) format on standard output. Which information is present in the automaton depends on the options given. It is possible to generate an automaton that uses both test step numbers and 'super' STATEID information, which results in an automaton that is no longer valid Aldebaran (.aut) format. Nevertheless it may be useful to do so, and **autexp**(1) can deal with such automata.

By default, in the automaton, the number of time an edge has been taken is indicated by a ***@****number* (with *number* a decimal number) string added to the end of the edge labels. By default, the automaton will use 'super' STATEID numbers for its states, if present in the TorX log, or test step numbers otherwise.

OPTIONS

The idea is that options **-a**, **-t**, and **-T** allow the choice of a 'scheme', and the other options (**-c**, **-C**, **-n**, **-N**, **-s**, **-S**) allow choice to add or not to add a particular feature.

The following 'scheme' command line options are supported:

-a show edge count numbers, and use the 'super' STATEID (if present in the log) as automaton states. (mnemonic: this results in an automaton)

-t use test step numbers but no 'super' STATEID numbers in the automaton state numbers (mnemonic: this results in a trace)

-T use both test step numbers and 'super' STATEID numbers in the automaton state numbers.

The following feature choice command line options are supported:

-c show edge count numbers

-C do not show edge count numbers

-n use test step numbers in the automaton state numbers

-N do not use test step numbers in the automaton state numbers

-s use 'super' STATEID numbers in the automaton state numbers

-S do not use 'super' STATEID numbers in the automaton state numbers

When both **-n** and **-s** are given, or when **-T** is given, the state numbers will be of the form *super_teststep***.** In that case, the following option can be used to influence how the edges are sorted in the automaton. Note that this is *not* valid Aldebaran (.aut) format (where states should just be numbers).

-r if the automaton state numbers contain a '_', give priority to the rightmost element when sorting (by default priority is given to the leftmost element)

BUGS

The order in which the options is given has no effect. (so, it is not possible to first enable an option and then disable it and then enable it once more).

SEE ALSO

torx-intro(1), **autexp**(1), **anifsm**(1), **torx-log**(4)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of torx.

NAME

log2jararacy − extract states from torx log file for jararacy

SYNOPSIS

log2jararacy [-p *pattern* **] [** *logfile* **]**

DESCRIPTION

log2jararacy extracts state information from a **TorX torx-log**(4) logfile. It takes **init** and **trans** states and edges from **STATEID** lines matching the given patterns in the given logfile, and outputs them on standard output in the **jararacy**(1) format. Multiple **-p** *pattern* command line options may be given. If no *logfile* is given, **log2jararacy** reads from standard input.

If there are states from multiple torx tool components present in the logfile, the **-p** *pattern* command line option may be used to extract only the state information of a single tool component.

SEE ALSO

torx-intro(1), **jararaca**(1), **jararacy**(1), **torx-log**(4)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of torx.

NAME

log2mctrl − translate TorX log to mctrl input

SYNOPSIS

log2mctrl [*logfile* **]**

DESCRIPTION

log2mctrl reads a **torx-log**(4) from *logfile* or from standard input when no *logfile* is given and writes corresponding input for **mctrl**(1) on standard output, to update the slider bar. From the given logfile it takes step numbers from **ABSTRACT** and **EXPECTED** lines.

SEE ALSO

torx-intro(1), **mctrl**(1), **log2aut**(1), **log2jararacy**(1), **torx-log**(4)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of torx.

NAME

log2msc − extract Message Sequence Chart from TorX log file

SYNOPSIS

log2msc

DESCRIPTION

The **log2msc**(1) program reads a **torx-log**(4) file from standard input, and writes a corresponding MSC file in ‘‘event oriented textual representation’’ on standard output.

It writes each ‘‘statement’’ of the MSC on its own line, to comply with a limitation in the **mscviewer**(1) program.

SEE ALSO

torx-intro(1), **mscviewer**(1), **torx-log**(4), Ekkart Rudolph, Peter Graubmann and Jens Grabowski *Tutorial on Message Sequence Charts*

NAME

log2primer − generate torx-primer commands from TorX log file

SYNOPSIS

log2primer

DESCRIPTION

log2primer reads a **torx-log**(4) file from standard input, and writes a corresponding list of **torx-primer**(5) **C_INPUT** and **C_OUTPUT** commands on standard output. The idea is that these can then be used to replay (analyse) the trace from the log file with a primer, for example by copying and pasting the commands into a session with **pui**(1).

SEE ALSO

torx-intro(1), **pui**(1), **torx-log**(4), **torx-primer**(5)

NAME

ltsaexp − explore fsp specification using ltsa

SYNOPSIS

ltsaexp [-c *composite* **]** *file*
ltsaexp -l *file*

DESCRIPTION

ltsaexp implements a (non-symbolic) explorer for the language **fsp**, using the tool **ltsa.** It reads the **fsp** specification from the given *file*, and then offers the TorX explorer-primer interface on standard input and output, to ‘‘lazily’’ explore it.

By default **ltsaexp** explores the ‘‘DEFAULT’’ composite process. When invoked with the **-l** flag, **ltsaexp** reports the names of the composite processes that are present in *file* and exits. With the **-c** *composite* flag, **ltsaexp** will explore the composite process named *composite* in *file*, if present, or report an error and exit.

BUGS

The environment variable **TORX_ROOT** is not supported.

SEE ALSO

torx-intro(1), **torx-explorer**(5), **environ**(5)
Jeff Magee and Jeff Kramer, *Concurrency : State Models & Java Programs*, John Wiley & Sons Ltd, 1999
http://www-dse.doc.ic.ac.uk/concurrency/

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of torx.

NAME

m4mkprimer − preprocess input with m4 before invoking mkprimer

SYNOPSIS

m4mkprimer [*preproc-args ...* **] .***newsuffix specification***.***suffix*

DESCRIPTION

cppmkprimer invokes the preprocessor **m4**(1) on input file *specification***.***suffix* with the given *preproc-args* to generate the file *specification***.***newsuffix* on which then **mkprimer**(1) is invoked.

m4mkprimer is a simple wrapper around **preprocmkprimer**(1).

SEE ALSO

torx-intro(1), **mkprimer**(1), **m4**(1) **cppmkprimer**(1), **preprocmkprimer**(1)

BUGS

It is not possible to specify command line arguments for **mkprimer**(1).

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of **mkprimer**.

NAME

mctrl − animation progress scrollbar

SYNOPSIS

mctrl [-r] [-m *mcastid* **] [-t** *title* **]**
mctrlsrv
mctrl -exit

DESCRIPTION

mctrl 'animates' a scrollbar that allows to control other 'connected' animation viewers like **anifsm**(1), **aniwait**(1), **mscviewer**(1), and also **xtorx**(1). After start up, **mctrl** connects to the given **mcast**(1) session, creates a window (with the given *title*) containing a scrollbar, and waits for animation step commands on the mcast connection. It reads step numbers (integers, one per line) on its standard input, and uses them to update the scrollbar. The *mcastid* will usually be of the form **tcp!***hostname***!***portnumber* like tcp!localhost!3456. On end of file on standard input, **mctrl** waits for the user to remove the window (or press the **Quit** button), after which it exits.

Actually, **mctrl** is a shell-level command that uses a running **mctrlsrv** to create or reuse a scrollbar window, and animate it using animation commands received over the mcast connection. The connection between **mctrl** and a running **mctrlsrv** will not be closed until the complete standard input of the **mctrl** command has been processed by **mctrlsrv**. If **mctrl** cannot find a running **mctrlsrv**, it will start a new one. In general, it should not be necessary to start **mctrlsrv** by hand. However, if startup time of **mctrl** is an issue, it may be advantageous to start **mctrlsrv** (by hand) in advance, because a starting **mctrlsrv** may spend some time to check if another **mctrlsrv** is already running.

To display a new scrollbar, **mctrlsrv** will reuse windows that contain a completed animation and have the **Reuse** toggle activated. If more windows are needed, they are created.

The **-r** command line option of **mctrl** will activate the **Reuse** toggle button for the mctrl window.

The animation in the window will follow the step numbers read from standard input and the animation commands read from the mcast connection. The animation can be be done manually using the scrollbar, and/or with the **Step** up and down arrow buttons (as discussed below).

As stated above, the animation can be remotely controlled. Using the **-m** *mcastid* command line option, if given, or environment variable **TORXMCASTID** if set, **mctrl** will attempt to make a remote control connection to the tcp address in *mcastid*. (If neither **-m** *mcastid* is given nor **TORXMCASTID** if set, **mctrl** will exit with a usage message.) If it succeeds, it will then interpret lines of text read from the remote control connection consisting of a single number as commands to show the corresponding step in the animation. Additionally, whenever the user uses the scrollbar and/or navigation commands to show a different step, its step number is written to the remote control connection. The remote control connection allows multiple viewers to show the same test step.

To stop a running **mctrlsrv**, invoke **mctrl** with the **-exit** command line option.

BUTTONS

At the bottom of an mctrl window there are several buttons. The *step number* of the animation step in the trace is shown in the **Step** field. Step numbers start at 0, for the initial state. To visualize the animation step for a known *step*, enter the step number in the **Step** entry field, and hit the return key. If a step number is present in the **Step** field, the down and up arrow buttons can be used to step backwards resp. forwards in the animation.
The 'media player control buttons' can be used to go to the beginning or end of the animation, to (re)play the animation backwards or forwards with a given delay between the steps, and to stop or pause a playing animation. The delay between the steps in an animation is given in the **Delay** entry field (in seconds). To change the delay, enter a positive real number in the **Delay** entry field, and hit the return key, or use the up and down arrow buttons next to the entry field. If the **Loop** toggle button is set when a playing animation reaches the beginning (when playing backwards) or the end (when playing forwards) of the animation, the animation will 'wrap around' and restart at the end resp. beginning.
The **Reuse** toggle button indicates that its window may be reused for a new animation, when end-of-input

has been seen for the animation currently displayed in it. While an animation is in progress (so, when end-of-input has not yet been seen) the **Reuse** button is disabled. (default value: unset, except when overridden by a **-r** command line option of **mctrl**).
The **Close** button closes the window, and, if this was the last remaining mctrl window, exits the progam.
The **Quit** button closes all mctrl windows and exits the progam.

DIAGNOSTICS

Error messages and navigation diagnostics appear on standard error.

BUGS

The environment variable **TORX_ROOT** is not supported.

It should be possible to replay an animation or **torx-log**(4) using the timing information present in the original animation or log (i.e. use the same time between the steps as during the original test run).

A more appropriate name might be **anictrl**.

SEE ALSO

torx-intro(1), **torx-logclient**(1), **tmcs**(1), **anifsm**(1), **aniwait**(1), **mscviewer**(1), **xtorx**(1), **environ**(5)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of torx.

NAME

mctrl − animation progress scrollbar

SYNOPSIS

mctrl [-r] [-m *mcastid* **] [-t** *title* **]**
mctrlsrv
mctrl -exit

DESCRIPTION

mctrl 'animates' a scrollbar that allows to control other 'connected' animation viewers like **anifsm**(1), **aniwait**(1), **mscviewer**(1), and also **xtorx**(1). After start up, **mctrl** connects to the given **mcast**(1) session, creates a window (with the given *title*) containing a scrollbar, and waits for animation step commands on the mcast connection. It reads step numbers (integers, one per line) on its standard input, and uses them to update the scrollbar. The *mcastid* will usually be of the form **tcp!***hostname***!***portnumber* like tcp!localhost!3456. On end of file on standard input, **mctrl** waits for the user to remove the window (or press the **Quit** button), after which it exits.

Actually, **mctrl** is a shell-level command that uses a running **mctrlsrv** to create or reuse a scrollbar window, and animate it using animation commands received over the mcast connection. The connection between **mctrl** and a running **mctrlsrv** will not be closed until the complete standard input of the **mctrl** command has been processed by **mctrlsrv**. If **mctrl** cannot find a running **mctrlsrv**, it will start a new one. In general, it should not be necessary to start **mctrlsrv** by hand. However, if startup time of **mctrl** is an issue, it may be advantageous to start **mctrlsrv** (by hand) in advance, because a starting **mctrlsrv** may spend some time to check if another **mctrlsrv** is already running.

To display a new scrollbar, **mctrlsrv** will reuse windows that contain a completed animation and have the **Reuse** toggle activated. If more windows are needed, they are created.

The **-r** command line option of **mctrl** will activate the **Reuse** toggle button for the mctrl window.

The animation in the window will follow the step numbers read from standard input and the animation commands read from the mcast connection. The animation can be be done manually using the scrollbar, and/or with the **Step** up and down arrow buttons (as discussed below).

As stated above, the animation can be remotely controlled. Using the **-m** *mcastid* command line option, if given, or environment variable **TORXMCASTID** if set, **mctrl** will attempt to make a remote control connection to the tcp address in *mcastid*. (If neither **-m** *mcastid* is given nor **TORXMCASTID** if set, **mctrl** will exit with a usage message.) If it succeeds, it will then interpret lines of text read from the remote control connection consisting of a single number as commands to show the corresponding step in the animation. Additionally, whenever the user uses the scrollbar and/or navigation commands to show a different step, its step number is written to the remote control connection. The remote control connection allows multiple viewers to show the same test step.

To stop a running **mctrlsrv**, invoke **mctrl** with the **-exit** command line option.

BUTTONS

At the bottom of an mctrl window there are several buttons. The *step number* of the animation step in the trace is shown in the **Step** field. Step numbers start at 0, for the initial state. To visualize the animation step for a known *step*, enter the step number in the **Step** entry field, and hit the return key. If a step number is present in the **Step** field, the down and up arrow buttons can be used to step backwards resp. forwards in the animation.
The 'media player control buttons' can be used to go to the beginning or end of the animation, to (re)play the animation backwards or forwards with a given delay between the steps, and to stop or pause a playing animation. The delay between the steps in an animation is given in the **Delay** entry field (in seconds). To change the delay, enter a positive real number in the **Delay** entry field, and hit the return key, or use the up and down arrow buttons next to the entry field. If the **Loop** toggle button is set when a playing animation reaches the beginning (when playing backwards) or the end (when playing forwards) of the animation, the animation will 'wrap around' and restart at the end resp. beginning.
The **Reuse** toggle button indicates that its window may be reused for a new animation, when end-of-input

has been seen for the animation currently displayed in it. While an animation is in progress (so, when end-of-input has not yet been seen) the **Reuse** button is disabled. (default value: unset, except when overridden by a **-r** command line option of **mctrl**).
The **Close** button closes the window, and, if this was the last remaining mctrl window, exits the progam.
The **Quit** button closes all mctrl windows and exits the progam.

DIAGNOSTICS

Error messages and navigation diagnostics appear on standard error.

BUGS

The environment variable **TORX_ROOT** is not supported.

It should be possible to replay an animation or **torx-log**(4) using the timing information present in the original animation or log (i.e. use the same time between the steps as during the original test run).

A more appropriate name might be **anictrl**.

SEE ALSO

torx-intro(1), **torx-logclient**(1), **tmcs**(1), **anifsm**(1), **aniwait**(1), **mscviewer**(1), **xtorx**(1), **environ**(5)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of torx.

NAME

mkprimer-aut − generate a torx primer for aut using autexp

SYNOPSIS

mkprimer [*options ...* **]** *specification***.aut**

DESCRIPTION

From the specification file **mkprimer**(1) generates a **torx-primer**(5) program. In this manual page we describe specific features of primers generated using **autexp**(1).

When **mkprimer**(1) is invoked on a specification file with a **.aut** suffix, or when the command line option **--language AUT** is given, the specification file is interpreted as a Aldebaran (.aut) specification file. From the specification file **mkprimer**(1) generates a **torx-primer**(5) program: a shell-script that invokes the **primer**(1) and via it the explorer **autexp**(1).

LOGFILE

A **autexp**(1) Primer generates a STATEID **torx-log**(4) line containing the following whitespace-separated *name value* pairs:

super *nr*

where *nr* is just an integer number representing a superstate state set

init *state-list*

where *state-list* is a list of comma-separated state identifiers, of the states that are present in the superstate state set, by the last transition done.

trans *state-list*

is a list of comma-separated state identifiers, of the states that are present in the superstate state set, by expansion of the state-list given in the **init** field.

The state identifiers in the **init** and **trans** fields have the following form: *nodeid_edgeid***.***number* where the *nodeid* and *edgeid* correspond to node names and edge names in the **.dot** file that can be generated by **autexp**(1), and *number* represent a state. The *number* is not (directly) related to the structure of the automaton, but dynamically computed during exploration of the automaton (whereas the *nodeid* and *edgeid* names are statically derived from the automaton).

SEE ALSO

torx-intro(1), **mkprimer**(1), **torx-primer**(5), **torx-adaptor**(5), **torx-log**(4)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of **torx**.

NAME

mkprimer-cadp − generate a torx primer for lotos, bcg, fc2 or aut using cadp

SYNOPSIS

mkprimer [*options ...* **]** *specification***.lot**
mkprimer [*options ...* **]** *specification***.bcg**
mkprimer [*options ...* **]** *specification***.fc2**
mkprimer --language AUT-CADP [*options ...* **]** *specification***.aut**

DESCRIPTION

From the specification file **mkprimer**(1) generates a **torx-primer**(5) program. In this manual page we describe specific features of primers generated using CADP (the Caesar Aldebaran Development Package).

When **mkprimer**(1) is invoked on a specification file with a **.lot**, **.bcg**, or **.fc2** suffix, or when the command line option **--language LOTOS**, **--language BCG**, or **--language FC2** is given, the specification file is interpreted as a LOTOS, BCG resp. FC2 specification file, adapted for use with TorX and CADP. The command line option **--language AUT-CADP** can be given to generate a CADP primer for Aldebaran (.aut) files -- by default, when given a file with a **.aut** suffix **mkprimer**(1) generates a primer using **autexp**(1).

LOGFILE

A CADP Primer generates a STATEID **torx-log**(4) line containing the following whitespace-separated *name value* tuples:

super *nr*
where *nr* is just an integer number representing a superstate state set

init *state-list*
where *state-list* is a list of comma-separated state numbers, of the states that are present in the superstate state set, by the last transition done. In the state-list, monotonic increasing sequences of the form *m*,*m*+1,...,*n* are abbreviated as *m*-*n*

trans *state-list*
is a list of comma-separated state numbers, of the states that are present in the superstate state set, by expansion of the state-list given in the **init** field. In the state-list, monotonic increasing sequences of the form *m*,*m*+1,...,*n* are abbreviated as *m*-*n*

The STATS **torx-log**(4) line generated by a CADP Primer consists of a number of whitespace separated ‘‘name value’’ tuples.

#statesini *nr*
number of states in stateset (representing current state) reached by direct action (‘‘visible’’ transition) from the last menu, before expanding (by following internal steps)

#statesall *nr*
number of states in stateset (representing current state), reached by direct action from the last menu, after expanding (by following internal steps)

#statesexp *nr*
number of states added during expansion (by following internal steps)

#statesmaxrun *nr*
maximum number of states reported in #statesall during this test run

#statesmatchini *nr*
number of states matched from the states in #statesini

#statesmatchexp *nr*
number of states matched during expansion (following internal steps)

#statesmatchall *nr*
number of states matched from those in #statesall

#sinkstates *nr*

number of sink states (without outgoing edges)

#events *nr*
number of different actions possible (after expansion)

#eventsmax *nr*
max number of different actions possible in individual state

#eventsexp *nr*
number of different actions encountered during analysis of the states in the stateset

SEE ALSO

torx-intro(1), **mkprimer**(1), **torx-primer**(5), **torx-adaptor**(5), **torx-log**(4)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of **torx**.

NAME

mkprimer-jararaca − generate a torx primer using jararaca

SYNOPSIS

mkprimer [*options ...* **]** *specification***.tp mkprimer [** *options ...* **]** *specification***.jrrc**

DESCRIPTION

From the specification file **mkprimer**(1) generates a **torx-primer**(5) program. In this manual page we describe specific features of primers generated using **jararaca**(1).

When **mkprimer**(1) is invoked on a specification file with a **.tp** suffix, or when the command line option **--language TP** is given, the specification file is interpreted as a jararaca test purpose specification file, to be used in the role of 'guide' with an **intersector**(1). In particular, an epsilon selfloop is added to the accept state, to help the **intersector**(1) to detect when the test purpose (guide) has arrived in its accept state.

When **mkprimer**(1) is invoked on a specification file with a **.jrrc** suffix, or when the command line option **--language JARARACA** is given, the specification file is interpreted as a jararaca specification file, to be used as specification. In particular, no epsilon selfloop is added to the accept state.

In both cases, from the specification file **mkprimer**(1) generates a **torx-primer**(5) program: a shell-script that invokes the **primer**(1) and via it the explorer **jararaca**(1).

LOGFILE

A **jararaca**(1) Primer generates a STATEID **torx-log**(4) line containing the following whitespace-separated *name value* pairs:

super *nr*
: where *nr* is just an integer number representing a superstate state set

init *state-list*
: where *state-list* is a list of comma-separated state identifiers, of the states that are present in the superstate state set, by the last transition done.

trans *state-list*
: is a list of comma-separated state identifiers, of the states that are present in the superstate state set, by expansion of the state-list given in the **init** field.

The state identifiers in the **init** and **trans** fields have the following form: *nodeid_edgeid***.***number* where the *nodeid* and *edgeid* correspond to node names and edge names in the **.dot** file that can be generated by **jararaca**(1), and *number* represent a state.

SEE ALSO

torx-intro(1), **mkprimer**(1), **intersector**(1), **torx-primer**(5), **torx-adaptor**(5), **torx-log**(4)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of **torx**.

NAME

mkprimer-ltsa − generate an fsp primer for torx using ltsa

SYNOPSIS

mkprimer [*options ...* **]** *specification***.lts**

DESCRIPTION

When **mkprimer**(1) is invoked on a specification file with a **.lts** suffix, or when the **--language fsp** command line option is given, the specification file is interpreted as an FSP specification file, by the FSP (LTSA) explorer **ltsaexp**(1), using the 'generic' **primer**(1). From the specification file **mkprimer**(1) generates a **torx-primer**(5) program: a shell-script that invokes the **primer**(1) and via it the LTSA explorer **ltsaexp**(1). In this manual page we describe specific features of primers generated by ltsa from FSP specifications.

LOGFILE

An LTSA Primer generates a STATEID **torx-log**(4) line containing the following whitespace-separated *name value* pairs:

super *nr*

where *nr* is just an integer number representing a superstate state set

init *state-list*

where *state-list* is a list of comma-separated state numbers, of the states that are present in the superstate state set, by the last transition done.

trans *state-list*

is a list of comma-separated state numbers, of the states that are present in the superstate state set, by expansion of the state-list given in the **init** field.

SEE ALSO

torx-intro(1), **mkprimer**(1), **torx-primer**(5), **torx-adaptor**(5), **torx-log**(4)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of **torx**.

NAME

mkprimer-mcrl − generate a torx primer for mcrl using mcrl2 and mcrl

SYNOPSIS

mkprimer [*options ...* **]** *specification***.mcrl**

DESCRIPTION

From the specification file **mkprimer**(1) generates a **torx-primer**(5) program. In this manual page we describe specific features of primers generated using the mcrl and mcrl2 toolkits.

When **mkprimer**(1) is invoked on a specification file with a **.mcrl** suffix, or when the command line option **--language MCRL** is given, the specification file is interpreted as a mCRL specification file and translated to a **.lpe** file which is then accessed using explorer **lpe2torx**(1) using the 'generic' **primer**(1).

LOGFILE

A mcrl Primer generates a STATEID **torx-log**(4) line containing the following whitespace-separated *name value* pairs:

super *nr*

where *nr* is just an integer number representing a superstate state set

init *state-list*

where *state-list* is a list of comma-separated state numbers, of the states that are present in the superstate state set, by the last transition done.

trans *state-list*

is a list of comma-separated state numbers, of the states that are present in the superstate state set, by expansion of the state-list given in the **init** field.

SEE ALSO

torx-intro(1), **mkprimer**(1), **torx-primer**(5), **torx-adaptor**(5), **torx-log**(4)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of **torx**.

NAME

mkprimer-mcrl2 − generate a torx primer for mcrl2 using mcrl2

SYNOPSIS

mkprimer [*options ...* **]** *specification***.mcrl2**

DESCRIPTION

From the specification file **mkprimer**(1) generates a **torx-primer**(5) program. In this manual page we describe specific features of primers generated using the mcrl2 toolkit.

When **mkprimer**(1) is invoked on a specification file with a **.mcrl2** suffix, or when the command line option **--language MCRL2** is given, the specification file is interpreted as a mCRL2 specification file and translated to a **.lpe** file which is then accessed using explorer **lpe2torx**(1) using the 'generic' **primer**(1).

LOGFILE

A mcrl2 Primer generates a STATEID **torx-log**(4) line containing the following whitespace-separated *name value* pairs:

super *nr*

where *nr* is just an integer number representing a superstate state set

init *state-list*

where *state-list* is a list of comma-separated state numbers, of the states that are present in the superstate state set, by the last transition done.

trans *state-list*

is a list of comma-separated state numbers, of the states that are present in the superstate state set, by expansion of the state-list given in the **init** field.

SEE ALSO

torx-intro(1), **mkprimer**(1), **torx-primer**(5), **torx-adaptor**(5), **torx-log**(4)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of **torx**.

NAME

mkprimer-trojka − generate a promela primer for torx using trojka

SYNOPSIS

mkprimer [*options ...* **]** *specification***.trojka**

DESCRIPTION

When **mkprimer**(1) is invoked on a specification file with a **.trojka** suffix, or when the **--language promela** command line option is given, the specification file is interpreted as a promela specification file, adapted for use with TorX. From the specification file **mkprimer**(1) generates a **torx-primer**(5) program. In this manual page we describe how to adapt a promela specification for use with TorX, and we describe specific features of primers generated by trojka from promela specifications.

PREPROCESSOR

The promela specification compiler of TorX, **trojka**, runs its input file through the **cpp**(1) preprocessor, with the preprocessor symbol **TROJKA** defined. This can be used to make a single specification that can be used with both spin and trojka (TorX).

CHANNELS and ENVIRONMENT PROCTYPE

In a promela specification for trojka, we describe the implmementation or system under test (IUT, SUT). All communication via channels is taken to be ‘‘internal’’, invisible, except for channels that are declared **observable**. On an **observable** channel, we either only send or only receive; such a channel is used to communicate with the ‘‘environment’’. Because we describe the system (and its context, as far as necessary), seen from ‘‘within’’ the IUT/SUT, we interpret a send statement (‘‘!’’) on an observable channel as output from the IUT/SUT (i.e. as an observation from the tester’s view), and a receive statement (‘‘?’’) on an observable channel as input to the IUT/SUT, (i.e. as a stimulus from the tester’s view).

For use of such a specification with the spin tool, we have to do two things: to remove the **observable** keyword from the channel declaration (spin does not know about **observable**), and to provide an environment prorcess that produces and consumes the actions communicated over the observable channels. We usually do that as indicated in the example below. This example consists of four parts: 1) the definition of the macro OBSERVABLE, 2) the declaration of a channel, 3) the spefication of the environment, and 4) the **init** statement in which we conditionally start the *environment* process.

SYMBOLIC TESTING

The Trojka promela primer has limited support for symbolic testing. We first discuss the language support for them, and then how they appear in the Primer-Driver interface.

LANGUAGE SUPPORT

Traditionally, in spin it is possible to use variables in input (recveive, ?) statements, to bind variables to the values received. In Trojka, the promela language (syntax) has been extended to also allow the use of variables on output (send, !) statements. The syntax extension has been derived from the C programming language. Note: this extension is *not* compatible with spin, so when the extension is used, some #ifdef statements may be needed to make the specification also usable with spin. The following example statement specifies an output action on channel ‘‘c’’, with mtype ‘‘something’’, values ‘‘10’’, ‘‘value’’ and variable ‘‘variable’’.

```
c ! something,10,value,&variable
```

As usual, variables need to be declared in advance.

VARIABLES IN PRIMER

Variables in input and output actions will appear as **var_***type* (with *type* the type of the variable in promela) in the list of actions generated with the commands C_INPUTS and C_OUTPUTS (see **torx-primer**(5) for these and other commands from the Primer-Driver interface). That means that they also appear like that in the lists of possible stimuli and observations shown in **torx**(1) (when the **menu** command is given) and in **xtorx**(1).

When an input action (stimulus) is requested or done with the C_GETINPUT and C_INPUT commands, all variables are automatically instantiated (by choosing random values). This can be influenced by giving explicit values instead of **var_***type* fields in the event parameter given with C_GETINPUT and C_INPUT

commands. The **instantiator**(1) is a TorX tool component that uses this functionality, by substituting (amongs others) values randomly chosen from the values sets in its configuration file for **var_***type* fields in the event parameter of C_GETINPUT or C_INPUT (as specified in its configuration file).

Another way to influence the automatic subsitution is by invoking the generated primer with a **-I** command line option, to disable automatic instantiation for the C_GETINPUT command. This can be useful if (some) variable(s) should not be instantiated by the Primer, but by the Adapter (because the information needed to instantiate is not available in the primer, but only in the adapter; an example might be time-related information).

LOGFILE

A trojka primer does not generate a STATEID **torx-log**(4) line.

The STATS **torx-log**(4) line generated by a trojka Primer consists of a number of whitespace separated ‘‘name value’’ tuples.

#statevector *nr*
number of allocated statevectors

#statevectorstruct *nr*
number of allocated statevector structures

memusage *size*
size of memory used for the trojka state space representation, in bytes (the trojka primer also consumes memory for other things)

#cutbranch *nr*
number of branches that were cut in the analysis. A branch may be cut if it contains a large (100?) sequence consisting of only internal steps. Warning: if *nr* is non-zero, your test case may be unsound (because the behaviour reached by the branch that was cut will not be taken into accound).

#analysedstates *nr*
number of states analyed

#matchedstates *nr*
number of states that match states already analysed

maxdepth *nr*
lenght of the longest trace that was analysed

#superstate *nr*
number of states in the superstate (state set representing current state, to handle non-determinism)

#execaction *nr*
number of actions executed to do the chosen action. This *nr* gives an indication of the amount of non-determinism

#possaction *nr*
number of different actions possible

PRIMER-DRIVER INTERFACE

When the driver provides an mtype as part of a request to the primer, the matching is done in a case-insensitive way.

EXAMPLES

MACRO OBSERVABLE

```
#ifdef TROJKA
#define OBSERVABLE observable
#else
#define OBSERVABLE
#endif
```

CHANNEL DEFINITION

```
chan c  = [0] of { mtype, byte, byte, byte }  OBSERVABLE;
chan d  = [0] of { byte }  OBSERVABLE;
```

ENVIRONMENT

```
#ifndef TROJKA
proctype environment() {
    do
    :: c ? _,_,_,_
    :: d ! 1
    :: d ! 2
    :: /* all other actions that need to be produced or consumed */
    od
    /*
     * instead of this all producing, all consuming environment,
     * we may want to give a special scenario here
     */
}
#endif
```

INIT STATEMENT

```
init {
    atomic {
        run input(something);
        run underlying_service(something);
        ...
#ifndef TROJKA
        ; run environment()
#endif
    }
}
```

SEE ALSO

torx-intro(1), **mkprimer**(1), **torx-primer**(5), **torx-adaptor**(5), **torx-log**(4)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of **torx**.

NAME

mkprimer − generate a primer for torx

SYNOPSIS

mkprimer [*options ...* **]** *specification*
mkprimer --list
mkprimer

DESCRIPTION

mkprimer generates from a given specification file an executable *primer* program (see **torx-primer**(5)) and two configuration files. By default, **mkprimer** tries to use the suffix of the given specification file to detect the specification language. This can be overruled with the **--language** option. By default, the name of the generated primer file is derived from the given specification file name by omitting the file name suffix. This can be overruled with a **-o** option.

If no specification is given, **mkprimer** lists the specifcation languages that it can handle, together with their corresponding file name suffixes. If the **--list** flag is given, **mkprimer** lists the specifcation languages that it can handle, one per line, on standard output, and exits.

mkprimer takes its information about specification languages and file name suffixes from a number of **mkprimer**(5) modules in the distribution of TorX.

Use of a preprocessor with **mkprimer**(1) is discussed in **cppmkprimer**(1), **m4mkprimer**(1), and **preprocmkprimer**(1).

OPTIONS

-o *primer*
This option specificies the name of the *primer* that will be generated. By default the name is derived from *input-name*, by omitting the recognized language suffix.

--language *name*
This option specificies the format of the input file. It overrules the default file suffix association. This option has to be accompanied by the *-o* option.

--list This option makes **mkprimer** list the specifcation languages that it can handle, one per line, on standard output, after which it exits.

--config *config-file*
This option is only relevant for specifications for which the generic, language independent **primer**(1) is used. This option indicates that in the generatef primer program the **primer**(1) should be invoked with a **-f** *config-file* option.

--inputs *gate-names*
This option is not relevant for Promela specifications. This option indicates that events that 'happen' on gates in *gate-names* should be treated as inputs. It also triggers the generation of the *file***.torx** and the *file***.gates** configuration files. *(this command line option will probably be removed, eventually)*

--outputs *gate-names*
This option is not relevant for Promela specifications. This option indicates that events that 'happen' on gates in *gate-names* should be treated as outputs. It also triggers the generation of the *file***.torx** and the *file***.gates** configuration files. *(this command line option will probably be deprecated)*

FILES

primer The generated primer executable. The interface to it is described in **torx-primer**(5).

*primer***.gates**
this file contains the information given with the **--inputs** and **--outputs** options, as **torx-config**(4) **INPUT** and **OUTPUT** entries containing only gate names, which makes it a suitable (additional) configuration file for **torx**.

primer.**torx**

this file contains information about the location of the specification file, in the form of the following entries in the **torx-config**(4) format. The entries in this file can be used to generate a **SOURCESPEC** or **SOURCEIUT** entry for **torx-config**(4), for example in a **xtorx-extension**(n) file.

SOURCEGIVEN *specification-given*

The *specification* filename with which **mkprimer**(1) is invoked.

SOURCEABS *specification-absolute*

The *specification* filename, but translated to an absolute path.

SOURCEREL *specification-relative*

The *specification* filename, but translated to a relative path, relative to the directory in which the generated primer is written.

SOURCECWD *directory*

The absolute path name to the directory in which the generated primer is written.

TorXdir/**share/torx/Primer**/*language-or-toolkit*.**pm**

a **mkprimer**(5) description file for a language or toolkit supported by TorX.

SEE ALSO

torx-intro(1), **mkprimer-cadp**(1), **mkprimer-ltsa**(1), **mkprimer-trojka**(1), **torx**(1), **torx-primer**(5), **cppmkprimer**(1), **m4mkprimer**(1), **preprocmkprimer**(1), **torx-config**(4), **mkprimer**(5)

BUGS

For LOTOS specifications the input specification needs to have a known suffix, which will guarantee that this suffix is recognized by **open.caesar**, the tool that is used to generate part of LOTOS primers. Using the **mkprimer --language** option does not help here.

The *primer*.**gates** file (and the **--inputs** and **--outputs** options) are only useful if no other information needs to be specified with the **INPUT** and **OUTPUT torx-config**(4) configuration entries.

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of **mkprimer**.

NAME

mscviewer − view a Message Sequence Chart

SYNOPSIS

mscviewer [-r] [-m *mcastid* **] [** *files ...* **]**
Bmsc [-r] [-m *mcastid* **] [** *files ...* **]**
Bmsc -exit

DESCRIPTION

The **mscviewer** program reads MSC's from *files*, or from standard input if no files are given, and displays it to the user, step by step. Each MSC is displayed in a separate window. Instead of waiting for the whole MSC to be available, it will immediately start displaying what it has read, and update the display as soon as it has been able to read more of the MSC.

Bmsc is a shell-level command that causes a running **mscviewer** to load the named MSC files, or to display its standard input. The connection between **Bmsc** and a running **mscviewer** will not be closed until all *files* (or the complete standard input) of the **Bmsc** command have been processed by **mscviewer**, in order to allow the running **mscviewer** to report possible error messages (e.g. about syntax errors) about the files that it processes via the standard error of the **Bmsc** command that sent the files to it. If **Bmsc** cannot find a running **mscviewer**, it will start a new one. To display the new MSC file(s), **mscviewer** will reuse windows that contain a complete MSC and have the **Reuse** toggle activated. If more windows are needed, they are created.

In general, it is probably best to only use the **Bmsc** command, and let it start **mscviewer** when necessary. However, one should be aware of the fact that when a **Bmsc** command is given when no **mscviewer** is currently running, the **Bmsc** will ''become'' a **mscviewer** command, which is ''long-running'' and will only exit when all its windows are closed or the **Quit** button is pressed (or a **Bmsc -exit** command is given). In contrast, a **Bmsc** command given when a **mscviewer** is already running will exit as soon as its files or standard input are processed by the running **mscviewer**.

The **-r** command line option of both **mscviewer** and **Bmsc** will activate the **Reuse** toggle button for the windows that will contain the MSC's given on the same command line or via standard input.

When **Bmsc** is started with only command line option **-m** *mcastid*, or when environment variable **TORXMCASTID** was set, the MSC viewer tries to connect to the address given in the *mcastid* and to use the resulting connection as a remote control connection to synchronise displaying a particular step in the MSC viewer. Whenever the user does something in the user interface that selects a different step in the MSC, its step number is written to the remote control connection. Whenever a step number can be read from the remote control connection, the corresponding step is displayed in the MSC viewer.

When **Bmsc** is started with only one command line parameter: **-exit**, the running **mscviewer** will clean up and exit.

The MSC file should be in *event oriented* textual representation. **mscviewer** indicates both ''normal'' end-of-msc and ''abnormal'' end-of-input without having seen end-of-msc. The ''normal'' end-of-msc is visualized by drawing horizontal bars at the end of every instance in the MSC. The ''abnormal'' end-of-input is visualized by drawing at the end of each instance of the MSC a stippled/dotted contininuation of the instance, and ending that with stippled/dotted horizontal bars.

BUTTONS

At the bottom of the MSC viewer there are several buttons. The **Save as** button opens a dialog box that allows saving of the MSC in postscript form (by choosing or entering a file name ending in a .ps suffix) and in textual form (by choosing or entering any other file name).
The **Font** down and up arrow buttons decrement resp. increment the font size. When a font size change makes this necessary, labels are moved to the right to keep them visible.
The **Highlight** toggle button enables and disables highlighting (default: enabled). Independent of this button, the *step number* of the MSC item under the mouse is shown in the **Step** field. Step numbers start at 1, and are assigened when the *second* part (target) of a message is seen. Step number 0 is special: it used to refer to the instance headers. When highlighting is enabled, the item under the mouse is highlighted by

drawing a box arround it and making the arrow slightly bigger. Also, when a new item is added to the MSC, it is highlighed. To highlight the item for a known *step*, enter the step number in the **Step** entry field, and hit the return key. The MSC window automatically scrolls to make the highlighted item visible. If a step number is present in the **Step** field, the down and up arrow buttons can be used to decrement resp. increment the step number, to move the highlight up resp. down in the MSC.
The **Reuse** toggle button indicates that its window may be reused for a new MSC, when end-of-input has been seen for the MSC currently displayed in it. (default value: unset, except when overridden by a **-r** command line option of **mscviewer** or **Bmsc**).
The **Close** button closes the MSC window, and, if this was the last remaining window, exits the progam.
The **Quit** button closes all MSC windows and exits the progam.

SEE ALSO

torx-intro(1), **xtorx-showmsc**(1), **log2msc**(1), **torx-logclient**(1), **jararacy**(1), **tmcs**(1),
Ekkart Rudolph, Peter Graubmann and Jens Grabowski: *Tutorial on Message Sequence Charts*, Computer Networks and ISDN Systems, Volume 28, Issue 12, June 1996, Pages 1629-1641

FILES

/tmp/mscviewer-*$USER-$DISPLAY*
file to communicate tcp port number on which **mscviewer** listens for **Bmsc** to connect

/tmp/mscviewer-*$USER-$DISPLAY***.pid**
the file containing a list of process identifiers (one per line) of **mscviewer** and its subprocesses

NOTE

The **Bmsc** command was named (and designed) after the **B** shell-level command of the **sam**(1) editor.

BUGS

The current implementation expects each ‘‘statement’’ of the MSC in event oriented textual representation to be on a separate line. The output of **log2msc**(1) complies to this limitation.

The ‘‘endinstance’’ statements in the MSC are ignored; the ‘‘endsmsc’’ statement is used to close all instances.

Only a limited subset of the MSC language is implemented. Valid input is assumed; only very limited checking is done.

The syntax recognized for the MSC language is inferred from the tutorial mentioned above, but not checked with a more formal syntax description. In particular, **mscviewer** expects double quotes (") to be present for MSC items containing whitespace -- whether this is consistent with the MSC standard has not been checked.

When **mscviewer** is started, it checks if other instances of it are running. If so, they are killed. This was added to clean up run-away processes.

When **mscviewer** is given multiple files that are to be processed simultaneously, it has a tendency to process the files one after the other, in reverse order, instead of procesing them in parallel, step by step.

It is counter-intuitive that the **Step** *up* arrow button moves the highlight *down* (because the up button increments the step number, and the steps are numbered increasing from top to bottom).

NAME

partitioner − weight-based test primitive selection for primer

SYNOPSIS

partitioner [*options ...* **]** *configuration-file ...*

DESCRIPTION

partitioner is an *experimental* program to partition input test primitives (stimuli), associate weights with the partitions, and to use those weights when a stimulus has to be randomly chosen. In the TorX tool architecture it is placed between the Driver and the Primer. **partitioner** ''speaks'' the **torx-primer**(5) interface on its standard input and output, and it starts its Primer sub-program (process). It is possible to have multiple partitioners, one after the other.

partitioner partitions the input actions (stimuli) that it gets from the Primer based on information that it reads from a configuration file. It then adds weight information to the partitions, and uses this information when the Driver asks it for a random input.

partitioner takes the command line options as given in **torx-primer**(5). Most of these it just passes on to the Primer that it invokes. On start-up, the **partitioner** reads its *configuration-file* which is similar to the **torx-config**(4) configuration file, and the configuration file of the **intersector**(1). **partitioner** looks for the entries **PARTFILE** (which contains the name of the file containing the weights-patterns combination), **SPEC**, **SPECFLAGS**, **RUNDIR** and **SEED**.

OPTIONS

partitioner supports the following commandline options, which are all just passed to the **torx-primer**(5) that it invokes.

-s *number*
the seed for the random number generator

-i *gates1,gate2,gate3,...*
the list of input gates. Note there are no spaces between the gates!

-o *gates1,gate2,gate3,...*
the list of output gates. Note there are no spaces between the gates!

-S *algorithm*
the algorithm which can be **ioco**, **traces** or **simulation**.

-d *delta-event-tag*
the *delta-event-tag* is used for quiescense in the interface.

CONFIGURATION FILE

The configuration file consists of a number of single-line entries as in **torx-config**(4). Several entries have a field *id*. An *id* is just an arbitrary name, that is intended to group together entries that describe information about the same Primer: these entries should contain the same value for *id*.

PARTFILE *filename*
the name of the file that contains the association between actions (action patterns) and the weights. The format of this file is described in the section PARTITION FILE below.

SPEC *id filename*
The filename of explorer/primer program *id*. The explorer/primer program will be started from the directory given with the **RUNDIR** entry for *id*. Note that the default value for **RUNDIR** is *not* the current working directory!

SPECFLAGS *id arguments*
(Additional) arguments that will be given as arguments to the explorer/primer program of *id* when it is started. Default value: unset

RUNDIR *id directory*
The directory from which the explorer/primer program of *id* will be started. Default value: the

directory containing the explorer/primer program as specified in the **SPEC** entry for *id.*

SEED *number*

specifies the seed for the random number generator, and is also passed down the the invoked Primer. Note: it is better to *not* specify this in the configuration file, but to just use the value given with the **--seed** flag.

PARTITION FILE

The partition file format is experimental. Currently, it just contains Tcl commands to associate names and weights to patterns. The command to make the association is

em_add_pattern *pattern* **[list** *name weight***]**

(where the square brackets [and] are part of the Tcl command). The *name* gives the partition name, and *weight* gives the weight for that partition. The *weight* can be the empty string (which is interpreted as ‘‘1 diveded by the number of partitions’’. The weight of an individual input action is then computed as ‘‘1 divided by the number of elements in its partition’’. The Tcl variable **PARCOUNT** is set to the number of partititions.

PRIMER-DRIVER INTERFACE EXTENSION

The partition names and the weights of the individual actions is shown in the output of the **C_INPUTS** **torx-primer**(5) command. The **partitioner** extends the output that it gets from its Primer with two additional fields:

partition=*name1,name2,...*

where *name* is the name of the partition to which the event belongs. If there are more partitioners between Primer and Driver, then each **partitioner** adds its own partition name (preceded by a comma) to the right of the partition name(s) already put their by its subprocess (Primer). Rephrased: the partition names from partitioners from Primer to Driver appear in order, from left to right, separated by commas.

weight=*value*

the weight of the individual action. The weights are normalised: the sum of all weights of the input actions should be 1.

EXAMPLES

We give here as example the configuration files and partition files for a two-level partition scheme, i.e. we have a ‘‘top-level’’ partitioner and a ‘‘level-1’’ partititoner. The ‘‘top-level’’ partitioner is supposed to be invoked by the Driver, and the ‘‘level-1’’ partititoner invokes the Primer.

CONFIGURATION FILE: top.cfg

```
#============================================
SPEC s ./partitioner
SPECFLAGS s lvl1.cfg
PARTFILE top.part
#============================================
```

PARTITION FILE: top.part

```
#============================================
# initialise patterns
set user 1
while {$user < 20} {
#     em_add_pattern "*!*!user$user" [list user$user {1.0 / $PARCOUNT}]
      em_add_pattern "*!*!user$user" [list user$user "1.0 / $user"]
      incr purid
}
#============================================
```

SUB-PARTITIONER CONFIGURATION FILE: lvl1.cfg

```
#============================================
SPEC s dir/sub/spec/SUT.expr8
```

```
PARTFILE lvl1.part
#============================================
```

SUB-PARTITIONER PARTITION FILE: lvl1.part

```
#============================================
em_add_pattern "*!A!*" [list a "0.1"]
em_add_pattern "*!B!*" [list b "0.1"]
em_add_pattern "*!C!*" [list c "0.8"]
em_add_pattern "*!D!*" [list d ""]
em_add_pattern "*!E!*" [list e ""]
em_add_pattern "*!F!*" [list f ""]
em_add_pattern "*!error!*"   [list err "0.1"]
#============================================
```

PARTITIONER PRIMER-DRIVER INTERFACE OUTPUT

Below follows an example of output of the **C_INPUTS torx-primer**(5) command using the configuration and partition files shown above.

```
C_INPUTS
A_INPUTS_START
A_EVENT event=cin!A!user13  channel=in  partition=a,user13  weight=0.00268762549161
A_EVENT event=cin!C!user13  channel=in  partition=c,user13  weight=0.0215010039329
A_EVENT event=cin!A!user12  channel=in  partition=a,user12  weight=0.00291159428257
A_EVENT event=cin!C!user12  channel=in  partition=c,user12  weight=0.0232927542606
 ...

A_INPUTS_END
```

SEE ALSO

torx-intro(1), **torx-primer**(5), **intersector**(1)

BUGS

Th implementation is built reusing parts of already existing programs, and thus contains quite some ‘‘dead’’ code, even in the configuration file format (the *id* parameter was introduced in the **intersector**(1) but is not used here).

There should be a simpler format for the partition file. On the other hand, the full expressivity of Tcl may have advantages too, as demonstrated with the **while** construct in the example ‘‘top.part’’ partition file.

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of **torx**.

NAME

preprocmkprimer − preprocess input before invoking mkprimer

SYNOPSIS

preprocmkprimer *preprocessor* **[** *preproc-args ...* **]** **.***newsuffix specification***.***suffix*

DESCRIPTION

preprocmkprimer invokes the given *preprocessor* on input file *specification***.***suffix* with the given *preproc-args* to generate the file *specification***.***newsuffix* on which then **mkprimer**(1) is invoked.

SEE ALSO

torx-intro(1), **mkprimer**(1), **m4**(1) **cppmkprimer**(1), **m4mkprimer**(1)

BUGS

It is not possible to specify command line arguments for **mkprimer**(1).

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of **mkprimer**.

NAME

primer − compute test primitives using explorer

SYNOPSIS

primer [*options ...* **]** *explorer* **[** *explorer-options ...* **]**
torx --help

DESCRIPTION

primer starts the given *explorer* with the *explorer-options* and then offers the **torx-primer**(5) interface on its standard input and output. The *explorer* should implement the **torx-explorer**(5) interface.

The **primer** can be used in multiple ''roles''. Usually it will be used to derive test primitives using the **ioco** algorithm, (no **-a** or **-S** flag is needed: **ioco** is the default). It can be used to generate the traces of a specification, without building the suspension automaton as is done for **ioco**, for example, to generate the traces of a test purpose (as in the configuration example below, if used with a **-a traces** command line flag). It can also be used to execute a test that was derived in batch mode, by specifying special events (actions) that denote verdicts in the **primer** configuration file (see below).

OPTIONS

The following command line options are supported:

-s *nr* seed for the random number generator(s)

-a *algorithm* **-S** *algorithm* the test derivation algorithm to use. Accepted values: **ioco**, **sim** (use as simulator), and **traces**. Default value: **ioco**

-D *rundir* start the *explorer* in directory *rundir*

-f *configfile* the *configfile* defines the partitioning of the actions over input and output (and in the future, over multiple **mioco** channels). It also contains the representation of the suspension action. The *configfile* format is discussed below.

-T enable the **ioct** algorithm. This algorithm differs from **ioco** in the way tau (internal, invisible) actions are treated when the suspension automaton is built. **ioco** will not add a suspension action to a state if it contains an outgoing tau (internal, invisible) transition. If for **ioco** for a particular state the presence of such a tau action the only reason is to not add a suspension action, then the **ioct** algorithm will add the suspension action to the state.

-C *iokind* where *iokind* is **input**, **output**, or **input**,**output** enables input, output (or both) completion. When we want to do a ''next'', we look if the action that we want to do is in the menu, if so, we handle as before. If the action is not in the menu, and completion is enabled for this *iokind*, then we just accept (''do'') the action, but remain in the current superstate, and change nothing.

The following obsolete command line options are recognized but silently ignored:

-i *inputg1,g2,...* input gate names
this is obsoleted by the **-f** *configfile* option

-o *outputg1,g2,...* output gate names
this is obsoleted by the **-f** *configfile* option

-d *event* representation of delta (suspension) action
this is obsoleted by the **-f** *configfile* option

-t *iokind* channel types for which suspension events should be added
this is obsoleted by the **-f** *configfile* option, and is now set automaticly for the ''known'' algorithms

CONFIGURATION FILE

The (optional) configuration file contains blocks of *name=value* tuples. The first *name=value* pair of a block should start in column 0 (i.e. not be preceded by whitespace), the other *name=value* pairs of a block

should be preceded by whitespace. Multiple *name=value* pairs may appear on the same line, separated by whitespace.

In principle, the fields of a block may appear in any order. The following values for *name* are recognized:

channel the *value* should be a channel name (for **ioco** by convention **in** for the input channel and **out** for the output channel). A block should contain exactely one **channel** or **verdict** definition.

event the *value* is a pattern for the events that belong to the block. The pattern looks quite a bit like a LOTOS action. The pattern consists of one or more expressions, where the expressions in the pattern are separated by exclamation marks ('!'). An expression can be a word (consisting of letters, digits and underscore), or a word followed by a comma(',')-separated list of expressions between parentheses ('(' and ')'). In the pattern a star '*' can be used instead of an expression or subexpression. The number of '!' in a pattern must match the number of '!' of the event from the specification that it tries to match. A block may contain multiple **event** occurences.

iokind the *value* should be **input** or **output** to indicate whether the block lists input (stimuli) or output (observation) actions. A block should contain at most one **iokind** definition.

suspension the *value* is the event (action) that denotes quiescense (or suspension). For the algorithms **ioco**, **sim** and **traces** an action ''delta'' is automatically added for channel **out**. A block should contain at most one **suspension** definition. If the **suspmode** (see below) for a channel is **recognize**, then **suspension** can be a pattern as described for **event**.

suspmode the *value* should be **compute** or **recognize** to indicate whether suspension actions should be computed or recognized for the channel. For the algorithms **ioco** and **sim suspmode** is automatically set to **compute** for channel **out**; for algorithm **traces suspmode** is automatically set to **recognize** for channel **out**. A block should contain at most one **suspmode** definition.

verdict the *value* should be a verdict: the verdict that is to be associated with the **event** *name=value* pairs of the block. A block should contain exactely one **channel** or **verdict** definition.

EXAMPLE

The following example **primer** configuration file defines two channels (**in**,**out**) with their respective input and output types (**iokind=input**) resp. (**iokind=output**), and the actions that ''belong'' to the channels, which are given using patterns. The default algorithm (**ioco**) automatically adds the implicit action ''delta'' to the list of output actions. This value ''delta'' can be overruled by explicitly adding the wanted value to the **channel=out** block, for example **suspension=my_delta**.

```
channel=in
    event=cfsap_in!*!join(*,*)
    event=cfsap_in!*!datareq(*)
    event=cfsap_in!*!leave
    event=udp_in!udp1!*
    event=udp_in!udp2!*
    event=udp_in!udp3!*
    iokind=input


channel=out
    event=cfsap_out!*
    event=cfsap_out!*!*
    event=udp_out!udp1!*
    event=udp_out!udp2!*
    event=udp_out!udp3!*
    iokind=output
```

SEE ALSO

torx-intro(1), **torx-primer**(5), **torx-explorer**(5)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of **torx**.

NAME

primexp − provide torx-explorer interface to torx primer

SYNOPSIS

primexp *primerprog* **[** *primerprog-args ...* **]**

DESCRIPTION

primexp provides access to a TorX primer via the TorX **torx-explorer**(5) interface. **primexp** starts the given *primerprog* program with the optional *primerprog-args* and communicates with it using the **torx-primer**(5) interface on the standard input and output of *primerprog.* **primexp** offers the TorX **torx-explorer**(5) interface on its own standard input and output.

This allows the use of CADP tools with TorX primers using **torx_open**(1). For example, the CADP **ocis**(1) simulator can be run on program *primerprog* as follows:

torx_open 'primexp *primerprog***' ocis**

BUGS

The environment variable **TORX_ROOT** is not supported.

NOTE

TorX used to contain a different (undocumented) program with the name **primexp**. That program was just a wrapper around **primer**(1). It did not add any functionality to TorX and was therefore removed, and the name is now reused.

SEE ALSO

torx-intro(1), **torx_open**(1), **torx-explorer**(5), **torx-primer**(5), **environ**(5)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of torx.

NAME

pui − simple primer user interface

SYNOPSIS

pui *primer* **[***primer-args* **...]**

DESCRIPTION

pui offers a simple (textual) user interface to a primer, that is just a bit nicer than the **torx-primer**(5) commands offered directly by a primer. It is mainly meant for debugging (or getting a better understanding of) a primer. When **pui** is started, it prints an overview of the commands that it recognizes.

SEE ALSO

torx-intro(1), **torx-primer**(5).

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of torx.

NAME

smileexp − use smile as symbolic explorer for LOTOS

SYNOPSIS

smileexp *cr-file*

DESCRIPTION

smileexp implements a symbolic explorer for LOTOS using the symbolic LOTOS simulator **smile.** It starts **smile** with the given common representation file *cr-file* and offers the TorX explorer-primer interface on standard input and output.

BUGS

The environment variable **TORX_ROOT** is not supported.

Because **smile** need X Windows to run, also **smileexp** can only be run when X Windows is running.

SEE ALSO

smile(1), **torx-intro**(1), **torx-explorer**(5), **environ**(5)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of torx.

NAME

tcp − tcp connection program

SYNOPSIS

tcp [-w] [*peerhost* **]** *peerport*

DESCRIPTION

tcp opens a connection to port number *peerport* of host *peerhost*, if given, or the local host otherwise. If takes more than 10 seconds to make the connection, **tcp** reports an error and exits. If this succeeds, it waits for input on standard input and messages that arrive over the connection, until end of file is detected on either one of these. If the **-w** flag is given, after end of file on standard input it will continue to wait for end of file on the connection.

Input arriving on standard input is send over the connection to the peer, and messages arriving over the connection are printed on standard output. When end of file is detected either on standard input or on the connection, **tcp** prints a diagnostic and exits.

Diagnostics are printed on standard error.

SEE ALSO

torx-intro(1), **udp**(1), **hexcontext**(1), **unhexify**(1)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of **torx**.

NAME

tmcs − tcp multicast service

SYNOPSIS

tmcs [-i] [-p *portnr* **]**

DESCRIPTION

tmcs implements a simple tcp multicasting service. It is used by **torx**(1) to keep multiple viewers (that visualize the test run) synchronized.

tmcs opens a socket at port *portnr*, if given, or the first free one. If this succeeds, it prints on standard output a line of the form

ipaddr hostname localportnr

where *localportnr* is the number of the opened port. Both *ipaddr* and *hostname* may have the value **0.0.0.0**. After this it waits for connections that are made to the socket, and for messages that arrive over these connections. Each message (usually: line of text) received over one connection is forwarded over all other connections. **tmcs** will exit when its last connection is closed. Initially it has no connections, so in order to make it exit (without explicitly killing it) at least one connection must be made to it.

When invoked with the -i flag, it will start a command interpreter. Otherwise, invoke it as for example

var=**‘tmcs|head -1‘**

to get the port number of tmcs in *var*.

COMMANDS

The following commands can be given on the standard input of **tmcs**, when it was invoked with the **-i** command line option. The command keyword (printed in capitals in this section) is recognized regardless of case (uppercase, lowercase, mixed).

PARTNERS

prints a list of partners (connections) on standard output

ADDRESS

prints line containing the local address to standard output (as done after startup)

DEBUGLVL [*nr***]**

sets debugging level. Debug level 0 disables debugging, for the other modes, see the source.

HELP prints an overview of accepted commands on standard output

SEE ALSO

torx-intro(1), **torx-logclient**(1), **tcp (1), torx (1), kill (1)**

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of **torx**.

NAME

torx-hostname − print hostname taken from network database

SYNOPSIS

torx-hostname

DESCRIPTION

torx-hostname prints the result of

gethostbyaddr(gethostbyname(hostname()), AF_INET)

Rephrased: it prints the name of the current host, as seen by the networking code.

torx-hostname is invoked by **adaptor**(1) to get the name of the current host, when an **ADDRESS** entry in a **torx-config**(4) configuration file contains the special host name **currenthost**. See the first example in **torx-config**(4).

This ugly hack is there, to avoid problems when we run torx in cygwin under windows, where the normal **hostname**(1) and **uname**(1) (when invoked with the **-n** option) commands return the hostname as set under windows, which may have no relation whatsoever with the current (networking) host name.

Essentially, this command is only there to make the conference protocol example work.

SEE ALSO

torx-intro(1), **adaptor**(1), **torx**(1), **torx-config**(4), **hostname**(1), **uname**(1)

BUGS

If, in the **adaptor**(1) and the encoding/decoding rules and configuration, we would use IP addresses instead of host names to identify machines, we would not have this problem.

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of **torx**.

NAME

torx-logclient − connect torx log monitor command to torx

SYNOPSIS

torx-logclient [-m *mcastid* **]** *host port* **--** *command* **[** *options* ... **]**

DESCRIPTION

Utility program to connect the given *command* (with its *options*) as log monitor to the **torx**(1) that is already running and waiting for log monitor connections at tcp port number *port* of host (or ip number) *host*.

When invoced with the **-m** *mcastid* command line option, it sets environment variable **TORXMCASTID** to *mcastid* before starting the given *command*.

Usually this command will be invoked by **torx**(1); it is not expected to be used directly by the user.

SEE ALSO

torx-intro(1), **torx**(1), **torx-log**(4)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of **torx**.

NAME

torx-mans − list TorX manual page file names

SYNOPSIS

torx-mans

DESCRIPTION

torx-mans prints the names of the manual pages of TorX (followed by a newline character) on standard output. (Actually, it prints the names of the files in the **man** subdirectory of the directory in which TorX is installed.) The **torx-intro**(1) man page will be the first in the list; the other pages will apear per section in alfabetical order. This list can then be used to produce a listing of all manual pages.

EXAMPLES

```
nroff -man ‘torx-mans‘
groff -man ‘torx-mans‘ > /tmp/torx-mans.ps
```

NOTE

Here, TorX refers to the distribution, not to the program; the program **torx**(1) will usually be installed in a subdirectory ‘‘bin’’ of the TorX installation directory.

SEE ALSO

torx-intro(1), **torx-root**(1), **environ**(5)

BUGS

The environment variable **TORX_ROOT** is not supported.

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of torx.

NAME

torx-querypr − query the TkGnats database

SYNOPSIS

torx-querypr

DESCRIPTION

torx-querypr is a (Tcl/Tk) utility to query the TkGnats problem database for the available problems or about the status of a submitted problem.

The **torx-sendpr** main window supports context sensitive online help for most fields shown on the window. By clicking on the text field, for example 'Class:', the online help will popup.

If you click the 'Do Query' button without selecting anything else you will get all problems available. To restrict this number you fill in one or several fields avaible on the window. This selects only those problem reports which comply with the selection you just entered.

SEE ALSO

torx-intro(1), **mkprimer**(1), **torx-primer**(5), **torx-adaptor**(5), **torx-config**(4), **torx-log**(4), **xtorx**(1).

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of **torx**.

NAME

torx-root − report TorX installation directory

SYNOPSIS

torx-root

DESCRIPTION

torx-root prints the name of the directory in which TorX is installed (followed by a newline character) on standard output. If the variable **TORX_ROOT** is present in the environment, and its value is not the empty string, its value will be printed. Otherwise, the directory name that was configured during the installation of TorX will be printed.

NOTE

Here, TorX refers to the distribution, not to the program; the program **torx**(1) will usually be installed in a subdirectory ‘‘bin’’ of the TorX installation directory.

SEE ALSO

torx-intro(1), **environ**(5)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of torx.

NAME

torx-sendpr − problem report utility

SYNOPSIS

torx-sendpr

DESCRIPTION

torx-sendpr is a utility to report problems, mistakes or requests for new features in TorX.

The main window contains a number of fields which have to be filled in by the user to report a problem. For a proper and quick diagnostic we advise you to fill in all the fields of the form.

Some fields are already filled in by **torx-sendpr** to lift the burden of the user of filling in all the fields, :-).

The **torx-sendpr** main window supports context sensitive online help of the most fields shown on the window. By clicking on the text field, for example 'Class:', the online help will popup.

The fields **Category**, **Submitter-Id**, **Originator** and are obligatory and need to be filled in.

The **Synopsis** field is used to describe the problem in one line, a short summary of the problem. This field is used in the ''Subject'' line of the emails send out by the problem-report tracking system.

The text fields **Description**, **How-To-Repeat** and **Fix** are used to get a full detailed description of the problem, how to repeat the problem, and how to fix the problem, if you know a solution.

SEE ALSO

torx-intro(1), **mkprimer**(1), **torx-primer**(5), **torx-adaptor**(5), **torx-config**(4), **torx-log**(4), **xtorx**(1).

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of **torx**.

NAME

torx − execute test on-the-fly

SYNOPSIS

torx [*options ...* **]** *configuration-file ...*
torx --help

DESCRIPTION

torx reads the given configuration file(s), starts the **primer** and **adaptor** specified in the configuration file(s) (see **torx-config**(4)), after which it either starts on-the-fly test generation and execution (when the *--depth* option was given) or prints its prompt (**tester>**) on standard output and waits for user commands from standard input. When **torx** detects an error, it prints a *Fail* verdict together with the the list of expected output events to standard output and exits. The output of the commands given to **torx** is written to standard output; diagnostics of the **primer** and **adaptor** are printed to standard error.

To be added: the command line arguments with which **primer** *and* **adaptor** *are invoked.*

A basic concept in **torx** is the *basic test action* or *event*. As representation of an event **torx** basically uses LOTOS-like notation (consisting of a *gate* name followed by zero or more *value-expressions*, where each value-expression is preceded by an exclamation mark (**!**)), with the extension that it may contain *variables* in the value-expressions. These variables have the following form: **var_***type***$***nr* where the **$***nr* part is optional. Here *type* is the type of the variable, and *nr* is a sequence number to make variable occurences unique in the *event*. The special event *suspension* is represented by **(Suspension)**.

Each test step that **torx** executes (as result of a command, or during execution in automatic mode) is reported on standard output. Each test step appears on a separate line, containing from left to right: the test step number, the *iokind* (**input** or **output**), an opening parenthesis (**(**) the *channel* name, a colon (**:**), the *pco* name, a closing parenthesis (**)**), a colon (**:**), followed by the *event*. For *suspension* events the *pco* name and preceding colon are omitted.

OPTIONS

Apart from the **--help** option, all **torx** options that can be set from the command line can also be set as configuration file option (see **torx-config**(4)). The command line options overrule settings specified in a configuration file.

The following command line options are supported:

--help print the version number and an overview of the command line options, and exit.

--batch start the tester in batch mode. In this mode no **adapter** is started, but a batch test in Aldebaran (.aut) format is generated on standard output, of depth as given with the **--depth** flag. By default, this will have the format of a tree (i.e. there will be no cycles) *This is rather new and not*

--batch-automaton
generate the batch test in the form of an automaton (may contain cycles). Note: this flag does not imply the **--batch flag** which must be separately given.

--depth *nr* (config: MAXSTEPS) start the tester in automatic mode, try to execute *nr* of test events, and exit when done. The automatic execution can be interrupted by giving a **stop** command, which causes a prompt to be printed.

--no-depth (config: MAXSTEPS) start **torx** in manual mode, with unlimited depth (this is the default)

--seed *nr* (config: SEED) use *nr* as seed for random number generator(s)

--no-seed (config: SEED) use random seed (based on current time) (this is the default)

--log *file* (config: LOGFILE) write log to *file.* If *file* already exists **torx** will extend the file name to be unique, by extending it with the string **.~***n***~** where *n* is the smallest number (from 0) that makes the file name unique.

--no-log (config: LOGFILE) do not write a log-file (this is the default)

--logmon *command*
(config: LOGMON) start torx log monitoring *command* as background process that can continue to run even after torx itself has exited, and write log to its standard input. Multiple **--logmon** command line options may be given, to start multiple commands (e.g. to use multiple viewers). The connection to *command* is made via the utility program **torx-logclient**(1).

--no-logmon (config: LOGMON) do not start a log monitoring command (this is the default)

--trace *file* (config: TRACEFILE) read trace from *file*.

--no-trace (config: TRACEFILE) do not use a trace-file (this is the default)

COMMANDS

The commands of **torx** are grouped in three sections: 1) general ones, 2) commands that give information about the current test execution status, without changing the current execution 'state', and 3) commands that execute a (sequence of) testing step(s).

general

help print an overview of the recognized commands.

quit (clean up and) exit the tester

informational

The following commands only print information, without doing a test step. These commands do not cause a state change in **primer** or **adaptor**.

path print the path (the events done from start till now)

menu print the menu (distinquishes inputs from outputs). Each menu-element is printed on a separate line, containing from left to right: the *iokind* (**input** or **output**), followed by a colon (**:**), the *channel* name between parentheses, followed by the *event*.

trace print the current event from the trace that we follow

state print the state(vector) (internal format of the **primer**)

menusize print the size of the menu *(this command will probably be deprecated)*

statesize print the size of the state(vector) (internal format) *(this command will probably be deprecated)*

execution

The following commands (may) cause the execution of a testing step. Some commands can only be executed if the (parenthesized) condition at the start of their explanation below holds. If **torx** is unable to execute a command, it will print an error message to standard output, and issue a new prompt. Note: in the commands **next**, **step**, **input**, and **output**, the parentheses around the channel name argument are part of the command syntax and can not be omitted.

io select randomly input or output, to be used in next step

next (*chan***)** *input-event*
(*input-event* isin menu) do one step using *input-event*

next do one step (using the result of last **io** command)

step (*chan***)** *input-event*
synonym for **next (***chan***)** *event*

step synonym for **next**

input (*chan***)** *event*
(*event* isin menu) do one input step using *event*

input *event* (*event* isin menu) do one input step using *event*

input (*chan***)**	(menu of *chan* is non-empty) do one input step, from channel *chan*
input	(input menu is non-empty) do one input step, from randomly chosen channel
output (*chan***)**	do one output step, from channel *chan*
output	do one output step, from randomly chosen channel
auto	switch to automatic mode: do steps, randomly choosing in- and outputs until end of test, or until interrupted by the **stop** command
auto *nr*	as **auto**, but do at most *nr* steps
usetrace	use the current trace event for next step
autotrace	switch to automatic mode: do steps, following the trace, until end of test, or end of trace, or until interrupted by the **stop** command
autotrace *nr*	as **autotrace**, but do at most *nr* steps
stop	interrupt the **auto** or **autotrace** command and print a prompt.

EXAMPLE

Below we show a **sh**(1) shell script that demonstrates how **torx** can be used in 'batch' mode to repeatedly execute tests upto a given number of test steps, for a given set of mutants, using a different random number seed in each execution run. We assume here that the mutant can be selected by setting the variable MUTANT in the environment. The command used to invoke **torx** is split-up over several lines for clarity.

```
#!/bin/sh
first=1
beyond=1000
depth=1000000
mutants="111 222 333 444 555 666 777 888 999 000"
export MUTANT

i=$first
while test $i -lt $beyond
do
    for m in $mutants
    do
        MUTANT=$m
        torx --depth $depth \
            --seed $i \
            --log testloop.$i.$m.log \
             config.if \
            > testloop.$i.$m.out 2>&1
        sleep 60
    done
    i=‘expr $i + 1‘
done
```

The assumption behind this script is that the implementation under test will be started by (a shell script started by) **torx** (actually: by the torx **adaptor**) which means that it (the implementation resp. the shell script) will see variable MUTANT in its environment, and act on it.

SEE ALSO

torx-intro(1), **mkprimer**(1), **torx-logclient**(1), **torx-primer**(5), **torx-adaptor**(5), **torx-config**(4), **torx-log**(4), **xtorx**(1), **sh**(1)

DIAGNOSTICS

The diagnostic messages of **primer** and **adaptor** are passed on to the standard error. If an error is encountered in an command given to **torx**, an appropriate error message is given (on standard output), and a new prompt is printed. The error messages should be self-explanatory.

RETURN-VALUE

In case of normal termination (whether or not an error is found) **torx** always returns with a 0 exit status. A non-zero exit status will only be given when an (unforeseen?) internal or external error makes normal termination impossible.

BUGS

The diagnostics of **primer** and **adaptor** appear interspersed with the output of **torx**; giving **torx** an empty command (just press return) prints a new prompt.

Using a syntactically wrong *event* as argument to a command will cause **torx** to exit.

The **--depth** flag should be treated in a slightly different way: after automatically doing the test steps required by this flag, **torx** should execute commands given on standard input, until end-of-file on standard input, or until a **quit** command is given. (however, this would require batch scripts to be updated, to invoke **torx** with standard input redirected from /dev/null)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of **torx**.

NAME

torx_open − run open/caesar tool on torx explorer program

SYNOPSIS

torx_open *explorerprog-with-arguments* **[** *c-options* **]** *tool* **[.a|.o|.c] [** *tool-options* **]**

DESCRIPTION

torx_open applies Open/Caesar tool *tool* to the (state space made accessable by) TorX explorer (with optional arguments) *explorerprog-with-arguments*. The *explorerprog* is accessed via its standard input and output using the **torx-explorer**(5) interface. If *explorerprog* needs to be invoked with *arguments* then the whole string *explorerprog-with-arguments* must be put between quotes in the invocation of **torx_open** such that to **torx_open** it appears as a single command line argument, as shown below.

This allows the use of CADP tools with TorX explorers. For example, the CADP **ocis**(1) simulator can be run on program **jararaca**(1) with specification *spec***.jrrc** as follows:

torx_open 'jararaca -s -e *spec***.jrrc' ocis**

This allows the use of CADP tools with TorX primers using **primexp**(1). For example, the CADP **ocis**(1) simulator can be run on primer program *primerprog* (with arguments *args* ...) as follows:

torx_open 'primexp *primerprog args* ...**' ocis**

The automaton of the tester of *primerprog* (with arguments *args* ...) can be generated in **bcg** format as file *tester***.bcg** as follows:

torx_open 'primexp *primerprog args* ...**' generator** *tester***.bcg**

BUGS

The environment variable **TORX_ROOT** is not supported.

NOTE

TorX used to contain a different (undocumented) program with the name **primexp**. That program was just a wrapper around **primer**(1). It did not add any functionality to TorX and was therefore removed, and the name is now reused.

SEE ALSO

torx-intro(1), **primexp**(1), **torx-explorer**(5), **torx-primer**(5), **bcg**(LOCAL), **environ**(5)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of torx.

NAME

udp − udp connection program

SYNOPSIS

udp [-debug [*nr* **]] [-port** *portnr* **] [-[no]printdata] [-[no]printdatahex] [-[no]delay]**

DESCRIPTION

udp opens a socket at port *portnr*, if given, or the first free one. If this succeeds, it prints on standard output a line of the form

HOSTPORT *localportnr*

where *localportnr* is the number of the opened port, after which it waits for commands on standard input and messages that arrive on the socket, until it detects end of file on its standard input, after which it closes the socket and exits. The recognized commands are discussed below. When a message arrives on the socket, it outputs on standard output a line of the form

RECV *peerhost peeraddr peerport data*

if printing of data is enabled, and/or, if printing of data in hexadecimal form is enabled, a line of the form

RECVHEX *peerhost peeraddr peerport datahex*

In these lines *peerhost*, *peeraddr* and *peerport* are the hostname, the IP number and the port number of the peer, and *data* and *datahex* are the contents of the message, as received resp. in hexadecimal form. By default, output in hexadecimal format is enabled, and output in ''normal'' format is disabled. This can be changed using the command line options **-[no]printdatahex** and **-[no]printdata** and with corresponding commands, as discussed below.

The **-delay** option enables randomly chosen 1-second delays (sleeps) between receipt of a message on standard input and forwarding of the message over the socket. If messages arrive at the socket (from over the network) during the delay (sleep), also they suffer from the delay. However, the FIFO behaviour of the program is untouched. This option is meant to (crudely) simulate the behaviour of buffering channels, such that if there are multiple channels we may see random interleavings of the messages on the different channels.

The **-debug [***nr***]** option opens a hardcoded pseudo terminal (pty) on which debugging information is printed. The amount of information printed depends on the numeric debug mode given (see the source).

COMMANDS

The following commands can be given on the standard input of **udp**. The command keyword (printed in capitals in this section) is recognized regardless of case (uppercase, lowercase, mixed).

SENDHEX *peerhost peerport datahex*
send the data (given as hexadecimal string) to peer at port *peerport* of host *peerhost*.

LOCALADDR
print a **HOSTPORT** *localportnr* line to standard output (as done after startup)

PRINTDATA
enable printing of data ''as received'', in the form of **RECV** lines

NOPRINTDATA
disable printing of data ''as received'', in the form of **RECV** lines

PRINTDATAHEX
enable printing of data in hexadecimal form, in the form of **RECVHEX** lines

NOPRINTDATAHEX
disable printing of data in hexadecimal form, in the form of **RECVHEX** lines

DEBUG [*nr***]**
set debugging mode. Debugging mode 0 disables debugging, for the other modes, see the source.

NODEBUG
disable debugging

BUGS

For the **-[no]delay** command line option there is no corresponding command that can be given on standard input.

There is no option to set the seed of the random number generator used for the **-[no]delay** command line option.

SEE ALSO

torx-intro(1), **tcp**(1), **hexcontext**(1), **unhexify**(1)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of **torx**.

NAME

unhexify − translate from hexadecimal to ascii

SYNOPSIS

unhexify

DESCRIPTION

unhexify reads hexadecimal strings from standard input and writes them in ascii form to standard output.

SEE ALSO

torx-intro(1), **tcp**(1), **udp**(1), **hexcontext**(1)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of **torx**.

NAME

xtorx-showmsc − show a TorX run log as Message Sequence Chart

SYNOPSIS

xtorx-showmsc

DESCRIPTION

The **xtorx-showmsc** program is invoked by **xtorx**(1) whenever the user asks **xtorx**(1) to show an MSC. The TorX run log is read from standard input.

The **xtorx-showmsc** program uses **log2msc**(1) and **mscviewer**(1) (via **Bmsc**(1)) to do its job. The command line arguments given to **xtorx-showmsc** are passed on to **Bmsc**(1). This is used by **xtorx**(1) to pass a **-r** option to **Bmsc**(1).

SEE ALSO

torx-intro(1), **log2msc**(1), **mscviewer**(1), **torx-log**(4)

NAME

xtorx-showspec − show a specification file

SYNOPSIS

xtorx-showspec *specification-file*

DESCRIPTION

The **xtorx-showspec (1)** program is invoked by **xtorx**(1) whenever the user asks **xtorx**(1) to show the specification of a primer or mutant (IUT).

The **xtorx-showspec**(1) program takes a single argument: the file name of the specification that is to be shown.

xtorx-showspec(1) tries to identify the specification file contents from the file suffix, in order to be able to show the specification in a nicer way than just as text. Currently, the following file suffixes are recognized:

.bcg Binary Coded Graph file; **xtorx-showspec**(1) shows it with **bcg_edit**(1)

.aut Automaton file; **xtorx-showspec**(1) will check if there is a corresponding **.bcg** file, and, if so, show it with **bcg_edit**(1)

Other files are shown using the command: **xterm -e less** *file*

The user is expected to extend (adapt, replace) **xtorx-showspec**(1) when more advanced behaviour is needed.

BUGS

No check is done whether or not the viewing tools used are present.

SEE ALSO

torx-intro(1), **bcg_edit**(1), **less**(1), **xterm**(1).

NAME

xtorx − gui for the torx on-the-fly tester

SYNOPSIS

xtorx [*options ...* **]** *configuration-file ...*

DESCRIPTION

xtorx(1) reads the configuration file(s), if given, and opens a window that contains the following from top to bottom: a *Menu* bar, a *Button* bar, an *Executed test steps* pane, a *Spec* pane, a *Verdict* bar and a *Messages* pane. The window manager may add a *Window Title*, usually at the top.

Many buttons and menu entries are enabled or disabled depending on the state in which **xtorx**(1) is. For example, buttons and menu entries that set test execution parameters are only enabled while no test execution is taking place, i.e. either when **xtorx**(1) has just been started, or after a test execution has come to an end. A test exection can come to an end in two ways: 1) when an error is found, the test execution run ends automatically, and 2) the user can end a running test execution by clicking the **Stop** button in the *Button* bar.

The *Window Title* displays information about the current configuration, primers, and mutants files, and about currently selected primer and mutant. This same information is also added to the *Messages* pane at the start and end of each test run.

The *Menu* bar contains a *File* menu to deal with files (and to clean the *Messages* pane, and to exit), a *Preferences* menu, a *Primers* and a *Mutants* menu to choose a primer or a mutant (see below), a *View* menu to view a primer or a mutant source file (when available and/or configured), a *Tools* menu to enable or disable visualiation tools for the test run, and a *Help* menu. The *Primers*, *Mutants*, *View*, and *Tools* menus are context dependend; their menu-entries depend on the configuration file loaded and the primers and mutants files (again, see below) that are loaded.

The *Button* bar contains the buttons to start and stop a test execution run, and to switch between manual (single-stepping) and automatic test execution mode. When a test execution has been started, **xtorx**(1) is by default in manual mode. In manual mode, the user is in full control, and test steps can be done using the buttons from the *Spec* pane. In automatic mode, **xtorx**(1) controls the testing, by repeatedly executing test steps, making its own decisions about observing and stimulating. The automatic mode can be enabled by clicking the **Auto** radio-button in the **Mode** field of the *Button* bar, *after which automatic testing will immediately begin*! Clicking the **Manual** button in the **Mode** field switches back to manual mode. If a positive integer value has been entered in the **Steps** box in the *Button* bar when the automatic mode is enabled, **xtorx**(1) will try to execute the specified number of test steps, after which it will switch back to manual mode.

The *Executed test steps* pane contains the trace (steps executed so far) of the current execution run. Each test step that is executed is appended here, preceded by the test step number.

The *Spec* pane contains the test actions for the current state (possible inputs, or stimuli, and expected outputs, or observations), together with buttons to stimulate (**Random Input**, **Selected Input**) to observe (**Output**), or to do an arbitrary test step, where **xtorx** decides between stimulating and observing (**Random**). Double-clicking on an input action will execute this action. Additionally, if a *trace* of a previous test execution run is replayed, the 'current' action of the trace will be shown, together with a button (**Use Trace**) to re-execute this action.

The *Verdict* bar displays the status of testing process. If the pane has color red it means **failure**. If the color is green it means **test purpose was hit** or **trace is passed** (when an attempt was done to replay a log as a trace). If the color is orange it means **test purpose was missed** or **inconclusive** (was unable to completely replay the given trace).

The *Messages* pane is mostly used for logging and debugging purposes. It will contain the output produced by the SUT (System Under Test), if any, together with debugging messages of various TorX tool components. The messages are grouped together for each test execution run by lines (one above and one below all messages for the test run) that start with a triangular button, followed by the configuration parameters for

that test run. Depending on the version of the **wish** (tcl/tk) interpreter that is being used (xtorx is a tcl/tk script), it is possible to hide and show the messages for each individual test execution run by clicking the first (usual left) mouse button on the triangular button. Clicking the second (usual middle) mouse button on a triangular button scrolls the window to make its ''partner'' visible.

DETAILS

The main window of **xtorx**(1) is divided vertically in three main panes. The top pane consists of the *Title* bar, the *Button* bar and the *Executed test steps* pane. The middle pane contains the *Spec* pane, and the *Verdict* bar. The bottom pane justs consists of the *Messages* pane.

The relative size of these panes can be changed by clicking and dragging the first (usally left) mouse button in the small square resize buttons that appear on top of the pane borders. How the panes are resized can be influenced with the **Preferences -> OpaquePaneResize** setting.

WINDOW TITLE

The window title displays the current configuration, primers and mutants file(s), and the selected primer and mutant (if any). The window title is added by the window manager; not all window managers do add one, though.

MENU BAR

The *menu* bar contain all required operations for testing. The operations are divided into the categories **File**, **Preferences**, **Primers**, **Mutants**, **View**, **Tools**, and **Help**. We will discuss each of them in its own section.

FILE MENU

The *File* menu contains all operations which involve operations with files.

Open Configuration... Open a configuration file. If the **Init Gui From Config** and **Reset Gui From Config** toggle buttons in the **Preferences** menu are set, a number of settings and menu entries will be initialised from the information in the configuration file. The following items will initialised (the corresponding **torx-config**(4) keywords appear between parentheses): log file (**LOGFILE**), trace file (**TRACEFILE**), seed (**SEED**), primers file and *Primers* menu (**PRIMERS**), selected primer (**PRIMER**), guides file and *Guides* menu (**GUIDES**), selected mutant (**GUIDE**), mutants file and *Mutants* menu (**MUTANTS**), selected mutant (**MUTANT**), *View* menu (**SPECSOURCE** and **IUTSOURCE**), and visualization *Tools* menu (**LOGMON**).

Open Trace... Choose a trace file that is to be used in the next test execution run.

Close Trace Do not use a (the previously) chosen trace file in the next test execution run.

Open Primers... Select a primer configuration file, and if this is succesful, update the *Primers* menu, and enable the *Primers* menu button and its menu entries.

Close Primers Clear the *Primers* menu, and disable the *Primers* menu button, and deselect the primer, if one was chosen from the *Primers* menu.

Reopen Primers Reloads the primer configuration file, to refresh the *Primers* menu when the primer configuration file is changed.

Open Mutants... Select a mutant configuration file, and if this is succesful, update the *Mutants* menu, and enable the *Mutants* menu button and its menu entries.

Close Mutants Clear the *Mutants* menu, and disable the *Mutants* menu button, and deselect the mutant, if one was chosen from the *Mutants* menu.

Reopen Mutants Reloads the mutant configuration file, to refresh the *Mutants* menu when the mutant configuration file is changed.

Set Logfile Name... Select the logfile name that is to be used in the next test execution run.

Unset Logfile Name... Clears the logfile name, i.e. do not create a log in the next test execution run.

Clear Messages Clear the *Messages* pane.

Save Messages to File...
Save the messages in the *Messages* pane to the specified file.

Exit Exit the program. This will, however, not exit ''independent'' visualization tools, even if they are started from **xtorx**(1).

PREFERENCES MENU

In the *Preferences* menu the following options can be enabled or disabled.

Start MSC If this option is set when the **Start** button is pressed, the **mscviewer**(1) will be started via **xtorx-showmsc**(1) to visualize the test run. (default value: enabled)

Set MSC Window Reuse Button
This option sets the initial value of the **Reuse** button of the **mscviewer**(1). (default value: enabled)

Use Bigger Fonts Change all fonts to a different one (that should be bigger). This option is meant for demo purposes. (default value: disabled)

Init Gui From Config If this option is set, settings indicated in the description of the **Open Configuration...** command from the **File** menu will be updated if corresponding entries are present in the configuration file that is opened. (default value: enabled)

Reset Gui From Config
If this option is set, settings indicated in the description of the **Open Configuration...** command from the **File** menu will be reset, unless the **Init Gui From Config** option is enabled, and the configuration file contains corresponding entries. (default value: enabled)

OpaquePaneResize An unpronounceable option with means that the contents of the panes are moved as well during resizing of panes (while the mouse button is pressed; otherwise, only the pane separator lines are moved, and actual resizing of the pane contents is only done once the mouse button is released). (default value: disabled)

Enable Trace Support Add a **Use Trace** button and a trace text field to the *Spec* pane, and an **AutoTrace** button next to the **Auto** button in the *Button* bar (by default, these are hidden to save screen space). It may be necessary to resize the **xtorx**(1) window, to make it wider, to make these buttons visible. (default value: disabled)

Enable Instantiation Support
Add a **Use Instantiation** button and an instantiation text entry field to the *Spec* pane (by default, these are hidden to save screen space). It may be necessary to resize the **xtorx**(1) window, to make it wider, to make these buttons visible. (default value: disabled)

Show Message Buttons
Add a **Clear Messages** button and a **Save Messages to File...** button to the end of the *Messages* pane (by default, these buttons are hidden to save screen space). (default value: disabled)

Show Refresh Buttons Add a **Refresh** button and a **auto-** toggle button (by default: enabled) to the end of the *Button* bar. If the **auto-** toggle button is disabled, the *Spec* pane (in particular: the lists of possible inputs and expected outputs) will not automatically be updated after a test step, but only after the **Refresh** button has been pressed (by default, these buttons are hidden to save screen space). (default value: disabled)

PRIMERS MENU

The *Primers* menu gives a list of primers of which one can be selected. This list is generated from the primers file that is loaded via the **File -> Open Primers** menu entry (or automatically via a **PRIMERS** configuration file entry, as described above). If a **PRIMER** entry is present in the configuration file, it will be used to set the default. Otherwise, if an entry **none** is present in the *Primers* menu, it will be the default.

Otherwise, initially none of the primers will be selected.

Depending on the details of a particular configuration, the test execution configuration parameters may be incomplete when no primer is selected, which may cause test execution runs to fail even before a test step has been done. So, if a test execution run fails in this way, make sure to check if you have forgotten to select a primer.

A side effect of selecting a primer in the menu can be that additional, primer-specific, configuration file(s) are loaded, like for example the **.torx** files generated by **mkprimer**(1). Such a file may contain, for example, a primer-specific **SPECSOURCE** entry.

Of course, what exactely happens when a primer is selected depends completely on the contents of the primers file that was loaded. For a description of what can be specified in a primers file, see **xtorx-extension**(n). Examples of primers files can be found in the torx-examples distribution.

GUIDES MENU

The *Guides* menu gives a list of guides of which one can be selected. This list is generated from the guides file that is loaded via the **File -> Open Guides** menu entry (or automatically via a **GUIDES** configuration file entry, as described above). If a **GUIDE** entry is present in the configuration file, it will be used to set the default. Otherwise, if an entry **none** is present in the *Guides* menu, it will be the default. Otherwise, initially none of the guides will be selected.

Depending on the details of a particular configuration, the test execution configuration parameters may be incomplete when no guide is selected, which may cause test execution runs to fail even before a test step has been done. So, if a test execution run fails in this way, make sure to check if you have forgotten to select a guide.

A side effect of selecting a guide in the menu can be that additional, guide-specific, configuration file(s) are loaded, like for example the **.torx** files generated by **mkprimer**(1). Such a file may contain, for example, a guide-specific **GUIDESOURCE** entry.

Of course, what exactely happens when a guide is selected depends completely on the contents of the guides file that was loaded. For a description of what can be specified in a guides file, see **xtorx-extension**(n). Examples of guides files can be found in the torx-examples distribution.

MUTANTS MENU

The *Mutants* menu gives a list of mutants of which one can be selected. This list is generated from the mutants file that is loaded via the **File -> Open Mutants** menu entry (or automatically via a **MUTANTS** configuration file entry, as described above). If a **MUTANT** entry is present in the configuration file, it will be used to set the default. Otherwise, if an entry **none** is present in the *Mutants* menu, it will be the default. Otherwise, initially none of the mutants will be selected.

Depending on the details of a particular configuration, the test execution configuration parameters may be incomplete when no mutant is selected, which may cause test execution runs to fail even before a test step has been done. So, if a test execution run fails in this way, make sure to check if you have forgotten to select a mutant.

A side effect of selecting a mutant in the menu can be that additional, mutant-specific, configuration file(s) are loaded, like for example the **.torx** files generated by **mkprimer**(1). Such a file may contain, for example, a primer-specific **IUTSOURCE** entry.

Of course, what exactely happens when a mutant is selected depends completely on the contents of the mutants file that was loaded. For a description of what can be specified in a mutants file, see **xtorx-extension**(n). Examples of mutants files can be found in the torx-examples distribution.

VIEW MENU

The *View* menu contains two entries to view the source of the specification resp. the implementation (or mutant).

Primer source show the source file of the primer using **xtorx-showspec**(1). This button is only enabled if the **torx-config**(4) configuration file contained an **SPECSOURCE** (or,

deprecated, **SOURCESPEC**) entry.
As mentioned above in the section about the *Primers* menu, it is possible to set up a primers file in such a way that selecting a primer from the menu causes an additional primer-specific configuration file to be loaded, that defines (o.a.) a **SPECSOURCE** entry for the selected primer.

Mutant source show the source file of the mutant using **xtorx-showspec**(1). This button is only enabled if the **torx-config**(4) configuration file contained an **IUTSOURCE** (or, deprecated, **SOURCEIUT**) entry for the selected primer.
As mentioned above in the section about the *Mutants* menu, it is possible to set up a mutants file in such a way that selecting a mutant from the menu causes an additional mutant-specific configuration file to be loaded, that defines (o.a.) a **IUTSOURCE** entry for the selected mutant.

TOOLS MENU

The *Tools* menu contains a list of toggle buttons to enable and disable visualization tools, or more generally, tools that work on the **torx-log**(4) log file of a test execution run. This list is generated from **LOGMON** entries in the **torx-config**(4) configuration file(s) when a configuration file is loaded (or from a primer- or mutant-specific configuration file when a primer or mutant is selected). By default, all entries in the list are enabled.

HELP MENU

The *Help* menu contains the following entries to get more information and browse (query) and submit problem reports.

Help on TorX (to be implemented)

About TorX displays a dialog box containing copyright and contact information.

On Version (to be implemented)

Query Problem Reports (using tkgnats)...
Invokes **torx-querypr**(1) to open a **tkgnats**(1) window to query problem reports.

Query Problem Reports (using web-browser)...
(to be implemented)

Report Problem (using tkgnats)...
Invokes **torx-sendpr**(1) to open a **tkgnats**(1) window to submit problem reports. This can be used to report problems about the tool, inconsistencies, etc. to us.

Report Problem (using web-browser)...
(to be implemented)

BUTTON BAR

The *Button* bar contains buttons and text entry fields to control the execution of a test run:

Start button Start a test execution run. This means that the **torx**(1) program is started under the control of **xtorx**(1); **xtorx**(1) is merely a graphical wrapper around **torx**(1).

Stop button Stop the test execution run, by asking the **torx**(1) program that was started via the **Start** button to quit (exit).

Seed field here the seed can be entered for the random number generator that **torx**(1) and the other TorX components will use. If the user has not filled in this entry when the **Start** button is pressed, **xtorx**(1) will itself randomly choose a value and fill in the field. By default, this field is empty. Once filled, **xtorx**(1) will not overwrite it.

Manual mode button swiches to the manual mode of on-the-fly testing. In this mode the user is in complete control, and can use the buttons in the *Spec* pane. This is the default mode, that is entered every time that the **Start** button is pressed,

Auto mode button switches to automatic mode of on-the-fly testing. In this mode **xtorx**(1) will make

all decisions; it is like a user who continously presses the **Random** button in the *Spec* pane. If the *Steps* field is filled with an integer value when the **Auto** button is pressed, only the specified number of steps will be done, after which **xtorx**(1) will switch back to manual mode. When an error is found while running in Auto mode, the test run is ended. *Warning*: when this mode is selected, **xtorx**(1) will *immediately* start (continue) running the test.

AutoTrace mode button

This mode is only available when a trace file is loaded, and this button is only visible when **Preferences -> Enable Trace Support** is selected.
switches to the automatic trace mode of on-the-fly testing. This mode is like a user who continously presses the **Use Trace** button in the *Spec* pane. If the *Steps* field is filled with an integer value when the **AutoTrace** button is pressed, only the specified number of steps will be done, after which **xtorx**(1) will switch back to manual mode. When an error is found while running in AutoTrace mode, the test run is ended. *Warning*: when this mode is selected, **xtorx**(1) will *immediately* start (continue) running the test.

Steps field — this field can be used to specify the number of test steps that should be done (at most) when the **Auto** or **AutoTrace** button is pressed (if an error is found before the specified number of test steps is done, the test will be ended).

auto- toggle button — This button is only visible if the **Preferences-> Show Refresh Buttons** setting is enabled.
When the **auto-** toggle button is enabled, the lists of inputs and outputs events in the *Spec* pane are automatically updated after each test step. Otherwise, these lists are only updated after the **Refresh** button is pressed. (default value: enabled)

Refresh button — This button is only visible if the **Preferences-> Show Refresh Buttons** setting is enabled.
Update the lists of inputs and outputs events in the *Spec* pane.

EXECUTED TEST STEPS PANE

The *Executed test steps* pane displays a trace of the test steps which have been executed: inputs that have been sent as simulus to, or outputs that have been received as observation from the SUT. Each test step is preceded by the test step number. The test step that is currently being visualized (or highlighted) by the tools that were enabled in the *Tools* menu when the test run was started has a yellow background. It is possible to change the 'currently visualized' test step in **xtorx** by clicking the third mouse button in a test step, or dragging the mouse over the test steps with the third button down. Note that also each of the individual visualization tools can be used to change the 'currently visualized' test step.

SPEC PANE

The *Spec* pane has two lists next to each other, each in its own sub-pane: a list of *Inputs* and a list of *Outputs*. During a test execution run, the *Inputs* list contains the possible input events (possible stimuli that can be sent to the SUT) for the current state, and the *Outputs* list contains the expected output events (observations that are expected from the SUT). If a test execution run is ended because an error is found, the *Inputs* list will be empty, and the *Outputs* list will contain the expected observations. At the same time, the last event in the *Executed test steps* pane is the last (erroneous, conflicting, invalid) event that was received from the SUT.

The relative horizontal sizes of these sub-panes can be changed by clicking and dragging the first (usally left) mouse button in the small square resize buttons that appears on top of the pane border. How the panes are resized can be influenced with the **Preferences -> OpaquePaneResize** setting.

Under the *Inputs* and *Outputs* lists the following buttons are present to control individual test steps during the test execution run:

Selected Input — send the input event that is selected in the *Inputs* list as stimulus to the SUT.

Random Input let the program randomly select an input event and send it to the SUT.

Random let the program randomly decide between stimulating and observing, and then, depending on the result of this ‘‘decision’’, behave as if the **Random Input** resp. the **Output** button was pressed.

Output get an observation from the SUT, and check if it is in the list of expected output events.

If instantiation support was enabled (see the **Preferences -> Enable Instantiation Support** button above), under the buttons mentioned above another button and a text entry field are added. When an event is selected in the *Inputs* list, it is copied to the text entry field, where it can be edited. The copied event may contain a predicate (enclosed between square brackets ‘‘[’’ and ‘‘]’’) that consists of one or more constraints on the values of the variables in the event. The constraints are separated by ‘‘;’’, there may be an optional ‘‘;’’ after the last constraint, before the closing ‘‘]’’. When editing the event, the predicate may be deleted; the constraints can be used as inspiration when choosing values for the variables. Note that it is not mandatory to choose values for all variables -- it is also possible to change the constraints (or add new ones) to reduce the number of possible values for the variables, and let ‘‘the system’’ (an **instantiator**(1)?) then come up with a single value. The button can be used to try to apply the edited event, in a similar way as the other buttons in the *Spec* pane:

Use Instantiation try to apply the action shown in the instantation field as the next input event, i.e. check if the instantiation is a valid one, and, if so, use it as a stimulus and send it to the SUT.

If trace support was enabled (see the **Preferences -> Enable Trace Support** button above), under the buttons mentioned above another button and a text field are added. The text field is used to display the subsequent event from the trace, and the button can be used to apply it, in a similar way as the other buttons in the *Spec* pane:

Use Trace try to apply the action shown in the trace field as the next input or output event, i.e. if the action in the trace field is an input event, use it as a stimulus and send it to the SUT, and otherwise, if the action in the trace field is an output event, get an observation from the SUT and check if it is valid (in the list of output events), and if it is identical to the action shown in the trace field.

VERDICT BAR

This bar contains a text field in which the verdict will be given at the end of a test execution run. Depending on the particular verdict, the color if the bar will be changed.
The pane is colored red when an error was found (usually this means that an observation received from the SUT was not in the list of expected output events) (in tradional terms: **fail**).
The pane is colored green when a **test purpose was hit** or, when an attempt was done to replay a log as a trace, the end of the trace was succesfully reached without finding an error (in tradional terms: **pass**).
The pane is colored orange when a **test purpose was missed** or, when an attempt was done to replay a log as a trace, the actual test run deviated from the trace, but without finding an error (in tradional terms: **inconclusive**).

MESSAGES PANE

The *Messages* pane is mostly used for logging and debugging purposes. It will contain the output produced by the SUT (System Under Test), if any, together with debugging messages of various TorX tool components. The messages are grouped together for each test execution run by lines (one above and one below all messages for the test run) that start with a triangular button, followed by the configuration parameters for that test run. Depending on the version of the **wish** (tcl/tk) interpreter that is being used (xtorx is a tcl/tk script), it is possible to hide and show the messages for each individual test execution run by clicking the first (usual left) mouse button on the triangular button. Clicking the second (usual middle) mouse button on a triangular button scrolls the window to make its ‘‘partner’’ visible.

X DEFAULTS

Currently, it seems, there are no X defaults setting to be used for configuration.

FILES

∗.**if**	**torx**(1) **torx-config**(4) configuration files
∗.**primers**	primers files
∗.**mutants**	mutants files

SEE ALSO

torx-intro(1), **torx**(1), **torx-config**(4), **torx-log**(4), **xtorx-extension**(n), **xtorx-showspec**(1), **xtorx-showmsc**(1), **anifsm**(1), **aniwait**(1), **jararacy**(1), **mscviewer**(1), **torx-querypr**(1), **torx-sendpr**(1)

BUGS

It is not possible to instantiate variables in events by hand when giving the **Selected Input** command.

Sometimes, during a test execution run, **xtorx**(1) may get into a state in which all ‘‘useful’’ buttons are disabled. In such a case, the only sensible thing to do is to use the **File -> Exit** button to exit, and use the **ps**(1) and **kill**(1) command to check for, and kill, runaway TorX processes.

Occasionally it happens that when **xtorx** is started, in the window only the *Messages* pane is visible. The solution is to use the small square resize button, half of which is hidden under the menu bar, to resize the panes. When the *Spec* pane is visible again, it is very well possible that there only the Outputs are visible (the Inputs are hidden as well). The solution is here as well to use the small square resize button that is now completely at the left of the window, to resize the Outputs pane to make the Inputs visible.

A number of commands in the *Help* menu have remained unimplemented too long.

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of **xtorx**.

NAME

torx-config − configuration file for torx tester

DESCRIPTION

A **torx**(1) configuration file contains all information that is needed for a test execution run. Some of the the configation file entries can be overriden with command line options of **torx**(1), as will be indicated in the ENTRIES section below. Currently, the configuration file contains information that is used by different components of torx: it contains information for the PRIMER, for the DRIVER, and for the ADAPTER. In some cases, the same information is used by more than a single component.

For most settings, only one entry should appear in the configuration file(s) of **torx**(1). The settings for which multiple entry lines are allowed, are prefixed with an asterisk (∗) in the ENTRIES section below.

Currently, the configuration file has the following format. Each configuration entry consists of a single line. Such a line starts with a keyword that indicates the kind of entry. Empty lines and lines containing only whitespace are ignored. Comment lines start with optional whitespace followed by a hash symbol (#). All relative file- and directory names are interpreted relative to the (location of) the configuation file in which they appear, *not* relative to the location from which **torx**(1) is started. If an argument of an entry contains whitespace, it should be enclosed between curly brackets ‘‘{’’ and ‘‘}’’ (because we treat each entry line as a **tcl** list). See the EXAMPLES below to get an idea of how these rules work out in practice.

ENTRIES

The configration file entries are grouped in the following sections: 1) general ones, 2) execution-run related, 3) PRIMER related, 4) ADAPTER related, 5) INSTANTIATOR related, 6) internal, implementation related: settings that should not need to be changed.

For most settings, only one entry should appear in the configuration file(s) of **torx**(1). The settings for which multiple entry lines are allowed, are prefixed with an asterisk (∗) below. Note that this asterisk is *not* part of the entry keyword!

GENERAL

This section contains the general configuration entries that may be useful in general.

∗**INCLUDE** *filename*

Read the entries from *filename* at the point where this entry is encountered (and then continue processing the file in which the **INCLUDE** entry was encountered). Warning: currently **NO** check is made against recursive inclusion. Be careful!

EXECUTION-RUN

This section contains the settings that you may want to vary from one test-execution-run to another, even without varying the other settings, to re-execute with a different *seed* or *depth*. It is no coincidence that most of these settings can be overruled from the command line of **torx**(1).

MAXSTEPS *number*

(**torx**(1) options: --depth, --no-depth) The number of steps to do. Also indicates that these steps should be done in ‘‘automatic’’ mode, when **torx**(1) is started. Default value: unset (i.e. **torx**(1) will by default be in single-stepping mode).

LOGFILE *filename*

(**torx**(1) options: --log, --no-log) The filename of the logfile that should be written. If a relative filename is given, it is taken to be relative to the directory containg the configuration file. If a file with the resulting name already exists, the filename will be extended with the sequence ˜*number*˜ where *number* is the smalles number that makes the filename unique. Default value: unset

∗**LOGMON** *command*

(**torx**(1) options: --logmon, --no-logmon) **torx**(1) will start torx log monitoring *command* as a background process that can continue to run even after torx itself has exited, and provides the *command* with the text of the log file on its standard input. No **LOGFILE** need to be set for this to work. **torx**(1) captures the standard output and standard error output of *command* (as long as torx is running) and prints this on standard error (preceded by a prefix). This is typically used to start

tools that give a particular ‘‘view’’ of (an aspect of) the test run. Visualization can be such view; see for example **anifsm**(1). The *command* may have the form *command***with***arguments* **#** *xtorx***Tools** in which case the *command with arguments* will be run and the *xtorx Tools menu entry* will be present in the **xtorx**(1) *Tools* menu. Default value: unset

TRACEFILE *filename*

(**torx**(1) options: --trace, --no-trace) The filename of the tracefile that should be read. If a relative filename is given, it is taken to be relative to the directory containg the configuration file. Default value: empty

SEED *number*

(**torx**(1) options: --seed, --no-seed) The seed of the randum number generators used by the components of TorX. Default value: 4

PRIMER

This section contains the settings that you usually only have to specify once when you start a test-campaign.

SPEC *filename*

The filename of the explorer/primer program. The explorer/primer program will be started from the directory given with the **SPECRUNDIR** entry. Note that the default value for **SPECRUNDIR** is *not* the current working directory! Default value: unset

∗**SPECFLAGS** *arguments*

(Additional) arguments that will be given as arguments to the explorer/primer program when it is started. Default value: unset

SPECRUNDIR *directory*

The directory from which the explorer/primer program will be started. Default value: the directory containing the explorer/primer program as specified in the **SPEC** entry.

SPECSOURCE *filename*

The filename of the specification from which the explorer/primer program was built. If specified, **xtorx**(1) will enable the ‘‘Show primer’’ menu entry, and when that menu entry is activated, **xtorx**(1) will invoke **xtorx-showspec**(1) with the *filename* as argument. Default value: unset

SOURCESPEC *filename*

Deprecated. Use **SPECSOURCE** *filename* instead.

PRIMERS *filename*

The filename from which **xtorx(1)** will initialise its *Primers* menu (as if it was opened with its **Open Primers...** menu entry). If **torx**(1) is invoked without command line option **--gui** and if both **PRIMERS** and **PRIMER** are set, it will use **PRIMERS** to initialise its list of primers, and use **PRIMER** to select one from this list, in order to produce the side effects necessary to run the selected primer. Default value: unset.

PRIMER *entryname*

The menu entry that **xtorx(1)** will select in its *Primers* menu, or show in its title-bar if no *Primers* menu is present. If **torx**(1) is invoked without command line option **--gui** and if both **PRIMERS** and use **PRIMER** are set, it will use **PRIMERS** to initialise its list of primers, and **PRIMER** to select one from this list, in order to produce the side effects necessary to run the selected primer. Default value: empty.

GUIDES *filename*

The filename from which **xtorx(1)** will initialise its *Guides* menu (as if it was opened with its **Open Guides...** menu entry). If **torx**(1) is invoked without command line option **--gui** and if both **GUIDES** and **GUIDE** are set, it will use **GUIDES** to initialise its list of guides, and use **GUIDE** to select one from this list, in order to produce the side effects necessary to run the selected guide. Default value: unset.

GUIDE *entryname*

The menu entry that **xtorx(1)** will select in its *Guides* menu, or show in its title-bar if no *Guides* menu is present. If **torx**(1) is invoked without command line option **--gui** and if both **GUIDES** and **GUIDE** are set, it will use **GUIDES** to initialise its list of guides, and use **GUIDE** to select one from this list, in order to produce the side effects necessary to run the selected guide. Default value: empty.

∗**INPUT** *gatename ignored encoding-routine* [*pcoOf-routine*]
This feature specifies that the events on LOTOS gate *gatename* are to be interpreted as input events. See the **ADAPTER** section for an explanation of the remaining arguments.

∗**OUTPUT** *gatename*
This feature specifies that the events on LOTOS gate *gatename* are to be interpreted as output events. In addition, it can have the same additional arguments as the **INPUT** feature, but these are all ignored for the **OUTPUT** feature.

CHOOSEINPUTS *boolean*
Indicate whether or not the DRIVER should select inputs from the menu, if the user does not choose. This is needed if an *iochooser* is used to choose values for ''symbolic'' events in the *Promela* specification. Allowed values: 0 (false), 1 (true). Default value: 0

LABEL-DELTA *string*
The string representation of the action/event that represents the *delta* action, i.e. quiescence, in the communication with the explorer/primer component. This value should be parseable as a LOTOS event. Default value: Delta Note: this value is just a default which is available to primer and adapter when they are started, and both can choose to ignore it. This value is also used when the driver has to send a quiescence event to the primer, and the adapter did not include an event.

ADAPTER
This section contains the settings that you usually only have to specify once when you start a test-campaign.

ADAPTOR *filename*
The filename of the adapter. It will be started as a subprocess of **torx**(1). It will be invoked with the configuration file arguments that are given to **torx**. Note: the **adapter** will be started from the directory given with the **ADAPTORRUNDIR** entry. Note that the default value for **ADAPTORRUNDIR** is *not* the current working directory (except for the adapters supplied with TorX, for which the current working directory *is* the default **ADAPTORRUNDIR**) Default value: adaptor

∗**ADAPTORFLAGS** *arguments*
(Additional) arguments that will be given as arguments to the explorer/primer program when it is started. Default value: unset

ADAPTORRUNDIR *directory*
The directory from which the **adapter** program will be started. Default value: the directory containing the **adapter** program as specified in the **ADAPTOR** entry.

ADAPTORCONTEXT *filename*
The filename of the program that will be used to as filter between TorX and **adapter** when TorX starts the **adapter**. The filter works in two ways: both the standard input written to the IUT/SUT and the standard output read from the IUT/SUT will be filtered by the program. Default value: unset

IUT *filename*
The filename of the SUT/IUT. The presence of this feature indicates that the SUT/IUT has to be started by **torx**(1). It will be started as a subprocess of **torx**(1), and **torx**(1) will have three pipes to it: to its standard input, standard output and standard error. The standard input and standard output pipes constitute the PCO address **pipe** (see the **ADDRESS** entry, below) Note: the SUT/IUT will be started from the directory given with the **IUTRUNDIR** entry. Note that the default value for **IUTRUNDIR** is *not* the current working directory! If this feature is not present,

TorX supposes that the SUT/IUT is already running, or started outside TorX, and TorX only has to be able to connect to it. Default value: unset

∗**IUTFLAGS** *arguments*
(Additional) arguments that will be given as arguments to the SUT/IUT program when it is started. Default value: unset

IUTRUNDIR *directory*
The directory from which the SUT/IUT program will be started. Default value: the directory containing the SUT/IUT program as specified in the **IUT** entry.

IUTSOURCE *filename*
The filename of the specification from which the IUT/SUT program was built. This is particularly useful when we use a ''simulator'' as IUT/SUT. If specified, **xtorx**(1) will enable the ''Show mutant'' menu entry, and when that menu entry is activated, **xtorx**(1) will invoke **xtorx-showspec**(1) with the *filename* as argument. Default value: unset

SOURCEIUT *filename*
Deprecated. Use **IUTSOURCE** *filename* instead.

MUTANTS *filename*
The filename from which **xtorx(1)** will initialise its *Mutants* menu (as if it was opened with its **Open Mutants...** menu entry). If **torx**(1) is invoked without command line option **--gui** and if both **MUTANTS** and **MUTANT** are set, **torx** will use **MUTANTS** to initialise its list of mutants, and use **MUTANT** to select one from this list, in order to produce the side effects necessary to run the selected mutant. Default value: unset.

MUTANT *entryname*
The menu entry that **xtorx(1)** will select in its *Mutants* menu, or show in its title-bar if no *Mutants* menu is present. If **torx**(1) is invoked without command line option **--gui** and if both **MUTANTS** and **MUTANT** are set, it will use **MUTANTS** to initialise its list of mutants, and use **MUTANT** to select one from this list, in order to produce the side effects necessary to run the selected mutant. Default value: empty.

IUTCONTEXT *filename*
The filename of the program that will be used to as filter between TorX and IUT/SUT when TorX starts the IUT/SUT. The filter works in two ways: both the standard input written to the IUT/SUT and the standard output read from the IUT/SUT will be filtered by the program. This feature is only useful (and currently only used) when a tcl-style adapter is used. The filter program will be invoked with as arguments the IUT program together with its arguments. The **hexcontext**(1) program can be used here. Default value: unset

IUTCONTEXTFLAGS *flags*
Additional arguments for the program given as **IUTCONTEXT**. For **hexcontext**(1) a useful flag is **--**. Default value: unset

IUTTIMEOUT *real*
The timeout value for the IUT/SUT, in seconds. Fractions are allowed. Infinity is denoted with ''-1''. Default value: 11 seconds.

USEGCI *boolean*
Indicate whether or not the GCI-style adapter should be used. If the GCI-style adapter is not used, the tcl-style adapter will be used. Allowed values: 0 (false, i.e. use tcl-style adapter), 1 (true, i.e. use GCI). Default value: 0

GCIADAPTER *name*
Indicate the name of the tcl-package that implements the adapter. This package is expected to be found either in the library directory of TorX, or in the directory specified in the **CODING** feature. Default value: gcitcl

CODING *directory*

The directory that contains the conding routines, either in the form of tcl files (when the tcl-style adapter is used), or in the form of a tcl package (when the gci-style adapter is used). Default value: unset

∗**INPUT** *gatename ignored encoding-routine* [*pcoOf-routine*]
This feature specifies that the events on LOTOS gate *gatename* are to be interpreted as input events. In addition, it specifies that the **tcl** routine *encoding-routine* is to be used to encode events on gate *gatename*, and the optional *pcoOf-routine* has to be used to map an event on gate *gatename* to its pco (if all events on a gate are mapped onto the same pco, this routine may be omitted). The *encoding-routine* routine is invoked with two arguments: 1) the gatename, and 2) a list of value-expressions. The result should be a string that can be send to the IUT. The *pcoOf-routine* routine is invoked with two arguments: 1) the gatename, and 2) a list of value-expressions. The result should be a pconame that appears in the configuration file. If feature **USEGCI** is set, then the *encoding-routine* and *pcoOf-routine* are ignored, and the GCI-specific configuration is used. The deprecated **ignored** field was meant to specify the sort-list of the gate − but it is simply ignored.

∗**OUTPUT** *gatename*
This feature specifies that the events on LOTOS gate *gatename* are to be interpreted as output events. In addition, it can have the same additional arguments as the **INPUT** feature, but these are all ignored for the **OUTPUT** feature.

∗**ADDRESS** *addressname protocol* [*host*] [*port*] [*program*] [*flags*]
Specification of an address to connect to the SUT/IUT, with optional *host*, *port*, *program* to be used to connect, and *flags* to be given to the *program*. For the *host* field, the special name **currenthost** can be used as a ‘‘wildcard’’ referring to the current host: that value is substituted by the result of **torx-hostname**(1). If the connector *program* is omitted, it defaults to a program with the same name as the *protocol*. The *addressname* can be used in the encoding- and decoding routines to be referred to as abstract address name (i.e. to allow change of the actual hostname/port-number without having to change the coding routines).

Currently implemented values for *protocol*:

udp in which case *host* and *port* should be given; the connector *program* may be omitted (i.e. defaults to a program with the same name as the *protocol*, i.e. **udp** and the connector *flags* should be **{--port ${P}}** (where **${P}** is a variable that represents the port number) for the **udp**(1) program supplied with TorX. (See the first example in the EXAMPLES section below.)

telnet in which case *host* and *port* should be given, the connector *program* and *flags* may be omitted, and the **telnet**(1) program will be used to connect to the IUT.

pipe in which case *host*, *port*, connector *program* and *flags* may be omitted, and the pipes made to IUT when it is started from XtorX are used.

manual (currently only supported for stimulation, not for observation; currently only supported via the GCI-based **adapter**) in which case, for all stimili that have to be given on the pco with this address, TorX will ask the human operator to perform the actual stimulation.

∗**PCO** *pconame gatenames addressnames decoding-tuple-list*
where *gatenames* is either a single gatename (for an uni-directional pco), or an output gatename followed by an input gatename, enclosed between ‘‘{’’ and ‘‘}’’ (for a bi-directional pco), and *decoding-tuple-list* is a list of white-space separated tuples. Each tuple is enclosed between ‘‘{’’ and ‘‘}’’, and contains the fields **ignored**, *decoding-routine*, and optional *regexp*. The **ignored** field was meant to specify the abstract and concrete types of the data sent over the pco, but this information has never been put to use. This feature specifies that pco *pconame* functions as uni-directional or bi-directional pco for events on the given *gatename* or *gatenames*. In addition, this feature assigns address *addressname* to this pco (*addressname* should be specified in an

ADDRESS entry elsewhere in the configuration file). Finally, this feature specifies how to do observations on this pco: all *regexp* regular expressions of the *decoding-tuple-list* are concatenated. When an observation is made, for each pco the combined regular expression is tried on the output received on the pco. For the first pco for which the combined regular expression matches the output received on a pco, the first specified *decoding-routine* of the pco is invoked with two arguments: 1) the pconame, and 2) a list of strings. The first string in this list is the match of the regular expression on the observed output; the optional remaining strings correspond to sub-expressions of the regular expression. NOTE: it is currently best to specify for each pco exactely one *decoding-routine* and one *regexp*.

INSTANTIATOR

This section contains the settings that you usually only have to specify once when you start a test-campaign.

INST *filename*

The filename of an instantiator. The presence of this feature indicates that the instantiator has to be started by **torx**(1). It will be started as a subprocess of **torx**(1), and **torx**(1) will have three pipes to it: to its standard input, standard output and standard error. Over the standard input and output, **torx**(1) will issues the commands and expect the responses described in **torx-instantiator**(5). Note: the instantiator will be started from the directory given with the **INSTRUNDIR** entry. Note that the default value for **INSTRUNDIR** is *not* the current working directory! Note that the **adaptsim**(1) adaptor also honors the **INST** config entry, and that when TorX starts an adaptor, it gives it the same configuration files. If this feature is not present, TorX supposes that no instantiator is needed. Default value: unset

∗**INSTFLAGS** *arguments*

(Additional) arguments that will be given as arguments to the instantiator program when it is started. Default value: unset

INSTRUNDIR *directory*

The directory from which the instantiator program will be started. Default value: the directory containing the instantiator program as specified in the **INST** entry.

INTERNAL

It is not advised to change the settings in this section, unless you know very well what you are doing, because they influence the ‘‘heart’’ of the system − changing a setting here might break the system.

SPECTIMEOUT *real*

The timeout value for the communication with the explorer/primer, in seconds. Fractions are allowed. Infinity is denoted with ‘‘-1’’. Default value: -1 (i.e. infinity).

ADAPTORTIMEOUT *real*

The timeout value for the communication with the adaptor, in seconds. Fractions are allowed. Infinity is denoted with ‘‘-1’’. Default value: -1 (i.e. infinity).

PROMPT *string*

The prompt used by the non-GUI interface to TorX, i.e. by **torx**(1). Note that currently **xtorx**(1) uses the prompt string to keep itself synchronised with **torx**(1). Default value: "tester> ".

DEBUG *number*

The level of debugging information that should be printed. Default value: 1

EXAMPLES

The example configuration below is what is used for the LOTOS primer for the Conference Protocol case study. Note that we have expanded here some of the **ignored** arguments of the **INPUT**, **OUTPUT**, and **PCO** entries.

```
#===========================================
IUTTIMEOUT 2
# MAXSTEPS 7
```

```
CODING ./CODING/LOTOS

IUTCONTEXT hexcontext --
IUT ./IUT/conf.jan.longrun.sh

SPEC ./LOTOS/primer.sh
SPECSOURCE ./LOTOS/cf-pe-sut.caesar.lot

# for input, the conversion function of the INPUT def IS used
# for input, the pcoOf function of the INPUT def for CFSAP is NOT used
# for input, the pcoOf function of the INPUT def for udp IS used
# for output, the conversion function of the OUTPUT def is NOT used
# for output, the pcoOf function of the OUTPUT def is NOT used
INPUT  CFSAP_in   { CFAddr CFsp } enCodingOfCFsp
OUTPUT CFSAP_out  { CFAddr CFsp }
INPUT  udp_in     { udpAddr udpsp } enCodingOfUdp pcoOfUdp
OUTPUT udp_out    { udpAddr udpsp } deCodingOfUdp pcoOfUdp

ADDRESS cf1   pipe
ADDRESS udp1  udp currenthost 1075 {--port ${P}}
ADDRESS udp2  udp currenthost 1076 {--port ${P}}
ADDRESS udp3  udp currenthost 1077 {--port ${P}}

# for input, the conversion function of the PCO def is NOT used
# for output, the conversion function of the PCO def IS used
PCO cf1 { CFSAP_out CFSAP_in } cf1 { SFsp_nl CFsp_nl2CFsp {RECVHEX[^\n]+\n} }
PCO udp2 { udp_out udp_in }   udp2 { udp_nl udp_nl2udpsp {RECVHEX[^\n]+\n} }
PCO udp3 { udp_out udp_in }   udp3 { udp_nl udp_nl2udpsp {RECVHEX[^\n]+\n} }

### we may want to use several SEEDs, to reproduce errors.
# do not specify seed here; use from command line
# SEED  4

# do not specify logfile here; use from command line
# LOGFILE logs/conference.lotos.log

# next line used a log file as initial trace
# do not specify tracefile here; use from command line
# TRACEFILE conf.trace
#============================================
```

The example configuration below shows the very minimal that is needed to use a simulator as IUT. We assume here that the specification is in file sim.lot. We also assume that the specification that is to be used as simulator-IUT is in impl.lot. Finally, we assume that when **mkprimer**(1) was invoked to process sim.lot and impl.lot, for both files the same input and output gates have been specified with the **--inputs** and **--outputs** flags of **mkprimer**(1).

```
#============================================
# instead of using INPUT and OUTPUT to specify the input and output gates,
# we just include a file generated by mkprimer.
# we could just as well include impl.gates: spec.gates and impl.gates
# are identical (w.r.t. INPUT and OUTPUT entries that they contain),
# because we invoked mkprimer with the same --inputs and --outputs flags
# when we invoked in to process spec.lot and impl.lot .
INCLUDE sim.gates
```

```
# use the adapter supplied with torx for simulator-as-iut usage
ADAPTOR adaptsim

# specify spec and iut, and the source of it
SPEC sim
SPECSOURCE sim.lot
IUT sim
IUTSOURCE sim.lot

# This PCO entry is not really used, but without it torx will complain.
PCO ignored
#============================================
```

BUGS

The interpretation of relative paths in file and directory names, relative to the configuration file, only works when **torx**(1) is started in the directory containing the configuration file (when **torx**(1) is started via **xtorx**(1) this is the case − **xtorx**(1) takes care of this). For configuration files included via the **INCLUDE** entry, it only works if the included file is in the same directory as the file containing the **INCLUDE** line.

The **INPUT** and **OUTPUT** entries combine information for the Primer with information for the Adapter; it would be better to separate those (or at least to allow the user to specify this information in separate entries).

The configuration of the **PCO** entry should be simplified.

The tcl-list interpretation of the entry lines, resulting in all those ‘‘{’’ and ‘‘}’’, is not very nice.

When a GCI-style Adapter is used, very little from the configuration file is used − instead, configuration information is hard-coded into the C code of the GCI-style Adapter. This should be changed, such that a GCI-style Adapter also uses the information from a (the) configuration file, preferably in such a way that the same configuration file can be used both for a tcl-style and for a GCI-style Adapter.

The use of adaptor/adapter should be normalised!

SEE ALSO

torx-intro(1), **torx**(1), **torx-primer**(5), **torx-adaptor**(5), **torx-instantiator**(5), **xtorx**(1), **xtorx-showspec**(1), **torx-hostname**(1), **telnet**(1), **udp**(1), **hexcontext**(1), **anifsm**(1)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of **torx**.

NAME

torx-log − log file generated by torx tester

DESCRIPTION

A **torx**(1) log file should contain all information that is needed for analysis of a test execution run. The log file consists of a number of lines of text. Each line starts with a keyword that identifies the type of line. The keyword is followed by a test-step nummber (except for the keywords **CONFIG** and **CLICKSPERMS**). The interpretation of some lines may depend on the particular instantiation of a particular tool-component, like the **STATE** and **STATS** lines.

DETAILS

Below follows the list of lines, by keyword.

ABSTRACT *test-step-nr iokind-interface action suspension*

where *iokind-interface* is *iokind***[(***pco***)]** i.e. the iokind **input** or **output** followed by optional interface information, enclosed in parentheses (and). The interface information is **[***channel***:]***pco* i.e. the PCO name, optionally preceded by the channel name. Where *action* is, enclosed in parentheses (and), the action derived from the specification, ''executed'' by the adapter, and *suspension* is, enclosed in parentheses (and), **1** in case of input suspension or quiescent observation, and **0** otherwise

CLICKSPERMS *nr*

the number of Tcl clicks per milli-second (used as primitive high-resolution counter)

CONCRETE *test-step-nr iokind-interface ??? ???*

the concrete information send over the interface, where both ??? are enclosed in parentheses (and)

CONFIGSTART

start of **CONFIG** section

CONFIG *keyword value...*

information copied from the **torx-config**(4) configuration files

CONFIGEND

end of **CONFIG** section

EOF *test-step-nr*

end-of-lof-file indication

EXPECTED *test-step-nr iokind-interface action suspension*

abstract action that was expected to be observed, in the same format as the **ABSTRACT** lines

LOG *test-step-nr details...*

TorX tools components **torx-primer**(5) and **torx-adaptor**(5) may add arbitrary **LOG** lines with logging information.

MODE *test-step-nr mode*

where *mode* is either **normal**, for test step done in manual mode, or **auto**, for test step done in automatic mode

STATS *test-step-nr details...*

statistics from the primer component, where the *details...* depend on the particular primer used. See the section **LOGFILE STATS** in **intersector**(1), **mkprimer-cadp**(1), and **mkprimer-trojka**(1)

STATEID *test-step-nr details...*

(super)state-identifier number(s) from the primer component, where the *details* depend on the particular **torx-primer**(5) See the section **LOGFILE STATEID** in **intersector**(1), **mkprimer-aut**(1), **mkprimer-cadp**(1), **mkprimer-ltsa**(1), **mkprimer-mcrl2**(1), and **mkprimer-trojka**(1)

where *epoch-seconds* is the unix time format (seconds since start of unix epoch, 1 jan 1970), *clicks*

is a high-resolution counter that should count in (approximately) milli-seconds, and *date-string* is a human-readable date string, as output by **date**(1)

VERDICT *test-step-nr correctness* **[***observation-objective***]**
where *correctness* is one of (the traditional) **pass**, **fail**, or **inconclusive**, and the optional *observation-objective* indicates the success of reaching the observation-objective (or the test-purpose), so it is one of **hit** or **miss**

SEE ALSO

torx-intro(1), **torx**(1), **intersector**(1), **mkprimer-aut**(1), **mkprimer-cadp**(1), **mkprimer-ltsa**(1), **mkprimer-mcrl2**(1), **mkprimer-trojka**(1) **adaptlog**(1),

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of **torx**.

NAME

mkprimer − support specification language or toolkit

BUGS

(Manual page needs still to be written)

SEE ALSO

torx-intro(1),

NAME

torx-adaptor − a program that implements an interface to the SUT

SYNOPSIS

ADAPTOR

DESCRIPTION

In this man page we discribe the interface of the ADAPTOR module. The interface is very similar to the one of the PRIMER (See **torx-primer**(5)).

When active, the ADAPTER receives commands from the standard input and writes answers in return to the standard output. See section **COMMANDS** for the available commands and answers. When the ADAPTER is used in the TORX tool it is connected to the DRIVER which has the control and initiative of all communications. When the ADAPTER is used standalone the user plays the role of the DRIVER.

A command consist of a single line of text. The keyword (the first word) of each commands start with a prefix **C_**.

An answer consist of a single line of text or a multi-line of text. All single-line answers start with a keyword that has prefix **A_**. Usually, the keyword of single-line answer is identical to the command-keyword, but with the **C_** replaced by **A_**. A multi-line answer has specific markers around the body of the answer. These markers consist of a single line that starts with a prefix **A_**, followed by the command-keyword without the initial **C_,** followed by **_START** respectively **_END**. Each line of the body of a multi line answer often starts with a keyword (but not always).

Instead of the usual single- or multi-line answer, an **A_ERROR** answer may occur in case of an error. Such an **A_ERROR** answer comes instead of the usual single- or multi-line answer.

Additional message lines containing warnings (to be shown to the test run operator), diagnostics (to appear in the test run log), and/or debugging output (to be shown to the test run operator) may appear interspersed with the answer lines described so far. These message lines start with **A_WARNING**, **A_LOG** or **A_DEBUG**, respectively. Because these message lines may appear arbitrarily intermixed with the ‘‘regular’’ answer line(s), they do not play a role when the ‘‘end’’ of an answer has to be found: once the command-specific answer or **A_ERROR** answer has been read, the user of the ADAPTER may assume that no other output (specific to to the command) will be generated by the ADAPTER until the next command is send to it.

If the ADAPTOR decides autonomously that it will stop executing, it may try to indicate this by sending an **A_QUIT** answer, just before exiting. For the ADAPTOR, reading end-of-file on its standard input is a reason to stop executing.

In all cases, each command and each answer starts with a keyword that is separated from the remainder of the line by one or more spaces or tabs − except for two notable exceptions: **A_ERROR**, where currently exactely one space should be between the keyword and the category, and the lines between **A_STATE_START** and **A_STATE_END**, that do not start with a keyword (see **DETAILS** below). For most comands and most answers, the text following the keyword has a fixed format; for other commands and answers it is left unspecificed.

For command and answers that deal with channels, events, predicates, etc. the format of each line is as follows: the keyword is followed by a (possibly empty) TAB-separated list of *name=value* items, where the *name* cannot contain TABs or = characters, and the *value* cannot contain TAB characters. For the moment we assume that, when necessary, we will encode TAB characters for example using C conventions. We have chosen for this kind of item-lists because this makes it easy to pass information. Using *name=value* pairs instead of (for example) a list of *values* in a fixed order makes the interface more robust: the order of items is not important, and the users can simply ignore items which *name* they don’t recognize. This allows one party to send items which the receiving party doesn’t know about. This makes it easier to extend the interface by adding new items and keeping old programs backwards compatible.

For **A_ERROR** answers, currently the format is as follows: the A_ERROR keyword is followed by a category word indicating the category of the error, followed by arbitrary text that describes the error in more detail. The idea is that the user of the ADAPTOR can use the category word to get some idea of the nature of the error. In particular, the idea is that the user can use this to decide whether it makes sense to retry, or whether it is better to give up.

DETAILS

Here we describe first the syntax of the (textual) messages interchanged over the interface, followed by the possible message arguments, followed by the possible error categories.

SYNTAX

Below we describes the syntax of the commands and answers, by giving the syntax of just a single interaction.

interaction:
message∗ command message∗ answer-or-error

command: C_*name arguments-upto-end-of-line* nl

answer-or-error:
single-line-answer | *multi-line-answer* | *error*

single-line-answer:
A_*name arguments-upto-end-of-line* nl

multi-line-answer:
A_*name*_START nl *multi-line-element∗* A_*name*_END nl

multi-line-element:
message | *result-element*

result-element:
A_*element-name arguments-upto-end-of-line* nl

message: *debug* | *log* | *warning*

debug: A_DEBUG *text-upto-end-of-line* nl

log: A_LOG *text-upto-end-of-line* nl

warning: A_WARNING *text-upto-end-of-line* nl

error: A_ERROR space *category* : *text-upto-end-of-line* nl

arguments-upto-end-of-line:
ws∗ | *ws+ arguments*

arguments: *argument* | *argument* tab *arguments*

argument: *name = value*
/∗ value can contain spaces but no tabs ∗/

text-upto-end-of-line:
ws∗ | *ws+ arbitrary-text-without-newline*

ws: space | tab

ARGUMENTS

At the moment the following *names* are defined for the *name=value* arguments to the commands:

event with as value the LOTOS-like string representation of an event, used wherever ‘‘event’’ is used below;

channel with as value a channel name;

iokind where *iokind* is a value **input** or **output**;

suspension with as value either ‘‘0’’ or ‘‘1’’ to indicate a non-suspension resp. suspension event, used

wherever an output ‘‘event’’ is used below (in the future we expect to use it also for input events)

pco with as value a pco name;

concrete with as value a (string representation of the) concrete value (this string cannot contain tab or new-line characters);

timestamp with as value a (string representation of the) time

ERRORS

At the moment, for the A_ERROR *category* messages given by the ADAPTER, the following *categories* are defined. The user of the ADAPTER (i.e. torx) may use these categories to decide how to react to the **A_ERROR** answer (in particular, to decide whether it can cope with it, or whether it is better to give up).

UnknownCommand the command is not known

ArgumentMissing a mandatory argument of the command is missing or incomplete

WrongValue the value of an argument of the command is outside its domain of valid values

ParseErrorEvent the command contains an **event** argument which cannot be parsed

UnknownIOKind the command contains an **event** argument, and the **iokind** of the event cannot be computed

Inconsistency the command contains more than a single argument, and the values of (some of) the arguments conflict which each other

InternalError an internal error occurred during the execution of the command

COMMANDS

Below follows the list of the commands. The commands, and their answers, are described in more detail below. With each command, and answer, the possible arguments are indicated. Optional arguments are surrounded by brackets ’[’ and ’]’. Note: the order in which the arguments are named with the commands does *not* prescribe the *order* in which arguments may appear: arguments may appear in any order.

C_CHANNELS [iokind] [channel]
C_PCOS [iokind] [channel] [pco]
C_IOKIND [iokind] [channel]
C_INPUT channel event
C_OUTPUT channel
C_STATE
C_STATES
C_QUIT
C_GETCONFIG

The C_INPUT and C_OUTPUT commands are the most important ones, and have to be implemented correctly for TorX to work correctly; the other commands are less important, and most of them are mainly used to get configuration information from the ADAPTOR, only for logging purposes.

In the subsections below, each command is followed by a short explanation of its use. Additional comments are enclosed between /∗ and ∗/ after the command.

CHANNELS

The C_CHANNELS command is used to ask the ADAPTOR which channels exist. If the **iokind** and/or **channel** argument is used, then only the respective channels are returned. This command has no side-effects, and can repeatedly applied without changing the state of the ADAPTOR module.

Command:

C_CHANNELS [iokind] [channel]

Answer (multi-line):

A_CHANNELS_START

A_CHANNEL iokind channel
...
A_CHANNELS_END

Errors (single-line):

A_ERROR Inconsistency */*if the iokind and channel fields conflict*/*
A_ERROR WrongValue */*if a given field has value outside domain*/*

PCOS

The C_PCOS command is used to ask the ADAPTOR which pcos exist. If a pco is used both for input and output, then there will be two A_PCO lines for it, one with **iokind=input** and the other with **iokind=output**. If the **iokind**, **channel** and/or **pco** argument is used, then only the respective pcos are returned. This command has no side-effects, and can repeatedly be given without changing the state of the ADAPTOR module.

Command:

C_PCOS [iokind] [channel] [pco]

Answer (multi-line):

A_PCOS_START
A_PCO iokind channel pco
...
A_PCOS_END

Errors (single-line):

A_ERROR Inconsistency */*if the iokind, channel and pco fields conflict*/*
A_ERROR WrongValue */*if a given field has value outside domain*/*

IOKIND

The C_IOKIND command is used to ask the ADAPTOR whether the ADAPTOR has a preference for the next action to be an input or an output action, or wether the ADAPTOR doesn't care, and any action kind is possible. The idea of this command is to allow the ADAPTOR to indicate that it has observations qeueued. The command can have an optional suggestion, which will be honoured by the ADAPTOR if possible. This command has no side-effects, and can repeatedly be given without changing the state of the ADAPTOR module. If the ADAPTOR doesn't care about the action kind when no suggestion was given, it simplu should not return iokind nor channel paratmers in the A_IOKIND answer.

Command:

C_IOKIND [iokind] [channel]

Answers (single-line):

A_IOKIND iokind channel
A_IOKIND */*if adaptor doesn't care and no sugg. given*/*

Errors (single-line):

A_ERROR Inconsistency */*if the iokind and channel fields conflict*/*
A_ERROR WrongValue */*if a given field has value outside domain*/*

INPUT

The C_INPUT command is used to ask the ADAPTOR to stimulate with a given action. If the action given cannot be parsed, or if other errors occur when trying to ''do'' the action, the ADAPTOR will return A_ERROR (with an explanation of the error), and no action will be done; if the action can be parsed, but is not ''enabled'' (i.e. is not in the ''menu of possible input actions of the IUT''), and therefore cannot be ''done'', the ADAPTOR will return A_INPUT_ERROR, and no action will be done; otherwise the ADAPTOR will return A_INPUT with the action that has been ''done'', if possible including the pco on which the action was done, a timestamp, and the concrete representation of the action.
Questions: are we now mapping input-suspension onto A_INPUT_ERROR? Should we get rid of A_INPUT_ERROR here, and use either A_ERROR (for IOCO) or A_INPUT_OK with **suspension=1** (for MIOCO)? Note: currently, it is best to avoid using A_INPUT_ERROR; use A_ERROR instead.

Command:

C_INPUT channel event

Answers (single-line):

A_INPUT_OK channel event [pco] [timestamp] [concrete-event]
A_INPUT_ERROR */*if event cannot be done*/"*

Errors (single-line):

A_ERROR ParseErrorEvent */*if the event cannot be parsed*/*
A_ERROR ArgumentMissing */*if no event was given to simulator*/*
A_ERROR UnknownIOKind */*if iokind of event cannot be found*/*
A_ERROR Inconsistency */*if the iokind and channel fields conflict*/*
A_ERROR WrongValue */*if a given field has value outside domain*/*
A_ERROR InternalError */*if internal error occurred*/*

OUTPUT

The C_OUTPUT command is used to ask the ADAPTOR for an observation on a given channel. If observing fails, the ADAPTOR will return A_ERROR (with an explanation of the error). Otherwise, the ADAPTOR will return A_OUTPUT_OK with the observation, if possible including the pco on which the observation was done, a timestamp, and the concrete representation of the observation.

Command:

C_OUTPUT channel

Answer (single-line):

A_OUTPUT_OK channel event [pco] [timestamp] [concrete-event]

Errors (single-line):

A_ERROR ArgumentMissing */*if no event was given to* **IOCO** *or traces*/*
A_ERROR UnknownIOKind */*if iokind of event cannot be found*/*
A_ERROR WrongValue */*if a given field has value outside domain*/*
A_ERROR InternalError */*if internal error occurred*/*

STATE

The C_STATE command returns a textual representation of the current state. The contents of this textual representation depend on the implementation of the ADAPTOR. This command has no side-effects.

Command:

C_STATE

Answer (multi-line):

A_STATE_START
text **...**
...
A_STATE_END

STATS

The C_STATS command returns some statistics about the ADAPTOR. The statistics consists of a list of whitespace-separated key-value pairs, where also the key and the value are separated by whitespace. The value should not contain whitespace. An ADAPTOR should give just one line of statistics (both single- and multi-line form are allowed).

Command:

C_STATS

Answer (single-line):

A_STATS statistics

Answer (multi-line):

A_STATS_START
A_STATS statistics

...
A_STATS_END

QUIT

The C_QUIT command tells the ADAPTOR to clean up and exit. The ADAPTOR will acknowledge the command with A_QUIT. The side-effect of this command is that the ADAPTOR module exits.

Command:
C_QUIT

Answer (single-line):
A_QUIT

GETCONFIG

This command has not been implemented yet. It is to be used to get configuration information from the ADAPTOR.

IMPLEMENTATION NOTES

For the implementer of an adapter, the most important commands to implement are C_INPUT and C_OUTPUT. Commands C_INPUT and C_OUTPUT are needed so the driver can stimulate and observe.

The C_IOKIND command is important, because if it is implemented in the wrong way, the user can no longer control the tester with the commands in **torx**(1) or the buttons in **xtorx**(1). The best default implementations for C_IOKIND are those that either returns A_IOKIND with the same parameters that were given with the C_IOKIND command, or just returns A_IOKIND without parameters.

If the C_PCOS command is implemented such that it indeed returns a list of the pcos, and the pco parameter is used for the A_INPUT_OK and A_OUTPUT_OK answers, then TorX is able to draw better Message Sequence Charts.

For all the other commands, the implementer can choose to give simple answers, by returning just the answer keyword without parameters, for single-line answers, and returning just the answer start and answer end keywords for multiline answers. Of course, more informational answers proved more feedback during testing, and may make it more easy to investigate in case of errors -- the information above is mainly meant to make it easier to quickly implement a ‘‘basic’’ adapter that already ‘‘does the right thing’’.

SEE ALSO

torx-intro(1), **mkprimer**(1), **torx-primer**(5), **torx-config**(4), **torx-log**(4), **torx**(1), **sh**(1)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of **torx**.

NAME

torx-explorer − interface to program to explore a labelled transition system

SYNOPSIS

EXPLORER [*options* **]**

DESCRIPTION

The EXPLORER module implements an (textual) interface to explore a labelled transition system (lts).

When active, the EXPLORER receives commands from the standard input and writes answers in return to the standard output. See section **COMMANDS and ANSWERS** for the available commands and answers. When a EXPLORER is used in the TORX tool it is connected to a PRIMER which has the control and initiative of all communications. When the EXPLORER is used as standalone the user plays the role of the PRIMER.

A command consist of a single line of text. The first character of a command indicates the type of the command. This first character is always in lowercase.

An answer consist of a single line of text or a multi-line of text. All single line answers start with the character of their command turned into uppercase. A multi-line answer has specific markers around the body of the answer. These markers consist of two characters at a single line, made up from the usual answer prefix character, with **B** and **E** appended, respectively. Each line of the body of a multi line answer starts with a two character prefix consisting of the usual uppercase answer prefix character, followed by the same character in lowercase. Each answer line can contain multiple fields. Arbitrary whitespace (one or more spaces or tabs) separates the fields from the (one or two character) prefix at the start of the line. The fields themselves are separated by a singe tab character. A fields cannot contain tab characters, is assumed not to start or end with spaces.

Instead of the usual answer, an error answer may occur in case of an error. Also error answers start with a two character prefix, consisting of the usual uppercase answer prefix character, followed by the character "0".

The following describes the syntax of the commands and answers:

interaction:
message∗ *command message*∗ *answer-or-error*

command:
lowercase(*name*) *arguments-upto-end-of-line* nl

answer-or-error:
single-line-answer | *multi-line-answer* | *error*

single-line-answer:
uppercase(*name*) *arguments-upto-end-of-line* nl

multi-line-answer:
uppercase(*name*)B nl *multi-line-element*∗ uppercase(*name*)E nl

multi-line-element:
message | *result-element*

result-element:
uppercase(*element-name*) *arguments-upto-end-of-line* nl

message:
debug | *log* | *warning*

debug: A_DEBUG *text-upto-end-of-line* nl

log: A_LOG *text-upto-end-of-line* nl

warning:
A_WARNING *text-upto-end-of-line* nl

error*:* uppercase(*name*)0 *arguments-upto-end-of-line* nl

REPRESENTATION

The interface uses numbers to identify transitions (a.k.a. events) and states. A single number identifies at the same time both a transition and the state to which the transition ''points''. If two transitions point to the same state, they will have different numbers. However, we can still tell that they point to the same state, because that information is present in the additional information that is given in the output of each interface command that ''generates'' new transitions. This additional information contains a field ''identical'' that either is empty (if this transition is the first one to ''point'' to its state), or, otherwise, contains the number of the ''first'' transition that pointed to the state. Subsequently generated transitions that point to the same state all have the same value for this field ''identical'', so, in a way, a user can treat the value of this field as the ''canonical'' identification of the state that a transition points to.

Boolean values are represented by the characters ''1'' and ''0'' for respectively **true** and **false**.

For the variable names, types and predicates that are used in a symbolic explorer we expect the following representation. The *type* as used in the field *freevars* ''looks like'' an identifier, i.e. it consists of upper and lowercase characters, digits and underscores. The *normalised-varname* as used in the fields *label*, *preds*, and *freevars* is constructed from the type of the ''original'' variable and a sequence number. By convention it has the form **var_***type***$***nr* as show in the example at the end. The *original-varname* as used in the field *freevars* and in the *predicates* of the instantiate command ''looks like'' an identifier, i.e. it consists of upper and lowercase characters, digits and underscores. The *predicates* used in the instantiate command is a semi-colon separated list of predicates, where each of the individual predicates in the list should not contain newlines or semi-colons. One could imagine various syntaxes for the individual predicates in the list, depending on the purpose of the predicate. Possible purposes are to specify a specific value for a free variable, or to constrain the range of possible values for a free variable. The current symbolic **primer**(1) uses the instantiate command to specify specific values for free variables. It uses the syntax *original-varname* = *expression* for the predicates in the semi-colon separated list of predicates, as illustrated in the example at the end.

COMMMANDS and ANSWERS

The following commands give the core functionality of a non-symbolic explorer, to reset, to expand a state, to delete states, and to quit:

r
e *event*
d *event* ...
q

A symbolic explorer additionally offers the following commands to instantiate an event, and to ask for solutions to predicates:

i *event predicates*
p *predicates*

The command to ask whether a given event matches one of a list of events is implemented in the smile primer, but currently not used in the symbolic primer.

m *event event* ...

Below we describe these commands in more detail.

RESET

This commands tells the explorer to reset itself, and it returns the (transition pointing to) the initial state. It takes no parameters. The result is a single-line answer containing the fields describing the initial state:

event, *solved*, *preds*, *freevars*, *identical*

with

event the event number;

solved a boolean value that indicates whether or not the predicates for the initial state could be solved (i.e. whether or not it is known if a solution exists for the predicates of the initial state);

preds the normalised (semi-colon separated) predicates of the initial state (normalised by replacing

free variables names by names build from the type of the variable and a sequence number, to make it easier to compare them);

freevars the (space-separated) list of free variables information, which contains for each free variable a (space-separated) list of three items: *normalised-varname*, *original-varname*, and *type*;

identical the state to which the state reached by *event* is identical, or the empty string if this event is the first one that reaches that state.

Command:

r

Answer:

R *event* TAB *solved* TAB *preds* TAB *freevars* TAB *identical*

EXPAND

This commands tells the explorer to expand a given state. It takes as parameter a transition/state number. The result is a multi-line answer where each line of the body of the answer contains fields describing a transition and its corresponding resulting state:

event, *visible*, *solved*, *label*, *preds*, *freevars*, *identical*

with

event the event number;

visible a boolean value indicating whether or not the action is visible;

solved a boolean value that indicates whether or not the predicates to reach the resulting state could be solved (i.e. whether or not it is known if a solution exists for the predicates of the resulting state);

label a string containting a LOTOS-like event, containing normalised variables, if the event introduces free variables;

preds the normalised (semi-colon separated) predicates of the resulting state (normalised by replacing free variables names by names build from the type of the variable and a sequence number, to make it easier to compare them);

freevars the (space-separated) list of free variables information, which contains for each free variable a (space-separated) list of three items: *normalised-varname*, *original-varname*, and *type*;

identical

the state to which the state reached by *event* is identical, or the empty string if this event is the first one that reaches that state.

Command:

e *event*

Answer (multi-line):

EB

Ee *event* TAB *visible* TAB *solved* TAB *label* TAB *preds* TAB *freevars* TAB *identical*

...

EE

DELETE

This commands tells the explorer that its user is no longer interested in a (list of) events/states, and will never refer to them in the future. This allows the explorer to delete them, and free memory, if it wants to. Parameter: a (whitespace separated) list of events/states; result: a single-line answer, containing no result (currently).

Command:

d *event* ...

Answer:

D

QUIT

This commands tells the explorer to quit. It takes no parameters, and returns no result.

Command:

q

Answer:

Q

INSTANTIATE

This commands tells the explorer to instatiate a given event using a given list of predicates. It takes as parameter an event, followed by a (semi-colon separated) list of predicates. The resultis a single-line answer, containing the result of instantiation of the given event using the predicates. If the instantiation is successful, the result line contains the same fields as a result line of the expand command; if instantiation is not successful, the error result contains the same fields as the solve command.

The fields of the successful result contain a description of an action and corresponding resulting state:

event, *visible*, *solved*, *label*, *preds*, *freevars*, *identical*

with

event the event number;

visible a boolean value indicating whether or not the action is visible;

solved a boolean value that indicates whether or not the predicates to reach the resulting state could be solved (i.e. whether or not it is known if a solution exists for the predicates of the resulting state);

label a string containting a LOTOS-like event, containing normalised variables, if the event introduces free variables;

preds the normalised (semi-colon separated) predicates of the resulting state (normalised by replacing free variables names by names build from the type of the variable and a sequence number, to make it easier to compare them);

freevars the (space-separated) list of free variables information, which contains for each free variable a (space-separated) list of three items: *normalised-varname*, *original-varname*, and *type*;

identical

the state to which the state reached by *event* is identical, or the empty string if this event is the first one that reaches that state.

The fields of the error result contain information about the (lack of) success of finding a solution, and, if a solution was found, the solution itself:

solved, *found*, *preds*, *msg*, *err*

with

solved a boolean value that indicates whether or not the predicates could be solved;

found a boolean value that indicates whether or not a solution could be found (only of interest when *solved* is 1);

preds the (semi-colon separated) list of predicates that describe the solution found (only of interest when *solved* and *found* are both 1);

msg a (semi-colon separated) list of messages, produced during the narrowing process;

err a (semi-colon separated) list of error messages, produced during the narrowing process.

Command:

i *event* WS *predicates*

Answer (succesfull):

I *event* TAB *visible* TAB *solved* TAB *label* TAB *preds* TAB *freevars* TAB *identical*

Answer (error):

I0 *solved* TAB *found* TAB *preds* TAB *msg* TAB *error*

SOLVE

This commands tells the explorer to solve the given predicates. This does not influence the state of the explorer: the explorer is just used as ''ADT desk calculator''.
Note: this command is *not* used by the (symbolic) primer and is currently *only* there for the convenience of the user. It may be removed in later versions of this interface.
It takes as parameter a (semi-colon separated) list of predicates. The resultis a single-line answer, containing information about the success of finding a solution, and, if a solution was found, the solution itself:

solved, *found*, *preds*, *msg*, *err*

with

solved a boolean value that indicates whether or not the predicates could be solved; (i.e. whether or not it could find a solution or prove that no solution exists);

found a boolean value that indicates whether or not a solution could be found (only of interest when *solved* is 1);

preds the (semi-colon separated) list of predicates that describe the solution found (only of interest when *solved* and *found* are both 1);

msg a (semi-colon separated) list of messages, produced during the narrowing process;

err a (semi-colon separated) list of error messages, produced during the narrowing process.

Command:

p *predicates*

Answer:

P *solved* TAB *found* TAB *preds* TAB *msg* TAB *error*

MATCH

This commands tells the explorer to try to match an event (generally representing a set of events) with a list of events (numbers) (also, each of them generally representing a set of events).
This could be used by the primer to combine the menu's of several states. However, currently it is not used, and it could be removed in later versions of this interface.
It takes as parameter an event, followed by whitespace, followed by a (whitespace separated) list of events. The result is a single-line answer, containing

found, *event*

with

found a boolean value indication whether a matching event was found;

event a matching event from the list, if a match was found, or the empty string, otherwise (none of the events of the list match).

Command:

m *event event ...*

Answer (sccessfull):

M *found* TAB *event*

Answer (error):

M0 *error text upto end of line*

OPTIONS

There are no general command line options for the explorer, that apply to each explorer, because its behaviour can not be parameterised, and therefore does not need command line options. As a consequence, the command line options are currently explorer specific: they are different for each individual explorer.

EXAMPLE

In this example we show a session with the symbolic explorer for LOTOS (which uses the symbolic LOTOS simulator smile). In the session we use the **r** command to get the initial state (transition), which we explore using the **e** command; we instantiate one of the events two times, using different values for the free variables, and we explore the result of both the instantiations, one (10) only one ''step'' deeper, the other (5) until we have traverse all internal (invisible) transitions leading from it.

One line in the output of that explorer was too long to fit on a single line; we use a continuation mark \ to indicate that an output line continues on the next line.

```
$ smileexp cf-pe-sut-smile.cr
r
R 0  1
e 0
EB
Ee   3  1  1  udp_in ! udp2 ! udp_datareq(udp1, var_PDU$1)       var_PDU$1 Smile_227_0 PDU
Ee   2  1  1  udp_in ! udp3 ! udp_datareq(udp1, var_PDU$1)       var_PDU$1 Smile_226_0 PDU
Ee   1  1  1  CFSAP_in ! cf1 ! join(var_UserTitle$1, var_ConfIdent$1)            \
   var_UserTitle$1 Smile_224_0 UserTitle var_ConfIdent$1 Smile_225_0 ConfIdent
EE
i 1 Smile_225_0 = ut_A  ; Smile_224_0 = ci_one
I  4  1  1  CFSAP_in ! cf1 ! join(ut_A, ci_one) Smile_225_0 = ut_A  ; Smile_224_0 = ci_one      1
e 4
EB
Ee   9  1  1  CFSAP_in ! cf1 ! datareq(var_DataField$1)     var_DataField$1 Smile_230_0 DataField
Ee   8  1  1  CFSAP_in ! cf1 ! leave
Ee   7  1  1  udp_in ! udp3 ! udp_datareq(udp1, var_PDU$1)   var_PDU$1 Smile_229_0 PDU
Ee   6  1  1  udp_in ! udp2 ! udp_datareq(udp1, var_PDU$1)   var_PDU$1 Smile_228_0 PDU
Ee   5  0  1  i ! cf1 ! join(ut_A, ci_one)
EE
i 1 Smile_225_0 = ut_B  ; Smile_224_0 = ci_two
I  10   1  1  CFSAP_in ! cf1 ! join(ut_B, ci_two) Smile_225_0 = ut_B  ; Smile_224_0 = ci_two      1
e 10
EB
Ee   15   1  1  CFSAP_in ! cf1 ! datareq(var_DataField$1)     var_DataField$1 Smile_233_0 DataField
Ee   14   1  1  CFSAP_in ! cf1 ! leave
Ee   13   1  1  udp_in ! udp3 ! udp_datareq(udp1, var_PDU$1)       var_PDU$1 Smile_232_0 PDU
Ee   12   1  1  udp_in ! udp2 ! udp_datareq(udp1, var_PDU$1)       var_PDU$1 Smile_231_0 PDU
Ee   11   0  1  i ! cf1 ! join(ut_B, ci_two)
EE
e 5
EB
Ee   21   0  1  i ! udp1 ! udp_datareq(udp2, PDU_J(ut_A, ci_one))
Ee   20   0  1  i ! udp1 ! udp_datareq(udp3, PDU_J(ut_A, ci_one))
Ee   19   1  1  udp_in ! udp3 ! udp_datareq(udp1, var_PDU$1)       var_PDU$1 Smile_236_0 PDU
Ee   18   1  1  udp_in ! udp2 ! udp_datareq(udp1, var_PDU$1)       var_PDU$1 Smile_235_0 PDU
Ee   17   1  1  CFSAP_in ! cf1 ! datareq(var_DataField$1)     var_DataField$1 Smile_234_0 DataField
Ee   16   1  1  CFSAP_in ! cf1 ! leave
EE
e 20
EB
Ee   27   0  1  i ! udp1 ! udp_datareq(udp2, PDU_J(ut_A, ci_one))
Ee   26   1  1  udp_out ! udp3 ! udp_dataind(udp1, PDU_J(ut_A, ci_one))
Ee   25   1  1  udp_in ! udp2 ! udp_datareq(udp1, var_PDU$1)       var_PDU$1 Smile_239_0 PDU
Ee   24   1  1  udp_in ! udp3 ! udp_datareq(udp1, var_PDU$1)       var_PDU$1 Smile_238_0 PDU
Ee   23   1  1  CFSAP_in ! cf1 ! datareq(var_DataField$1)     var_DataField$1 Smile_237_0 DataField
Ee   22   1  1  CFSAP_in ! cf1 ! leave
EE
e 21
EB
Ee   33   0  1  i ! udp1 ! udp_datareq(udp3, PDU_J(ut_A, ci_one))
```

```
Ee   32   1  1  udp_out ! udp2 ! udp_dataind(udp1, PDU_J(ut_A, ci_one))
Ee   31   1  1  udp_in ! udp2 ! udp_datareq(udp1, var_PDU$1)      var_PDU$1 Smile_242_0 PDU
Ee   30   1  1  udp_in ! udp3 ! udp_datareq(udp1, var_PDU$1)      var_PDU$1 Smile_241_0 PDU
Ee   29   1  1  CFSAP_in ! cf1 ! datareq(var_DataField$1)     var_DataField$1 Smile_240_0 DataField
Ee   28   1  1  CFSAP_in ! cf1 ! leave
EE
e 27
EB
Ee   39   1  1  udp_out ! udp2 ! udp_dataind(udp1, PDU_J(ut_A, ci_one))
Ee   38   1  1  udp_out ! udp3 ! udp_dataind(udp1, PDU_J(ut_A, ci_one))
Ee   37   1  1  udp_in ! udp3 ! udp_datareq(udp1, var_PDU$1)      var_PDU$1 Smile_245_0 PDU
Ee   36   1  1  udp_in ! udp2 ! udp_datareq(udp1, var_PDU$1)      var_PDU$1 Smile_244_0 PDU
Ee   35   1  1  CFSAP_in ! cf1 ! datareq(var_DataField$1)     var_DataField$1 Smile_243_0 DataField
Ee   34   1  1  CFSAP_in ! cf1 ! leave
EE
e 33
EB
Ee   45   1  1  udp_out ! udp3 ! udp_dataind(udp1, PDU_J(ut_A, ci_one))
Ee   44   1  1  udp_out ! udp2 ! udp_dataind(udp1, PDU_J(ut_A, ci_one))
Ee   43   1  1  udp_in ! udp3 ! udp_datareq(udp1, var_PDU$1)      var_PDU$1 Smile_248_0 PDU
Ee   42   1  1  udp_in ! udp2 ! udp_datareq(udp1, var_PDU$1)      var_PDU$1 Smile_247_0 PDU
Ee   41   1  1  CFSAP_in ! cf1 ! datareq(var_DataField$1)     var_DataField$1 Smile_246_0 DataField
Ee   40   1  1  CFSAP_in ! cf1 ! leave
EE
q
Q
```

NOTE

The interface can likely be simplified with respect to the *freevars* fields: we should investigate whether or not we can 'hide' the 'original' free-variable names inside the explorer, and reduce the three-tuples of

normalised-varname, *original-varname*, *type*

to tuples of

varname, *type*

Question: would it be wise to change the interface such that the ''identical'' fields in the output of commands that ''generate'' events are never empty, but always contains the canonical state identifier -- which is identical to the transition identifier for transitions that are the ''first'' to point to a state? In that case, the user can *always* use the value of the field ''identical'' as canonical state identification (instead of having to check first if it is empty, and in that case using the transition identifier).

SEE ALSO

torx-intro(1), **mkprimer**(1), **torx-adaptor**(5), **torx-primer**(5), **torx-config**(4), **torx-log**(4), **torx**(1)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of **torx**.

NAME

torx-instantiator − a program that implements an instantiator

SYNOPSIS

INSTANTIATOR

NOTE

This is all very new (and experimental), such that the default **instantiator**(1) *does not (yet) use this interface!*

DESCRIPTION

In this man page we discribe the interface of the INSTANTIATOR module. The interface is very similar to the one of the PRIMER and the ADAPTOR (See **torx-primer**(5) and **torx-adaptor**(5)).

When active, the INSTANTIATOR receives commands from the standard input and writes answers in return to the standard output. See section **COMMANDS** for the available commands and answers. When the INSTANTIATOR is used in the TORX tool it is connected to the DRIVER which has the control and initiative of all communications. When the INSTANTIATOR is used standalone the user plays the role of the DRIVER.

A command consist of a single line of text. The keyword (the first word) of each commands start with a prefix **C_**.

An answer consist of a single line of text or a multi-line of text. All single-line answers start with a keyword that has prefix **A_**. Usually, the keyword of single-line answer is identical to the command-keyword, but with the **C_** replaced by **A_**. A multi-line answer has specific markers around the body of the answer. These markers consist of a single line that starts with a prefix **A_**, followed by the command-keyword without the initial **C_,** followed by **_START** respectively **_END**. Each line of the body of a multi line answer often starts with a keyword (but not always).

Instead of the usual single- or multi-line answer, an **A_ERROR** answer may occur in case of an error. Such an **A_ERROR** answer comes instead of the usual single- or multi-line answer.

Additional message lines containing warnings (to be shown to the test run operator), diagnostics (to appear in the test run log), and/or debugging output (to be shown to the test run operator) may appear interspersed with the answer lines described so far. These message lines start with **A_WARNING**, **A_LOG** or **A_DEBUG**, respectively. Because these message lines may appear arbitrarily intermixed with the ‘‘regular’’ answer line(s), they do not play a role when the ‘‘end’’ of an answer has to be found: once the command-specific answer or **A_ERROR** answer has been read, the user of the INSTANTIATOR may assume that no other output (specific to to the command) will be generated by the INSTANTIATOR until the next command is send to it.

If the INSTANTIATOR decides autonomously that it will stop executing, it may try to indicate this by sending an **A_QUIT** answer, just before exiting. For the INSTANTIATOR, reading end-of-file on its standard input is a reason to stop executing.

In all cases, each command and each answer starts with a keyword that is separated from the remainder of the line by one or more spaces or tabs − except for two notable exceptions: **A_ERROR**, where currently exactely one space should be between the keyword and the category, and the lines between **A_STATE_START** and **A_STATE_END**, that do not start with a keyword (see **DETAILS** below). For most comands and most answers, the text following the keyword has a fixed format; for other commands and answers it is left unspecificed.

For command and answers that deal with channels, events, predicates, etc. the format of each line is as follows: the keyword is followed by a (possibly empty) TAB-separated list of *name=value* items, where the *name* cannot contain TABs or = characters, and the *value* cannot contain TAB characters. For the moment we assume that, when necessary, we will encode TAB characters for example using C conventions. We have chosen for this kind of item-lists because this makes it easy to pass information. Using *name=value* pairs instead of (for example) a list of *values* in a fixed order makes the interface more robust: the order of items is not important, and the users can simply ignore items which *name* they don’t recognize. This

allows one party to send items which the receiving party doesn't know about. This makes it easier to extend the interface by adding new items and keeping old programs backwards compatible.

For **A_ERROR** answers, currently the format is as follows: the A_ERROR keyword is followed by a category word indicating the category of the error, followed by arbitrary text that describes the error in more detail. The idea is that the user of the INSTANTIATOR can use the category word to get some idea of the nature of the error. In particular, the idea is that the user can use this to decide whether it makes sense to retry, or whether it is better to give up.

DETAILS

Here we describe first the syntax of the (textual) messages interchanged over the interface, followed by the possible message arguments, followed by the possible error categories.

SYNTAX

Below we describes the syntax of the commands and answers, by giving the syntax of just a single interaction.

interaction:
message∗ *command message*∗ *answer-or-error*

command: C_*name arguments-upto-end-of-line* nl

answer-or-error:
single-line-answer | *multi-line-answer* | *error*

single-line-answer:
A_*name arguments-upto-end-of-line* nl

multi-line-answer:
A_*name*_START nl *multi-line-element*∗ A_*name*_END nl

multi-line-element:
message | *result-element*

result-element:
A_*element-name arguments-upto-end-of-line* nl

message: *debug* | *log* | *warning*

debug: A_DEBUG *text-upto-end-of-line* nl

log: A_LOG *text-upto-end-of-line* nl

warning: A_WARNING *text-upto-end-of-line* nl

error: A_ERROR space *category* : *text-upto-end-of-line* nl

arguments-upto-end-of-line:
ws∗ | *ws*+ *arguments*

arguments: *argument* | *argument* tab *arguments*

argument: *name = value*
/∗ value can contain spaces but no tabs ∗/

text-upto-end-of-line:
ws∗ | *ws*+ *arbitrary-text-without-newline*

ws: space | tab

ARGUMENTS

At the moment the following *names* are defined for the *name=value* arguments to the commands:

event with as value the LOTOS-like string representation of an event, used wherever ‘‘event’’ is used below;

channel with as value a channel name;

iokind where *iokind* is a value **input** or **output**;

suspension with as value either ‘‘0’’ or ‘‘1’’ to indicate a non-suspension resp. suspension event, used wherever an output ‘‘event’’ is used below (in the future we expect to use it also for input events)

pco with as value a pco name;

concrete with as value a (string representation of the) concrete value (this string cannot contain tab or new-line characters);

timestamp with as value a (string representation of the) time

ERRORS

At the moment, for the A_ERROR *category* messages given by the INSTANTIATOR, the following *categories* are defined. The user of the INSTANTIATOR (i.e. torx) may use these categories to decide how to react to the **A_ERROR** answer (in particular, to decide whether it can cope with it, or whether it is better to give up).

UnknownCommand the command is not known

ArgumentMissing a mandatory argument of the command is missing or incomplete

WrongValue the value of an argument of the command is outside its domain of valid values

ParseErrorEvent the command contains an **event** argument which cannot be parsed

UnknownIOKind the command contains an **event** argument, and the **iokind** of the event cannot be computed

Inconsistency the command contains more than a single argument, and the values of (some of) the arguments conflict which each other

InternalError an internal error occurred during the execution of the command

COMMANDS

Below follows the list of the commands. The commands, and their answers, are described in more detail below. With each command, and answer, the possible arguments are indicated. Optional arguments are surrounded by brackets ’[’ and ’]’. Note: the order in which the arguments are named with the commands does *not* prescribe the *order* in which arguments may appear: arguments may appear in any order.

C_EVENT [event] predicates
C_GETCONFIG

The C_EVENT command is the most important one, and has to be implemented correctly for TorX to work correctly; the other commands are less important, and most of them are mainly used to get configuration information from the INSTANTIATOR, only for logging purposes.

In the subsections below, each command is followed by a short explanation of its use. Additional comments are enclosed between /∗ and ∗/ after the command.

EVENT

The C_EVENT command is used to ask the INSTANTIATOR to generate a solution for the given predicates. Other fields may be given, and will appear unchanged in the answer. This command has no side-effects, and can repeatedly be applied without changing the state of the INSTANTIATOR module.

Command:
C_EVENT [event] predicates

Answer (single-line):
A_EVENT [event] predicates

Errors (single-line):
A_ERROR WrongValue */∗if a given field has value outside domain∗/*

GETCONFIG

This command has not been implemented yet. It is to be used to get configuration information from the INSTANTIATOR.

IMPLEMENTATION NOTES

For the implementer of an instantiator, the most important command to implement is C_EVENT.

For all the other commands, the implementer can choose to give simple answers, by returning just the answer keyword without parameters, for single-line answers, and returning just the answer start and answer end keywords for multiline answers. Of course, more informational answers proved more feedback during testing, and may make it more easy to investigate in case of errors -- the information above is mainly meant to make it easier to quickly implement a ‘‘basic’’ instantiator that already ‘‘does the right thing’’.

SEE ALSO

torx-intro(1), **mkprimer**(1), **torx-primer**(5), **torx-adaptor**(5), **torx-config**(4), **torx-log**(4), **torx**(1), **sh**(1)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of **torx**.

NAME

torx-primer − interface to program that derives test primitives from labelled transition system

SYNOPSIS

PRIMER [*options* **]**

DESCRIPTION

In this man page we discribe the interface of the PRIMER module. The PRIMER module, a UNIX program together with the EXPLORER module, implements an (textual) interface of a labelled transition system (lts).

When active, the PRIMER receives commands from the standard input and writes answers in return to the standard output. See section **COMMANDS and ANSWERS** for the available commands and answers. When the PRIMER is used in the TORX tool it is connected to the DRIVER which has the control and initiative of all communications. When the PRIMER is used as standalone the user plays the role of the DRIVER.

A command consist of a single line of text. The keyword of all commands start with a prefix **C_**.

A answer consist of a single line of text or a multi-line of text. All single line answers start with a prefix **A_**. A multi-line answer has specific markers around the body of the answer. These markers consist of a single keyword at a single line, made up from the usual answer keyword, with **_START** and **_END** appended, respectively. Each line of the body of a multi line answer often starts with a keyword (but not always). Usually, the PRIMER answers to a command with the identical command-keyword, but with the **C_** replaced by **A_**. Instead of the usual answer, an A_ERROR answer may occur in case of an error.

The following describes the syntax of the commands and answers:

interaction:
message∗ *command message*∗ *answer-or-error*

command:
C_*name arguments-upto-end-of-line* nl

answer-or-error:
single-line-answer | *multi-line-answer* | *error*

single-line-answer*:*
A_*name arguments-upto-end-of-line* nl

multi-line-answer:
A_*name*_START nl *multi-line-element*∗ A_*name*_END nl

multi-line-element:
message | *result-element*

result-element:
A_*element-name arguments-upto-end-of-line* nl

message:
debug | *log* | *warning*

debug: A_DEBUG *text-upto-end-of-line* nl

log: A_LOG *text-upto-end-of-line* nl

warning:
A_WARNING *text-upto-end-of-line* nl

error*:* A_ERROR *text-upto-end-of-line* nl

For some commands and some answers, the text following the keyword has a fixed format; for other commands and answers it is left unspecificed.

For command and answers that deal with channels, events, predicates, etc. the format of each such line is as follows: the keyword is followed by a (possibly empty) TAB-spearated list of *name=value* items, where the *name* cannot contain TABs or = characters, and the *value* cannot contain TAB characters. Note that the order of the *name=value* items is left unspecified, i.e. they may appear in any order, even though the presentation below, in the detailed specification of the commands and their answers, might suggest otherwise. For the moment we assume that, when necessary, we will encode TAB characters for example using C conventions. We have chosen for this kind of item-lists because this makes it easy to pass information. Using *name=value* pairs instead of (for example) a list of *values* in a fixed order makes the interface more robust: the order of items is not important, and, provided the users ignore items which *name* they don't recognize. This allows one party to send items which the receiving party doesn't know about. This makes it easier to extend the interface by adding new items and keeping old programs backwards compatible.

At the moment the following *names* of the *name=value* items, are defined:

event
: with as value the LOTOS-like string representation of an event, used wherever ''event'' is used below;

channel
: with as value a channel name;

iokind
: where *iokind* is a value **input** or **output**;

suspension
: with as value either ''0'' or ''1'' to indicate a non-suspension resp. suspension event, used wherever an output ''event'' is used below (in the future we expect to use it also for input events)

super
: with as value a number that is used to represent a state-vector;

init
: with as value a comma-separated list of those state numbers that were directly reached by an observable transition;

trans
: with as value a comma-separated list of those state numbers that were reached from **init** via (an) internal (non-observable) transition(s).

Note1:
: ''predicates'' is currently only implemented in the generic **primer**(1).

Note2:
: we may want to split the ''predicates'' into two (or more?) groups, depending on the complexity of the operation that we want (need) to perform on them. We could for example have one group that consists of equations, for which we need a single solution (witness), and another group consists of equations, for which we need proof that no solution exists.

Note3:
: The lists of comma-separated state numbers of **init** and **trans** may contain abbreviations of the form *first*-*last* as abbreviation of strictly increasing sequences of the form *first*, (*first*+1), (*first*+2), ..., (*last*-2), (*last*-1), *last*, where *last* is equal to or greater than (*first*+1).

Answers are outputted after the receival of a command, but always before the command-specific answer or A_ERROR answer. I.e. once the command-specific answer or A_ERROR answer has been read, the user of

the PRIMER may assume that no other output will be generated by the PRIMER until the next command is send to it.

Besides the "normal" (expected) answers the PRIMER can also respond with error and warning messages as answers.

Currently known errors:

A_ERROR UnknownCommand
A_ERROR ArgumentMissing
/*if arguments are missing or incomplete*/
A_ERROR ParseErrorEvent
/*if the event cannot be parsed*/
A_ERROR UnknownIOKind
/*if iokind of event cannot be found*/
A_ERROR Inconsistency
/*if fields conflict*/
A_ERROR WrongValue
/*if a given field has value outside domain*/
A_ERROR InternalError
/*if internal error occurred*/

In principle, each answer of the PRIMER can contain one or more diagnostic lines, indicating errors, warnings, end-of-test, or guidance information. Below some examples of ansers:

A_WARNING
A_EOT
A_GUIDANCE_VERDICT INCONSISTENT
A_GUIDANCE_VERDICT INCONCLUSIVE
A_GUIDANCE_VERDICT PASS

For logging of debugging information the following answers may appear anywhere.

A_DEBUG
A_LOG

OPTIONS

The PRIMER module support the following commandline options:

-s *number*
the seed for the random number generator

-i *gates1,gate2,gate3,...*
the list of input gates. Note there are no spaces between the gates!

-o *gates1,gate2,gate3,...*
the list of output gates. Note there are no spaces between the gates!

-S *algorithm*
the algorithm which can be **ioco**, **traces** or **simulation**. By default this is **ioco**. Note that this option is only implemented in LOTOS PRIMERs.

-d *delta-event-tag*
the *delta-event-tag* is used for quiescense in the interface. By default the tag is ‘‘Delta’’. Note that this option is not needed for the *traces* algorithm, and that this option is only implemented in the LOTOS PRIMERs.

COMMMANDS and ANSWERS

The core functionality of the PRIMER is given by the following commands which use the syntax described above (DESCRIPTION). (The brackets [and] indicate optional elements):

C_IOKIND [iokind] [channel]
C_GETINPUT [channel] [event] [predicates]
C_INPUT [channel] [event] [predicates]
C_OUTPUT [channel] [event] [predicates]
C_QUIT

Below follows the list of the all commands, which are described in more detail in the following subsections:

C_CHANNELS [iokind] [channel]
C_IOKIND [iokind] [channel]
C_GETINPUT [channel] [event] [predicates]
C_INPUT [channel] [event] [predicates]
C_GETOUTPUT [channel] [event] [predicates]
C_OUTPUT [channel] [event] [predicates]
C_INPUTS [channel]
C_OUTPUTS [channel]
C_STATE
C_STATEID
C_GOTO [super]
C_STATS
C_QUIT
C_GUIDANCE_INFO
C_GUIDANCE_QUIT
C_GETCONFIG
C_EOT

In the subsections below each command is followed by a short explanation of its use. Additional comments are enclosed between /* and */ after the command.

CHANNELS

This command has only been implemented for the Smile PRIMER. The C_CHANNELS command is used to ask the PRIMER which channels exist. If the *iokind* and/or *channel* argument is used, then only the respective channels are returned. This command has no side-effects, and can repeatedly be given without changing the state of the EXPLORER and PRIMER modules.

Commands:

C_CHANNELS [iokind] [channel]

Answers:

A_CHANNELS_START
    A_CHANNEL iokind channel **...**
        ...
A_CHANNELS_END
A_ERROR Inconsistency
    /*if the iokind and channel fields conflict*/
A_ERROR WrongValue
    /*if a given field has value outside domain*/

where iokind is either INPUT or OUTPUT.

IOKIND

The C_IOKIND command is used to ask the PRIMER whether the next action should be an input or an output action. The command can have an optional suggestion, which will be honoured by the PRIMER if possible. This command has no side-effects (apart from calling the random number generator), and can repeatedly be given without changing the state of the EXPLORER and PRIMER modules.

Commands:

C_IOKIND [iokind] [channel]

Answers:

A_IOKIND iokind channel
A_ERROR Inconsistency
/*if the iokind and channel fields conflict*/
A_ERROR WrongValue
/*if a given field has value outside domain*/

where iokind is either INPUT or OUTPUT.

GETINPUT

The C_GETINPUT command is used to ask the PRIMER for an input transition without doing a transition (the transition is done with the C_INPUT command). The command can have an optional suggestion for the action, which will be honoured by the PRIMER if the action is in the current list of input actions; if no action is given, an action will be randomly chosen from the 'current' list of input actions. If the action given as suggestion cannot be parsed, the PRIMER will return A_ERROR (with an explanation of the error); otherwise it will return the given or chosen action.

Commands:

C_GETINPUT [channel] [event] [predicates]

Answers:

A_GETINPUT_OK channel event predicates
A_GETINPUT_ERROR
/*if event is not in menu*/
A_ERROR ParseErrorEvent
/*if the event cannot be parsed*/
A_ERROR ArgumentMissing
/*if no event was given to simulator*/
A_ERROR UnknownIOKind
/*if iokind of event cannot be found*/
A_ERROR Inconsistency
/*if the iokind and channel fields conflict*/
A_ERROR WrongValue
/*if a given field has value outside domain*/
A_ERROR InternalError
/*if internal error occurred*/

INPUT

The C_INPUT command is used to ask the PRIMER to do an input transition. The command is given the action that has to be done, which should be in the current list of input actions (if not, A_INPUT_ERROR will be returned). If the action given cannot be parsed, the PRIMER will return A_ERROR (with an explanation of the error), and no transition will be done; otherwise it will return the action that is 'done'.

Commands:

C_INPUT [channel] event [predicates]

Answers:

A_INPUT_OK channel event predicates
A_INPUT_ERROR
/*if event is not in menu*/
A_ERROR ParseErrorEvent
/*if the event cannot be parsed*/
A_ERROR ArgumentMissing
/*if no event was given*/
A_ERROR UnknownIOKind
/*if iokind of event cannot be found*/
A_ERROR Inconsistency
/*if the iokind and channel fields conflict*/
A_ERROR WrongValue
/*if a given field has value outside domain*/
A_ERROR InternalError
/*if internal error occurred*/

GETOUTPUT

The C_GETOUTPUT command is used to ask the PRIMER for an output transition without doing a transition (the transition is done with the C_OUTPUT command). If the command does not contain an action, the SIMULATOR will randomly choose an action, and the **IOCO** and traces version will return an error message; if the command does contain an action, the PRIMER will check if it can 'do' the action (but will not actually do the action). If the action cannot be parsed, the PRIMER will return A_ERROR (with an explanation of the error). Otherwise, if the action can be performed, the PRIMER will return A_GETOUTPUT_OK; if the action cannot be performed (i.e. if the action is not in the list of current output actions) A_GETOUTPUT_ERROR will be returned.

Commands:

C_GETOUTPUT [channel] [event] [predicates]

Answers:

A_GETOUTPUT_OK channel event predicates
A_GETOUTPUT_ERROR
/*if the event is not in menu of **IOCO** */
A_ERROR ParseErrorEvent
/*if the event cannot be parsed*/
A_ERROR ArgumentMissing
/*if no event was given to **IOCO** or traces*/
A_ERROR UnknownIOKind
/*if iokind of event cannot be found*/
A_ERROR Inconsistency
/*if the channel and event fields conflict*/
A_ERROR WrongValue
/*if a given field has value outside domain*/
A_ERROR InternalError
/*if internal error occurred*/

OUTPUT

The C_OUTPUT command is used to ask the PRIMER to do an output transition. If the command does not contain an action, the SIMULATOR will randomly choose which action to do, and the **IOCO** and traces

version will return an error message; if the command does contain an action, the PRIMER will attempt to 'do' the action. If the action cannot be parsed, the PRIMER will return A_ERROR (with an explanation of the error). Otherwise, if the action can be performed, the PRIMER will return A_OUTPUT_OK; if the action cannot be performed (i.e. if the action is not in the list of current output actions) A_OUTPUT_ERROR will be returned.

Commands:

C_OUTPUT [channel] [event] [predicates]

Answers:

A_OUTPUT_OK channel event predicates
A_OUTPUT_ERROR
/*if the event is not in menu of **IOCO** */
A_ERROR ParseErrorEvent
/*if the event cannot be parsed*/
A_ERROR ArgumentMissing
/*if no event was given to **IOCO** or traces*/
A_ERROR UnknownIOKind
/*if iokind of event cannot be found*/
A_ERROR Inconsistency
/*if the channel and event fields conflict*/
A_ERROR WrongValue
/*if a given field has value outside domain*/
A_ERROR InternalError
/*if internal error occurred*/

INPUTS

The C_INPUTS command returns the list of current input actions. This list can be restricted with the optional channel argument. This command has no side-effects.

Command:

C_INPUTS [channel]

Answer (multi-line):

A_INPUTS_START
A_EVENT channel event predicates
...
A_INPUTS_END
A_ERROR WrongValue
/*if a given field has value outside domain*/
A_ERROR Inconsistency
/*if the channel has wrong iokind*/

OUTPUTS

The C_OUTPUTS command returns the list of current output actions. This list can be restricted with the optional channel argument. This command has no side-effects.

Command:

C_OUTPUTS [channel]

Answer (multi-line):

A_OUTPUTS_START
A_EVENT channel event predicates

...
A_OUTPUTS_END
A_ERROR WrongValue
/*if a given field has value outside domain*/
A_ERROR Inconsistency
/*if the channel has wrong iokind*/

STATE

The C_STATE command returns a textual representation of the current state. The contents of this textual representation depend on the implementation of the PRIMER. For the Open/Caesar implementation, the representation consists of two lines of 'legenda', followed by a line for each state in the current 'state-vector'. This command has no side-effects.

Command:

C_STATE

Answer (multi-line):

A_STATE_START
text **...**
...
A_STATE_END

STATEID

The C_STATEID command returns the current state identifier. The state identifier consists of a single number (*super*) that represents (identifies) the current state(-vector), together with a list of comma-separated state numbers of those states in the current state(-vector) that were directly reached via an observable transition (*init*), and a list of comma-separated state numbers of those states in the current state(-vector) that were reached from the *init* states via an internal (non-observable) transition (*trans*). In the lists of state numbers (sub)sequences of strictly increasing state numbers, where each element in the sequence is equal to the previous element plus one, may be abbreviated to *first-last* with *first* and *last* the first and last elements of the (sub)sequence. A PRIMER should give just one line state identifier (both single- and multi-line form are allowed). The COMBINATOR combines the state identifier of its primers, by giving the lines of its primers in a multi-line answer. When it does so, it prefixes the statistics of each primer with a name-value field *identity* that contains a white-space separated list of two white-space separated name-value pairs: one that gives the id of the primer (key: id) and a second one that gives the role of the primer (key: role, values: SPEC, GUIDE, TEST), both as specified in the configuration value of the combinator. This command has no side-effects.

Command:

C_STATEID

Answer (single-line):

A_STATEID super init trans

Answer (multi-line):

A_STATEID_START
A_STATEID identity super init trans
...
A_STATEID_END

GOTO

The C_GOTO command goes to the state identified by the given super state identifier. This state identifer must have been retrieved earlier using a STATEID command. This command does have a side-effect.

Command:

C_GOTO super

Answer (single-line):

A_GOTO_OK
A_GOTO_ERROR A_GOTO_ERROR UnknownStateID super
/*if an unknown super state was given*/
A_ERROR ArgumentMissing
/*if no super field was given*/

Note that the GOTO command is a rather recent addition. As an unfortunate result of that, the A_GOTO_ERROR response currently has different parameters in the various implementations that we have. The CADP primer only returns A_GOTO_ERROR, whereas the 'generic' primer also returns the string UnknownStateID and the failing argument super state. We hope to make this more uniform once we have more experience with this interface command.

STATS

The C_STATS command returns some statistics about the PRIMER (like: numer of states in state-vector, number of explored states, number of transitions etc.). The statistics consists of a list of whitespace-separated key-value pairs, where also the key and the value are separated by whitespace. The value should not contain whitespace. A PRIMER should give just one line of statistics (both single- and multi-line form are allowed). The COMBINATOR combines the statistics of its primers, by giving the lines of its primers in a multi-line answer. When it does so, it prefixes the statistics of each primer with two name-value pairs: one that gives the id of the primer (key: id) and a second one that gives the role of the primer (key: role, values: SPEC, GUIDE, TEST), both as specified in the configuration value of the combinator.

Command:

C_STATS

Answer (single-line):

A_STATS statistics

Answer (multi-line):

A_STATS_START
A_STATS statistics
...
A_STATS_END

QUIT

The C_QUIT command tells the PRIMER to clean up and exit. The PRIMER will acknowledge the command with A_QUIT. The side-effect of this command is that the PRIMER module exits.

Note: when this command is send to the COMBINATOR it will send a C_QUIT command to each of the PRIMERs that it controls, and then clean up and exit, without waiting for the responses of the PRIMERs. In this respect this command differs from the C_GUIDANCE_QUIT command (see below).

Command:

C_QUIT

Answer:

A_QUIT

GUIDANCE_INFO

This command is currently not used by the DRIVER; it has been implemented in the COMBINATOR. The C_GUIDANCE_INFO command asks the COMBINATOR to give the (tab-separated) ‘‘id’’ and ‘‘role’’ (SPEC, GUIDE, or TEST) for each of the PRIMERs that it is responsible for.

Command:

C_GUIDANCE_INFO

Answer:

A_GUIDANCE_INFO_START
    A_GUIDANCE_INFO id role
    ...
A_GUIDANCE_INFO_END

GUIDANCE_QUIT

This command is currently not used by the DRIVER; it has been implemented in the COMBINATOR. The C_GUIDANCE_QUIT command asks the COMBINATOR to send a C_QUIT command to each of the PRIMERs that it is responsible for, and wait for the answers. In this respect this command differs from the C_QUIT command (see above).

Command:

C_GUIDANCE_QUIT

Answer:

A_GUIDANCE_QUIT

GETCONFIG

This command has not been implemented yet. Get configuration information from the COMBINATOR (and via it, from its primers) or the primer.

EOT

This command is currently not used by the DRIVER; it has been implemented in the PRIMER and COMBINATOR. The C_EOT command asks the PRIMER whether ’end-of-test’ has been reached. The PRIMER will return A_EOT_YES if the lists of current actions are empty, or if the list of input actions is empty and the list of output actions contains only ‘delta‘. This command has no side-effects.

Command:

C_EOT

Answer:

A_EOT_YES
A_EOT_NO

SEE ALSO

torx-intro(1), **mkprimer**(1), **torx-adaptor**(5), **torx-config**(4), **torx-log**(4), **torx**(1)

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of **torx**.

NAME

xtorx-extension − api to specify the Mutants, Primers and Guides menus of xtorx

SYNOPSIS

MUTANT *entry title setCommand unsetCommand*
PRIMER *entry title setCommand unsetCommand*
GUIDE *entry title setCommand unsetCommand*
setCommand flagsVarName configVarName
unsetCommand flagsVarName configVarName

DESCRIPTION

The **xtorx-primers** and **xtorx-mutants** configuration file are actually **tcl** scripts, that allow the user to register primers and mutants with **xtorx**(1). The user is free to use whatever tcl commands needed in the **xtorx-primers**, **xtorx-guides** and **xtorx-mutants** scripts, as long as the net effect is that sourcing the script results in the invocation of the **PRIMER**, **GUIDE** and/or **MUTANT** routines that are necessary: one invocation for each primer respectively mutant that is to be registered. Additionally, the script should provide the commands used in the *setCommand* and *unsetCommand* arguments of **PRIMER**, **GUIDE** and/or **MUTANT**.

To register a primer means: invoking **PRIMER** with two text strings, and two tcl commands. To register a guide means: invoking **GUIDE** with two text strings, and two tcl commands. To register a mutant means: invoking **MUTANT** with two text strings, and two tcl commands. The *entry* string will appear in the *Primers*, *Guides* respectively *Mutants* menu; the *title* string will appear in the *Window Title* when the primer, guide or mutant is selected (and thus: active). An *entry* string containing just the single word **none** is special: it will be the one that is initially selected (active). The *setCommand* will be invoked by **xtorx**(1) when the primer, guide or mutant is selected (chosen) by the user, and the *unsetCommand* when the primer, guide or mutant is deselected by the user. When **xtorx**(1) invokes *setCommand* and *unsetCommand* it extends them with two arguments: *flagsVarName* and *configVarName*. Both these arguments can be used to change the command that will be used to invoke **torx**(1). The *flagsVarName* argument can be used to insert options (flags) and the *configVarName* argument can be used to insert configuration files into the command line that will be used to invoke **torx**(1). How they can be used is shown in the examples, below.

In addition to manipulating the command line arguments of **torx**(1), the commands that are called by *setCommand* and *unsetCommand* can, as needed, change the environment in which **torx**(1) will be invoked, by creating and/or deleting files, and/or by setting and unsetting environment variables. Usually, the *setCommand* command will make a change in the environment, such that when **torx**(1) is invoked, the right mutant is activated, and the *unsetCommand* command will undo this change in the environment, such that a subsequent invocation of **torx**(1) will cause the ''default'' implementation to be activated, and such that a subsequent invocation of a *setCommand* command will activate its corresponding mutant.

EXAMPLES

The examples below all register mutants. Adapting them to primers or guides is left as an exercise to the reader.

The following example registers a single mutant. When the mutant is selected, the environment variable *MUTANT* is set to 1. Here *setCommand* and *unsetCommand* just contain a tcl command.

```
proc setMut {flagsvar configvar} {
    global env
    set env(MUTANT) 1
}
proc unsetMut {flagsvar configvar} {
    global env
    set env(MUTANT) 0
}
MUTANT mut1 mut1 setMut unsetMut
```

In the example below, *setCommand* and *unsetCommand* are more than just proc names: here the *setCommand* consists of a a string containing a command together with an additional argument. In this case the command mentioned in *setCommand* must of course handle this additional argument. The following example registers two mutants, mut1 and mut2, for which the environment variable *MUTANT* is set to 1 respectively 2. When no mutant is selected, environment variable *MUTANT* will not be set.

```
proc setMut {nr flagsvar configvar} {
    global env
    set env(MUTANT) $nr
}
proc unsetMut {flagsvar configvar} {
    global env
    catch {unset env(MUTANT)}
}
MUTANT mut1 mut1 "setMut 1" unsetMut
MUTANT mut2 mut2 "setMut 2" unsetMut
```

Finally, we show how tcl commands can be used to register a number of mutants, and how the *torxConfigVarName* argument can be used to manipulate the **torx**(1) command line by adding an additional (mutant specific) configuration file.

```
proc setMut {nr flagsvar configvar} {
    global env
    upvar $flagsvar flags
    upvar $configvar config
    set env(MUTANT) $nr
    if {[file exists "myMutant.$nr.config"]} {
        set config "myMutant.$nr.config"
    }
}
proc unsetMut {nr flagsvar configvar} {
    global env
    upvar $flagsvar flags
    upvar $configvar config
    catch {unset env(MUTANT)}
    catch { unset config }
}
foreach m {0 1 2 3 4 5 6 7 8 9} {
    MUTANT mut$m Mut$m "setMut $m" "unsetMut $m"
}
```

SEE ALSO

torx-intro(1), **xtorx**(1)

BUGS

The output produced by **puts** *string* respectively **puts stderr** *string* commands in the **xtorx-primers** configuration file is written to standard output respecitvely standard error; it would be nice to redirect it to the **xtorx**(1) *Messages* pane or to a dialog box.

It is important to put **catch** around **puts** commands, because **puts** may fail when the window from which **xtorx**(1) was started is no longer there.

Currently **xtorx**(1) contains separate entries to load the primers, guides and mutants; these could be combined.

All primers, guides and mutants files are loaded in the same **tcl** name space, which means that a file opened later in **xtorx**(1) may redefine (override) routines defined in a file opened earlier − it is up to the user to avoid problems by choosing unique names.

If there are multiple entries named **none** in a menu, the last one will be the one that is selected initially.

CONTACT

By Email: <torx_support@cs.utwente.nl>

VERSION

This manual page documents version 3.9.0 of **xtorx**.