Applying TorX

Axel Belinfante

10/10/2005

Abstract

In this document we describe how to set up an experiment with TorX. The pdf version of this document contains hyperlinks to on-line reference information.

1 Introduction

We start this document with some general background information about TorX, after which we describe in more detail how to set up an experiment with it.

The TorX (unix style) manual pages are the definite source of information providing the details that we try to omit here for clarity. The `torx-intro(1)` manual page[1] gives an overview of the manual pages. Where appropriate the manual pages refer to each other (and `torx-intro(1)` refers to all of them).

The manual pages are available in several forms. The TorX distribution contains them as unix manual pages. The TorX website contains them in html, and there the references between the manual pages are made into hyperlinks. The TorX website also has postscript and pdf files that contain all manual pages – unfortunately these do not have the hyperlinks (it would be great if we could add those to the pdf).

1.1 What is TorX?

TorX is a tool for specification-based blackbox conformance testing. It does the testing in an "on-the-fly" (or "on-line") manner, by which we mean that test derivation and test execution are done together – each test step is derived on demand when the test execution needs it.

1.2 How is TorX organised?

TorX has a flexible architecture. It does not consist of one (big) single program, no, it consists of a number of specialized programs. Essentially, each component in the TorX architecture is mapped onto a separate program. (There are a few

[1] This is the usual unix way of referring to a manual page: when we write `torx-intro(1)` we mean manual page `torx-intro` of section `1` of the manual, which contains user commands.

exceptions to this, which we (intend to) discuss at a more appropriate point.) Each of these programs may need configuration information to be able to do its job.

1.3 How uniform is configuration of TorX components?

We have great plans to make the configuration of TorX and its programs more uniform, but unfortunately that is no reality yet. In general, programs can take configuration information from command line options, and/or (a) configuration file(s) (e.g. specified on the command line), or even from environment variables (these are not used much in the configuration of TorX). In TorX, usually, "simple" things are configured via command line options, and more complex configuration is done via (a) configuration file(s). It is very seldom that environment variables are used.

Which programs are needed, and how they are configured, depends on the particular "experiment" that we want to do. In this document we will describe the basic set-up. The basic set-up consists of a model (or specification), an implementation to test, together with the main test driver, the components that we need to access the model and derive test steps from it, and a component to connect to (interface with) the implementation under test. We may spent some time at the end to discuss more elaborate set-ups, e.g. involving a test-purpose to guide the testing.

1.4 Where do the TorX components come from?

Some of the programs used in an experiment are generic, and part of the TorX distribution. Other programs are specific, for example specification-specific or implementation-under-test-specific. Such programs are not part of the TorX distribution. Some of these programs can be generated by TorX from information supplied by the user. Other programs have to be supplied by the user.

For example, in general the program(s) that TorX uses to access a specification ("primers" in TorX-speak) can be generated using a program "mkprimer". More about mkprimer later. Programs that typically have to be supplied (written) by the user are those that TorX needs to access the implementation-under-test ("adapters" in TorX-speak).

1.5 What is the basic TorX set-up?

needs definite rewrite

The basic set-up of TorX is as follows. The actual test derivation and execution is driven by the program `torx(1)` of the TorX distribution. The TorX distribution also contains `xtorx(1)` which is a graphical user-interface wrapper around this torx program – whenever you press the "Start" button in `xtorx`, `torx` is started. `torx` uses a `primer` program (see `torx-primer(5)`) to access the specification and derive test steps from it, and an `adapter` program (see `torx-adaptor(5)`) to access the implementation. (Where we write `primer` here, the picture may be slightly more complex: the `primer` we talked about

so far usually consists of a (specification-independent) `primer` "proper" that is used to derive test steps, and a separate (specification-language dependent) `explorer` program (see `torx-explorer(5)`), which the `primer` "proper" uses to access the specification. When we use the `mkprimer(1)` program to generate a `primer` it hides this distinction and just generates a program (shell script) that does all.) Which `primer` and which `adapter` to use is specified in a configuration file for `torx`. When `torx` is started it reads its configuration file(s) and starts the `primer` and `adapter` programs. The configuration file(s) for `torx` can be handwritten or (partly) generated. When `xtorx` is used to start `torx`, `xtorx` may generate part of the configuration information for `torx` in (a) separate configurarion file(s) that it gives to `torx` when it starts it. This is used in some of the torx-demos, where the user can choose a `primer` and/or implementation (`mutant`) from a menu in xtorx before pressing the "Start" button.

1.6 How do TorX programs delegate work?

move directly after: how is torx organized?

The general way in which TorX programs delegate work to other programs is by starting (forking) the other programs, and then giving commands to these programs by writing lines of text to their standard input and reading the responses (answers) from the standard output of the programs. So, during an actual test run the programs that are running are connected in the form of a tree in which each running program (process) is the parent of the programs that it started to delegate work to. The root of this tree is `torx` (or `xtorx`). The nodes of the tree are the running programs (processes) and the edges of the tree are the pipes over which the parents communicate with their children.

Details about the commands that `torx` sends to a `primer` and `adapter`, and the answers that it expects from them, can be found in `torx-primer(5)` respectively `torx-adaptor(5)`. Details about the commands that a `primer` sends to an `explorer` and the answers that it expects from it, can be found in `torx-explorer(5)`.

The configuration of each of the programs consists of the information the program needs to do its own job, and, if it delegates work to other programs, the information that it needs to start these programs like their names, their command line options etc.

1.7 What configuration information is needed?

The main program, `torx`, delegates most of its work to other programs. In the basic set-up that we discuss here, these programs are the `primer` and the `adapter`. So, `torx` will have to know how to start them. Another important item of the `torx` configuration is the name of the file into which to store the test log. We will need this to analyse the results.

Other items of the `torx` configuration include for example the seed of the random number generator, or which visualization programs should be started. In general for these items the default values will be sufficient.

Also the `primer` and `adapter` usually need configuration information. The `primer` needs to know which actions (events, labels) of the specification represent stimuli that can be given as inputs to the implementation, and which of them represent observations (outputs) from the implementation. The `adapter` may start the implementation-under-test, or it may connect to an implementation that is already running. In both cases it needs to know how to access the implementation (which interfaces, host names, protocols, port names?), and in the first case it also needs to know which program to start, with which command line options.

In the remainder of this document we will discuss in more detail how to get resp. write a `primer` and `adapter`, and how to configure `torx`.

2 How to do an experiment with TorX?

A highlevel overview of the steps we usually take to do an experiment can be found in section 2, "Methodology", of [dVBF02]. To do an experiment, the steps are (not necessarily in this order):

1. get (or make) specification
2. make directory structure for the experiment
3. generate a primer program from/for specification
4. get (or make) an implementation
5. write or find an adapter program
6. write the configuration file(s) for TorX
7. run the experiment (using torx or xtorx)
8. (optionally) run the experiment multiple times using torx in batch mode item analyze results

We discuss them one by one below.

2.1 get specification

Make a specification in LOTOS, Promela, Aldebaran (.aut). FSP (for LTSA) or in another language supported by TorX (see 'What languages can we use?' on page 5).

Specification-language specific comments can be found in the tool-specific manual pages for `mkprimer(1)`, e.g. see `mkprimer-cadp(1)`, `mkprimer-ltsa(1)`, `mkprimer-trojka(1)`, and in the manual pages about the explorer programs themselves: `autexp(1)`, `ltsaexp(1)`, `smileexp(1)`. (Unfortunately, there is no `mkprimer-autexp(1)` yet; we do have `autexp(1)`.) These pages document the various `explorer` or `explorer/primer` programs that we have. It may be the

case that a language is supported by multiple `explorer` programs, e.g.this is the case for the Aldebaran (`.aut`) format, and that then for the same language different details need attention, depending on the tool that is used to "access" it. That is why we have tool-specific manual pages and not language-specific ones.

2.1.1 What languages can we use?

Invoking `mkprimer(1)` without arguments makes it print a list of language names and corresponding file name suffixes. The list shows the languages for which support is available, which may vary from one installation to another, because for a number of languages TorX depends on external tools which may or may not have been installed. Figure 1 shows an example.

```
% mkprimer
usage: mkprimer [-o outputfile [ --language lang ] ]
                [ --inputs inputgates ] [ --outputs outputgates ]
                [ --config configfile ] specfile
usage: mkprimer --list
makes an executable program ‘‘outputfile’’ for specfile
outputfile defaults to "specname" (basename of "specfile" without suffix)
recognized languages and suffixes:
        AUT                     .aut
        AUT-CADP
        BCG                     .bcg
        CR                      .cr
        FC2                     .fc2
        FSP                     .lts
        LOTOS                   .lot .lotos
        PROMELA                 .trojka
        TTA                     .tta
```

Figure 1: `mkprimer` languages listing

2.2 get implementation

Get/write an implementation, and keep in mind that you will have to be able to (let TorX) interact with it, essentially by mimicking (the behaviour of) its user or users (that is essentially what TorX does: it uses the implementation as if it is a user of it, and it uses the specification to know what the user can do, and to know how the implementation should react). If the implementation runs somewhere in a box, and you have no means of connecting with the box, or with the program running in the box, you cannot do anything.

Connecting to and interacting with the implementation will be done by the `adapter`, which we will discuss further on.

2.3 make directory structure

Set up a directory structure. One way to do this is the structure we have chosen for the TorX examples. The general scheme we use is as depicted in figure 2.

```
experiment/
    makefile          (includes make.subdirs)
    experiment.if     (configuration file for torx)
    SPEC/
        makefile      (includes make.settings and make.specdir)
        LANG1/
           makefile   (includes make.settings and make.primers)
           explor.cfg (configuration file for primer)
           spec1.sfxl1
           spec1      (generated from spec1.sfxl1)
        LANG2/        (if we have specs in multiple languages)
           makefile   (includes make.settings and make.primers)
           spec1.sfxl2
           spec1      (generated from spec1.sfxl2)
    IUT/
       makefile       (includes make.settings and make.primers)
       IMPL           (compiled from files in source)
       SOURCE/
           various
           source
           files
    bin/
    lib/
       make/
           make.primers
           make.settings
           make.subdirs
```

Figure 2: Example directory structure

At the top level we have a configuration file (`experiment.if`[2]) and separate sub directories `SPEC` and `SUT` (or `IUT`), and for general utilities (`bin`, `lib`). If we use more than a single specification language, the `SPEC` directory contains separate subdirectories for each of them.

We use `make` and `makefile` to build (generate) the `primer` program, and, if necessary, to compile the implementation(s) (there may be multiple implementations – in the conference protocol example we compile a set "mutants" in single source directory).

In our example `LANG1` and `sfxl1` etc. are placeholders for language names like `LOTOS`, `PROMELA`, `AUT`, etc., respectively file name suffixes like `lot`, `trojka`,

[2]The `.if` suffix is a historical accident.

`aut`, etc. We tend to use uppercase for directory names, and lowercase for file names. This is partly historical, probably mostlly under influence of the convention we used in `mkprimer`, in which language names are in uppercase (see 'What languages can we use?' on page 5).

As said ablove, we use `make` to help us to maintain the dependencies between our source files and the programs (and other files) that are generated from them. In the TorX examples distribution we have tried to isolate common `make` "patterns" into separate makefiles (see TorX examples directory `lib/make`) that are included by the makefiles in our examples directory tree. These makefiles allow us to have specification subdirections for many languages, and when "make-ing" the primers, `make` will only consider those language subdirectories for which `mkprimer` supports the language. Additionally, the makefiles use a naming scheme for the generated executable programs that allows executables for multiple platforms to co-exist in a single directory. Study the makefiles in the TorX examples for details.

or should we explain them here?

2.4 generate primer program

Generate a `primer` program for/from the specification. The generic TorX program `mkprimer` does this. Details can be found in manual page `mkprimer(1)`. Specific, specification-language dependent details can be found in more detailed manual pages like `mkprimer-cadp(1)`, `mkprimer-ltsa(1)`, `mkprimer-trojka(1)`. [Note: probably `mkprimer-cadp(1)` is out of date regarding the Aldebaran (`.aut`) language, need to write `mkprimer-expaut(1)`, there is `autexp(1)` though.]

The `primer` program can be run from the command line, as a kind of single-step simulator, by typing the commands that the `torx` program uses to interface with it. The utility program `pui(1)` gives a slightly more comfortable way of using the `primer`. If the generated `primer` is called `confprot` `pui` is invoked as:

```
pui confprot -with -command -line options
```

The generated primers use a LOTOS-like notation for the actions (events, labels), where the various parts of an action are separated by exclamation marks (`!`). The first element will be the LOTOS gate or the Promela channel etc. The individual parts of an action may contain subexpressions between parentheses, as in:

```
cfsap_in ! some ! thing
cfsap_in ! join( user, chan )
cfsap_in ! join ! 52 ! 31
```

In general you will now also have to configure which are the stimuli and which are the observations. Unfortunately, how to specify this currentely depends on the specification language, because we generate different programs, depending on the specification language. The different programs have to be configured in different ways. There are three possibilities:

1. using Promela

2. using LOTOS or language for which we use CADP
3. other language (FSP, Aldebaran (`.aut`))

We discuss them one by one below.

2.4.1 Promela

In this case the generated `primer` consists of a single program into which the specification is compiled. In this case no special configuration is needed, because in the "dialect" of Promela that we use in TorX the information about which actions represent stimuli and which represent stimuli is already implied in the specification.

The main difference between a Promela model for model-checking, and a Promela specification for testing with TorX is that the model for model-checking is closed: it contains both the system and its environment. The specification for testing is open: it contains only the system, but not its environment. We make a closed model open by removing (not running) its environment, and by marking the channels that link the system and the environment as `observable`. All messages that the system sends over such channels are turned into observations (outputs from the system), and all messages that the system reads from such channels are turned into stimuli (inputs to the system). All messages send/received over the other channels are internal (invisible outside the system). The Promela conference protocol specification in the TorX examples distribution shows how this can be done (see the files in directory confprot/SPEC/PROMELA/). It is also explained in more detail in `mkprimer-trojka(1)`.

2.4.2 LOTOS or other language for which we use CADP

In this case the generated primer consists of a single program into which the specification is compiled. In this case we use the simple convention that each LOTOS "gate" is only used for communication in a single direction, so we have "input gates" (for stimuli) and "output gates" (for observations). So, in the LOTOS specification of the conference protocol in the TorX examples distribution we use `udp_in!udp0!message` to represent an udp message that the system receives from the environment on udp port `udp0`, and `udp_out!udp1!message` to represent an udp message that the system sends to the environment via udp port `udp1`.

We configure the input and output gates in the configuration file for `torx`, using configuration lines with keywords `INPUT` and `OUTPUT` respectively, e.g.:

```
INPUT udp_in
INPUT cfsap_in
OUTPUT udp_out
OUTPUT cfsap_out
```

This can be seen in the confprot/confprot.lotos.if file in the TorX examples directory. These keywords are documented in the torx-config(5) manual page.

`mkprimer` can be made to generate a partial configuration file for torx that contains `INPUT` and `OUTPUT` configuration lines by invoking it with `--inputs` and `--outputs` command line options to specify the input and output gates. This is documented in the `mkprimer(1)` manual page.

For example, when invoked as

```
mkprimer --inputs udp_in,cfsap_in --outputs udp_out,cfsap_out spec.lot
```

`mkprimer` will generate a primer program in output file `spec` and an additional file `spec.gates` containing the `INPUT` and `OUTPUT` lines shown above. This `spec.gates` file can then be included in the configuration file of `torx` using the `INCLUDE` keyword (as discussed below).

When `torx` starts such a `primer` program it gives it the input and output gates (separated by commas, no whitespace in between) using `-i` and `-o` command line options, like in:

```
confprot -i cfsap_in,udp_in -o cfsap_out,udp_out
```

When such a primer is invoked without these options it will warn that no inputs/outputs have been configured. To use `pui` with `confprot` with these options just add them:

```
pui  confprot -i cfsap_in,udp_in -o cfsap_out,udp_out
```

2.4.3 other language (FSP, Aldebaran (.aut))

In this case we use a generic (specification-language independent) primer program from the distribution (called `primer`). Access to the specification is realized via a specification-language dependent `explorer` program. In this case we give the `primer` program a configuration file in which we give patterns to match the input and output actions. This is a bit more "detailed" than the above case where we only use the gate names.

This configuration file contains sets of whitespace-separated `name=value` pairs. The sets themselves are separated by empty lines. Each set (block) specifies one "channel" (distinguishable means of interaction between tester and implementation) For each "channel" we define its name, its direction (here called `iokind`), and we give patterns to match the events (actions) that "belong" to this "channel". In addition we can specificy the name of the pseudo-action that we use to represent quiescence (the observation of the absence of output) as value of field `suspension`. The channel name can be arbitrarily chosen, but usually they will be `in` or `out`. The value of `iokind` must be either `input` or `output`. The value of `event` gives the pattern that is used to match actions. There may be multiple occurences of `event` for a channel.

In the definitions of the events, whitespace is ignored in the patterns when matching actions. Be careful: it may be best to specify actions (event patterns) in lowercase, the case sensitivity/insensitivity may be kind of broken (possible bug).

Each such set starts with a name-value pair at the beginning of a line. Name-value pairs belonging to the same set can be given on the same line, or on lines that start with whitespace.

Example:

```
channel=in
    iokind=input
    event=udp_in!*!*
    event=udp_in!*
    event=cfsap_in!*!*

channel=out
    iokind=output
    event=udp_out!*!*
    event=udp_out!*
    event=cfsap_out!*!*
    suspension=Delta
```

The first set above can also be written as:

```
channel=input event=udp_in!*!* event=udp_in!* event=cfsap_in!*!*
```

The patterns for event can contain wildcards. Be careful, the number of "¡' in the patterns must match the number of "!" in the actions in the specification. Note that for output actions (observations) we also explictly indicate that we can observe quiescence on them, using the suspension name=value pair.

The name of such a configuration file can be given with the "–config" command line option to the mkprimer program, such that it is incorporated in the generated program (actually: shell script).

TODO: add example.

2.5 write or find an adapter program

2.6 write the configuration file(s) for TorX

The main configuration file is the one for torx. This is a textfile. Each item appears on its own line, which starts with a keyword. Comment lines (starting with a "#") and empty lines are ignored. When relative path names are given in the configuration, they are interpreted relative to the directory in which the configuration file is. For historical reasons the filename should have a ".if" suffix. The details are in manual page torx-config(4). This manual page contains also a few examples.

For historical and practical reasons the configuration file for torx may contain also configuration information for other TorX components. In particular, it may contain information for the primer (in the INPUT and OUTPUT configuration lines), for xtorx (configuration entries related to the Primer and Mutants menus, and for showing specifications) and for the adapter (e.g. IUT).

The minimal configuration consists of the following items:

- the path to the primer program (to be started by torx)
- the path to the adapter program (to be started by torx)
- the input and output actions
- (optionally) the path to the implementation program (if it has to be started by torx)
- a PCO declaration (even if not used, it has to be there or else torx will complain)

The path to the primer program has to be given as follows.

```
SPEC path/to/primer
```

The path to the adapter program has to be given as follows[3]

```
ADAPTOR path/to/adapter
```

By default, torx starts the primer and the adapter such that their "current working directory" is the directory that contains them, i.e. with the example configuration given above, the will be running in directory

```
path/to/
```

relative to the directory containing the configuration file.

Knowing this may avoid some surprises (even though in general it should not matter much, there may be cases where it does). This behaviour can be overruled by configuring items SPECRUNDIR and ADAPTORRUNDIR. There are other programs that are started by torx in the same way. For details see manual page torx-config(4) and look for "RUNDIR".

We already discussed above (when talking about "making a primer") how to configure which are the input actions and which are the output actions.

As discussed there this configuration either has to be in the torx configuration file, or in a separate file for the primer, depending on the explorer/primer that has to be used (which is a consequence of the specification language chosen). This distinction is unfortunate, and we hope to make it disappear in some future version of TorX, but for now we just have to live with it.

If it has to be part of the configuration file of torx, it has to be configured as follows.

```
INPUT udp_in
INPUT cfsap_in
OUTPUT udp_out
OUTPUT cfsap_out
```

[3] In the past there has been some confusion between "adapter" and "adaptor"; nowadays we usually speak about adapters, but the keyword still has the "o".

As discussed above, "mkprimer" can generate a file containing this if it is invoked with the –inputs and –outputs command line options. If that has been done, the file generated by mkprimer can be included as follows.

```
INCLUDE path/to/primer.gates
```

2.7 run the experiment (using torx or xtorx)

The easiest way to run the experiment, once all ingredients are there, is to start xtorx, load the configuration file, choose a file for the logs, and press the start button in xtorx. This will start torx. If all works fine, debugging output should appear in the bottom pane of xtorx, and in the "Inputs" and "Outputs" panes the actions currently possible should appear.

If things do not work, maybe the reason can be found in the messages in the bottom pane of xtorx. If it looks like nothing happens, it may be useful to start torx directly (and not via xtorx):

```
torx config.if
```

Details about the use of xtorx(1) and torx(1) can be found in the manual pages.

include more elaborate description from elsewhere (testing technigues handout)?

2.8 (optionally) run the experiment multiple times using torx in batch mode

The torx(1) manual page contains an example.

to be added

2.9 analyze results

The results appear in the log file – if one has been configured in the torx configuration file, and this has not been overruled in xtorx. The log file should contain all we need to know/save about a test run for later analysis.

Details about the format of this file can be found in torx-log(4). It is a text file.

The log file starts with the configuration of torx. (In the future we would like to have the configuration of the other components appear there too). This is followed by some lines of text for each test step. Each of these lines starts with a keyword followed by the test step number.

Some of the possible details will only be there if the primer or adapter provided it.

Not all primers (can) provide the same information. Details should be in the corresponding mkprimer(1) subpage or explorer page (like mkprimer-cadp(1) or autexp(1)).

Information about what happens "in the real world" is only present if the adapter tells torx about it. Examples of this are the concrete messages exchanged by the adapter with the implementation under test, or the PCOs. In general these are the elements that are optional in the torx-adapter(5) interface (look for example for the A_INPUT_OK and A_OUTPUT_OK messages).

How to get a "primer" program? "mkprimer" takes the name of the specification as argument. It may take other arguments, in particular ones that say which actions (events, labels) in the specification are to be used as stimuli (i.e. inputs for the implementation under test) and which ones are to be used as observations (i.e. outputs of the implementation under test). Unfortunately, the way in which it expects this depends on the

More details about mkprimer can be found in mkprimer(1). More details about the commands that torx sends to a primer, and the answers that it expects can be found in torx-primer(5).

Programs that typically are implementation-under-test-specific are what TorX calls "adapters" – programs that TorX uses to access the implementation under test. Adapter programs have on generice interface, which is used by TorX to "talk" to them, and one or more implementation-under-test specific interfaces, which it uses to interact with the implementation. Over the interface to torX it uses the actions (events, labels) from the specification; over the interfaces to the implementation it uses whatever the implementation expects.

References

[dVBF02] René G. de Vries, Axel Belinfante, and Jan Feenstra. Automated Testing in Practice: The Highway Tolling System. In Ina Schieferdecker, Hartmut König, and Adam Wolisz, editors, *Testing of Communicating System XIV*, pages 219–234, Berlin, 2002. Kluwer academic publishers. 2 [4]