# NovaProva Documentation

*Release latest*

Gregory Banks

May 19, 2020

# Contents

# CHAPTER 1

# Contents

## 1.1 Introduction

### 1.1.1 What is NovaProva?

NovaProva is a new generation unit test framework for C and C++ programs.

NovaProva starts with the well-known xUnit paradigm for testing, but extends some concepts in new directions. For example, NovaProva organises tests into trees rather than xUnit's two-level "suite and test" approach.

Most importantly, NovaProva is designed using real world experience and modern technologies to make it much easier to write tests, and much easier to discover subtle bugs in the System Under Test, bugs that unit other test frameworks miss. NovaProva's mission is to make testing C and C++ code as easy as possible.

### 1.1.2 Why Create NovaProva?

At the time NovaProva was first written, the author had a day job working on a large C (K&R in some places) codebase which was in parts decades old. This code had *zero* buildable tests. The author created hundreds of tests, both system tests using a new application-specific Perl framework and C regression tests using the venerable CUnit library.

This experience showed that writing more than a handful of tests using CUnit is very hard, and gets harder the more tests are written and the more insightful the tests. NovaProva is designed to make the process of writing and running C and C++ unit tests as easy as possible.

### 1.1.3 Design Philosophy

NovaProva's design philosophy is based on the following principles.

- Fully support C as well as C++. This means that NovaProva does not rely on C++ compile time features for test discovery or reflection. Test code must be buildable with only a C compiler, without C++.
- Choose power over portability. Portability is important, but for tests it's better to have good tests on one platform than bad tests on all platforms. Bad tests are a waste of everyone's time.

- Simplify the tedious parts of testing. Do as much as possible in the framework so that users don't need to.
- Choose correctness and test power over test run speed. Slow tests are annoying but remotely debugging buggy code which shipped is much more annoying and expensive.
- Choose best practice as the default. For example, maximise test isolation, and maximise detection of failure modes.
- Have the courage to do things that are "impossible" in C & C++, such as dynamic mocking.

## 1.2 Getting Started

### 1.2.1 Installation

First you need to download and install NovaProva. Here are several ways you can do this, starting with the easiest.

#### From The OpenSUSE Build Service

NovaProva is available in installable repositories for many recent Linux distributions at the OpenSUSE Build Service.

For Debian/Ubuntu systems, copy and paste the following commands.

```
# Choose a distro
distro=xUbuntu_12.04

# This is the base URL for the NovaProva repo
repo=http://download.opensuse.org/repositories/home:/gnb:/novaprova/$distro

# Download and install the repository GPG key
wget -O - $repo/Release.key | sudo apt-key add -

# Add an APT source
sudo bash -c "echo 'deb $repo ./' > /etc/apt/sources.list.d/novaprova.list"
sudo apt-get update

# Install NovaProva
sudo apt-get install novaprova
```

See here for RedHat and SUSE instructions.

#### From A Release Tarball

First, download a release tarball.

NovaProva is intended to be built in the usual way that any open source C software is built. It has a *configure* script generated by autoconf, which you run before building. To build you need to have various pieces of software installed, starting with a typical C development environment and adding the Valgrind header file *valgrind.h* and the XML and BFD libraries. Here are some example commands which download and install them.

```
# on Ubuntu
sudo apt-get install -y \
    gcc g++ git autoconf automake libxml2-dev libxml2-utils \
    pkg-config valgrind binutils-dev zlib1g-dev libiberty-dev

```

(continues on next page)

(continued from previous page)

```
# on RHEL / Fedora
sudo yum install -y \
   gcc gcc-c++ autoconf automake libxml2-devel pkgconfig \
   valgrind valgrind-devel binutils-devel zlib-devel
```

Once you have those prerequisites installed, you can download, install and build NovaProva.

```
# download the release tarball from https://github.com/novaprova/novaprova/releases

tar -xvf novaprova-1.1.tar.bz2
cd novaprova-1.1
./configure
make
make install
```

### From Read-Only Git

For advanced users only. NovaProva needs several more tools to build from a Git checkout than from a release tarball, mainly for the documentation. You will need to have Doxygen, XML::LibXML, Sphinx, and Breathe installed. Here are some example commands which download and install them.

```
# on Ubuntu
# install all the prereqs above, then add...
sudo apt-get install -y doxygen libxml-libxml-perl python-pip
sudo pip install breathe Sphinx

# on RHEL / Fedora
# install all the prereqs above, then add...
sudo yum install -y doxygen perl-XML-LibXML \
    python-breathe python-sphinx
```

Once you have those prerequisites installed, you can clone, install and build NovaProva.

```
git clone https://github.com/novaprova/novaprova.git
cd novaprova
autoreconf -iv
./configure
make
make install
```

## 1.2.2 Building a Test Executable

Because you're testing C code, the first step is to build a test runner executable. This executable will contain all your tests and the Code Under Test and will be linked against the NovaProva library and whatever other libraries your Code Under Test needs. Typically, this is done using the *check:* make target to both build and run the tests.

Start by creating a Makefile containing:

```
# Makefile
all: libmycode.a

MYCODE_SOURCE=      mycode.c
MYCODE_OBJS=        $(MYCODE_SOURCE:.c=.o)
```

(continues on next page)

(continued from previous page)

```

libmycode.a: $(MYCODE_OBJS)
        ar ruv $@ $(MYCODE_OBJS)
        ranlib $@

NOVAPROVA_CFLAGS= $(shell pkg-config --cflags novaprova)
NOVAPROVA_LIBS= $(shell pkg-config --libs novaprova)

CFLAGS= -g $(NOVAPROVA_CFLAGS)

check: testrunner
        ./testrunner

TEST_SOURCE= mytest.c
TEST_OBJS=  $(TEST_SOURCE:.c=.o)

testrunner:  $(TEST_OBJS) libmycode.a
        $(LINK.c) -o $@ $(TEST_OBJS) libmycode.a $(NOVAPROVA_LIBS)

clean:
        $(RM) testrunner libmycode.a $(TEST_OBJS) $(MYCODE_OBJS)
```

NovaProva uses the GNOME *pkgconfig* system to make it easy to find the right set of compile and link flags.

Note that you only need to compile the test code *mytest.c* with *NOVAPROVA_CFLAGS*. NovaProva does *not* use any magical compile options or do any pre-processing of your test code or Code Under Test.

However, you should make sure that at least the test code is built with the *-g* option to include debugging information. NovaProva uses that information to discover tests at runtime.

You do not need to provide a *main* routine. NovaProva provides a default *main* routine which implements a number of useful command-line options. You can write your own later, but you probably won't need to.

Now let's create an example Code Under Test. It contains the function *myatoi* which has the same signature and semantics as the well-known *atoi* function in the standard C library. We have a header file:

```
/* mycode.h */
#ifndef __mycode_h_
#define __mycode_h_ 1

extern int myatoi(const char *);

#endif /* __mycode_h_ */
```

and a source file:

```
/* mycode.c */
#include "mycode.h"

int myatoi(const char *s)
{
    int v = 0;

    for ( ; *s ; s++)
    {
        v *= 10;
        v += (*s - '0');
    }
```

(continues on next page)

(continued from previous page)

```
    return v;
}
```

The last piece of the puzzle is writing some tests. Each test is a single C function which takes no parameters and returns *void*. Unlike other unit test frameworks, there's no API to call or magical macro to use to register tests with the library. Instead you just name the function *test_something*, and NovaProva will automatically create a test called *something* which calls the function.

For example, let's create a test called *simple* which exercises the most basic functionality of *myatoi()*.

```
/* mytest.c */
#include <np.h>          /* NovaProva library */
#include "mycode.h" /* declares the Code Under Test */

static void test_simple(void)
{
    int r;

    r = myatoi("42");
    NP_ASSERT_EQUAL(r, 42);
}
```

The macro *NP_ASSERT_EQUAL* checks that it's two integer arguments are equal, and if not fails the test. Note that if the assert fails, the test function terminates immediately. If the test function gets to it's end and returns naturally, the test is considered to have passed.

If we build run this test we get output something like this.

```
% make check
./testrunner
np: starting valgrind
np: running
np: running: "mytest.simple"
PASS mytest.simple
np: 1 run 0 failed
```

As expected, the test passed.

NovaProva organises tests into a tree whose node names are derived from the test source directory, test source filename, and test function name. This tree is pruned down to the smallest possible size at which the root of the tree is unique. So the name *mytest.simple* derives from the name of the function *test_simple* in source file *mytest.c*.

Now let's add another test. The *myatoi()* function is supposed to convert the initial numeric part of the argument string, i.e. to stop when it sees a non-numeric character. Let's feed it a string which will exercise this behaviour and see what happens.

```
/* add this to the end of mytest.c */

static void test_initial(void)
{
    int r;

    r = myatoi("4=2");
    NP_ASSERT_EQUAL(r, 4);
}
```

Running the tests we see:

```
% make check
./testrunner
np: starting valgrind
np: running
np: running: "mytest.simple"
PASS mytest.simple
np: running: "mytest.initial"
EVENT ASSERT NP_ASSERT_EQUAL(r=532, 4=4)
at 0x80529F2: np::spiegel::describe_stacktrace (np/spiegel/spiegel.cxx)
by 0x804C0FC: np::event_t::with_stack (np/event.cxx)
by 0x804B2D2: __np_assert_failed (uasserts.c)
by 0x804AC27: test_initial (mytest.c)
by 0x80522D0: np::spiegel::function_t::invoke (np/spiegel/spiegel.cxx)
by 0x804C731: np::runner_t::run_function (np/runner.cxx)
by 0x804D5C4: np::runner_t::run_test_code (np/runner.cxx)
by 0x804D831: np::runner_t::begin_job (np/runner.cxx)
by 0x804E0D4: np::runner_t::run_tests (np/runner.cxx)
by 0x804E22C: np_run_tests (np/runner.cxx)
by 0x804AB12: main (main.c)

FAIL mytest.initial
np: 2 run 1 failed
make: *** [check] Error 1
```

Note also that the new test failed. Immediately after the “np: running:” message we see that the *NP_ASSERT_EQUAL* macro has failed, and printed both its arguments as well as a stack trace. We expected the variable *r* to equal to 4 but its actual value at runtime was 532; clearly the *myatoi* function did not behave correctly. We found a bug!

And now you’re testing with NovaProva. The remainder of this document contains everything you need to know to get the best out of NovaProva. Best of luck and good testing!

## 1.3 Platform Support

NovaProva is available on the following platforms.

- Linux x86
- Linux x86_64
- Coming soon: Darwin x86_64

NovaProva supports the GNU compiler up to gcc 4.8, including support for the DWARF-4 debugging standard.

You may note that this is not a lot of platforms. The reason is that the design of NovaProva requires it to deeply dependent on unobvious and undocumented aspects of the operating system it runs on, particularly the kernel, the dynamic linker, and the C runtime library. As such it is unlikely that it could be successfully ported to an OS for which source was not available. Also, NovaProva achieves its greatest value when paired with Valgrind, which itself is even more dependent on the underlying OS. For more information on porting NovaProva to other platforms, see *Porting NovaProva*.

## 1.4 Installing NovaProva

### 1.4.1 From The Open Build Service

NovaProva is available in installable package repositories for many recent Linux distributions at the OpenSUSE Build Service Visit that page and follow the instructions there.

Alternatively, for Debian/Ubuntu systems, copy and paste the following commands.

```
# Choose a distro
distro=xUbuntu_12.04

# This is the base URL for the NovaProva repo
repo=http://download.opensuse.org/repositories/home:/gnb:/novaprova/$distro

# Download and install the repository GPG key
wget -O - $repo/Release.key | sudo apt-key add -

# Add an APT source
sudo bash -c "echo 'deb $repo ./' > /etc/apt/sources.list.d/novaprova.list"
sudo apt-get update

# Install NovaProva
sudo apt-get install novaprova
```

See here for commandline instructions for other Linux distributions.

### 1.4.2 From A Release Tarball

First, download a release tarball from Sourceforge.

NovaProva is intended to be built in the usual way that any open source C software is built. It has a configure script generated by autoconf, which you run before building. To build you need to have g++ and gcc installed, as well as the following:

- Valgrind (Ubuntu `apt-get install valgrind`, RedHat `yum install valgrind`)
- autoconf (Ubuntu `apt-get install autoconf`, (RedHat `yum install autoconf`)
- libxml2 (Ubuntu `apt-get install libxml2-dev`, RedHat `yum install libxml2-devel`)
- pkg-config (Ubuntu `apt-get install pkg-config`, RedHat `yum install pkgconfig`)
- The BFD library (Ubuntu `apt-get install binutils-dev`, RedHat `yum install binutils-devel`)

After installing all that, run these commands.

```
tar -xvf novaprova-1.1.tar.bz2
cd novaprova-1.1

./configure

make
make install
```

### 1.4.3 From Read-Only Git

For advanced users only. NovaProva needs several more tools to build from a Git checkout than from a release tarball, mainly for the documentation. To build you need to have g++ and gcc installed, as well as the following:

- Doxygen (Ubuntu `apt-get install doxygen`, RedHat `yum install doxygen`)
- Valgrind (Ubuntu `apt-get install valgrind`, RedHat `yum install valgrind`)
- Breathe (Ubuntu `apt-get install breathe-doc`, or `pip install breathe`)
- autoconf (Ubuntu `apt-get install autoconf`, (RedHat `yum install autoconf`)
- libxml2 (Ubuntu `apt-get install libxml2-dev`, RedHat `yum install libxml2-devel`)
- pkg-config (Ubuntu `apt-get install pkg-config`, RedHat `yum install pkgconfig`)
- The BFD library (Ubuntu `apt-get install binutils-dev`, RedHat `yum install binutils-devel`)

After installing all that, run these commands.

```
git clone git://github.com/gnb/novaprova.git
cd novaprova

# ignore the errors, this is only needed to copy
# in install-sh which autoconf wants
autoreconf -iv

./configure

make
make install
```

## 1.5 Building a Test Executable

### 1.5.1 What is a Test Executable?

Because you're testing C code, the first step is to build a test runner executable. This executable will contain all your tests and the Code Under Test and will be linked against the NovaProva library and whatever other libraries your Code Under Test needs. When you want to run all or one of your tests, you run this executable with various arguments.

### 1.5.2 Setting Up the Makefile

Most C and C++ software is built using the venerable `make` utility, these days usually the GNU make implementation. While you can use any names you like, the GNU project defines a standard set of target names and their semantics, which you would be advised to stick to. The target `check:` is what you should be using to both build and run the tests.

Here is a fragment of an example Makefile. It assumes your code to be tested has been built into a local archive library `libmycode.a`, but it also works if you replace that with an explicit list of separate object files.

```
NOVAPROVA_CFLAGS= $(shell pkg-config --cflags novaprova)
NOVAPROVA_LIBS= $(shell pkg-config --libs novaprova)

CFLAGS= ... -g $(NOVAPROVA_CFLAGS) ...
```

(continues on next page)

(continued from previous page)

```
check: testrunner
        ./testrunner

TEST_SOURCE= mytest.c
TEST_OBJS=  $(TEST_SOURCE:.c=.o)

testrunner:  $(TEST_OBJS) libmycode.a
        $(LINK.c) -o $@ $(TEST_OBJS) libmycode.a $(NOVAPROVA_LIBS)
```

NovaProva uses the GNOME `pkgconfig` system to make it easy to find the right set of compile and link flags.

Note that you only need to compile the test code `mytest.c` with `NOVAPROVA_CFLAGS`, and link with the NovaProva library. NovaProva does *not* use any magical compile options or do any pre-processing of test code or the Code Under Test. All the magic happens at runtime.

However, you should make sure that at least the test code is built with the `-g` option to include debugging information. NovaProva uses that information to discover tests.

### 1.5.3 Using GNU Automake

Many C developers prefer to use GNU automake to build their projects. One good reason is that it's by far the easiest technique to build shared libraries in a cross-platform manner. NovaProva can be used to run tests in an automake-based project too.

First, ensure that your `configure.ac` has the following

```
dnl configure.ac

AC_INIT(15_automake, 1.4)
AM_INIT_AUTOMAKE([serial-tests])
AC_PROG_CC
AC_PROG_RANLIB

PKG_CHECK_MODULES(NOVAPROVA, novaprova)

AC_CONFIG_HEADERS([config.h])
AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

Note the use of the `serial-tests` automake option. Recent versions of automake feature a new "parallel tests" feature, which is enabled by default. This feature is complicated to use and provides no benefit at all when used with NovaProva, as NovaProva implements it's own parallelism and doesn't need help from automake. The `serial-tests` option disables the feature. If you're running with an older version of automake which does not offer the parallel test feature, you do not need to specify the option.

Note also the use of the `PKG_CHECK_MODULES` autoconf macro. This sets up the variables `NOVAPROVA_CFLAGS` and `NOVAPROVA_LIBS` to the correct value for compiling and linking with the NovaProva library.

Next, ensure your Makefile.am contains something like the following.

```
# Makefile.am

lib_LIBRARIES=      libmycode.a
libmycode_a_SOURCES=        mycode.c
```

(continues on next page)

(continued from previous page)

```
# Tell automake to build the testrunner on "make check"
check_PROGRAMS=     testrunner
# Tell automake to run the testrunner on "make check"
TESTS= $(check_PROGRAMS)

# List all your test source code here
testrunner_SOURCES= mytest.c
# Compile only test code with -g and the NovaProva flags
testrunner_CFLAGS= -g $(NOVAPROVA_CFLAGS)
# Link testrunner with the Code Under Test and the NovaProva library
testrunner_LDADD= libmycode.a $(NOVAPROVA_LIBS)
```

Now when you run `make check`, make will build the Code Under Test, build the test code, link the test runner, and run the test runner.

## 1.5.4 Main Routine

You do not need to provide a `main()` routine for the test executable to link. The NovaProva library provides a default `main()` routine which implements a number of useful command-line options. This section describes the behavior of test executables built with the default `main()`.

Note, you can always write your own `main()` later, but you probably won't need to. NovaProva has a hierarchical *Fixtures* feature which you should probably use instead.

## 1.5.5 Test Executable Usage

Here is a description of the test executable usage.

**./testrunner –list**
**./testrunner** [**-j** *number*] [**-f** *format*] [*test_spec*. . . ]

**-f** ***format*****, –format** ***format*** Set the format in which test results will be emitted. See *Output Formats* for a list of available formats.

**-j** ***number*****, –jobs** ***number*** Set the maximum number of test jobs which will be run at the same time, to *number*. The default value is 1, meaning tests will be run serially. A value of 0 is shorthand for one job per online CPU in the system, which is likely to be the most efficient use of the system.

**-l, –list** Instead of running any tests, print to stdout the fully qualified names of all the test functions (i.e. leaf test nodes) known to NovaProva, and exit.

***test_spec*** The fully qualified name of a test node (i.e. a test, a test source file file, or a directory containing test source files). All the tests at or below the test node will be run. Tests are started in test node traversal order. If no tests are specified, all the tests known to NovaProva will be run.

## 1.6 Writing Test Functions

### 1.6.1 Runtime Discovery

Test functions are discovered at runtime using Reflection. The NovaProva library walks through all the functions linked into the test executable and matches those which take no arguments, return `void`, and have a name matching one of the following patterns:

- `test_foo`
- `testFoo`
- `TestFoo`

Here's an example of a test function.

```
#include <np.h>

static void test_simple(void)
{
    int r = myatoi("42");
    NP_ASSERT_EQUAL(r, 42);
}
```

Note that you do not need to write any code to register this test function with the framework. If it matches the above criteria, the function will be found and recorded by NovaProva. Just write the function and you're done.

### 1.6.2 The Test Tree

Most other test frameworks provide a simple, 2-level mechanism for organizing tests; *tests* are grouped into *suites*.

By contrast NovaProva organizes tests into an **tree of test nodes**. All the tests built into a test executable are gathered at runtime and are fitted into a tree, with a single common root. The root is then pruned until the test names are as short as possible. Each test function is a leaf node in this tree (usually).

The locations of tests in this tree are derived from the names of the test function, the basename of the test source file containing the test function, and the hierarchy of filesystem directories containing that source file. These form a natural classifying scheme that you are already controlling by choosing the names of filenames and functions. These names are stuck together in order from least to most specific, separated by ASCII '.' characters, and in general look like this.

```
dir.subdir.more.subdirs.filename.function
```

Here's an example showing how test node names fall naturally out of your test code organization.

```
% cat tests/startrek/tng/federation/enterprise.c
static void test_torpedoes(void)
{
    fprintf(stderr, "Testing photon torpedoes\n");
}

% cat tests/startrek/tng/klingons/neghvar.c
static void test_disruptors(void)
{
    fprintf(stderr, "Testing disruptors\n");
}
```

(continues on next page)

(continued from previous page)

```
% cat tests/starwars/episode4/rebels/xwing.c
static void test_lasers(void)
{
    fprintf(stderr, "Testing laser cannon\n");
}

% ./testrunner --list
tests.startrek.tng.federation.enterprise.torpedoes
tests.startrek.tng.klingons.neghvar.disruptors
tests.starwars.episode4.rebels.xwing.lasers
```

### 1.6.3 Pass and Fail

A test passes in a very simple way: it returns without failing. A test can fail in any number of ways, some of them obvious, all of them indicative of a bug in the Code Under Test (or possibly the test itself). See *Assert Macros* and *Failure Modes* for full details.

Here's an example of a test which always passes.

```c
static void test_always_passes(void)
{
    printf("Hi, I'm passing!\n");
}
```

A test can also use the `NP_PASS` macro, which terminates the test immediately without recording a failure.

```c
static void test_also_always_passes(void)
{
    printf("Hi, I'm passing too!\n");
    NP_PASS;                                    /* terminates the test */
    printf("Now I'm celebrating passing!\n");   /* never happens */
}
```

Note that this does not necessarily mean the test will get a Pass result, only that the test itself thinks it has passed. It is possible that NovaProva will detect more subtle failures that the test itself does not see; some of these failures are not even detectable until after the test terminates. So, `NP_PASS` is really just a complicated `return` statement and you should probably never use it.

```c
static void test_thinks_it_passes(void)
{
    void *x = malloc(24);
    printf("Hi, I think I'm passing!\n");
    NP_PASS;        /* but it's wrong, it leaked memory */
}
```

A test can use the `NP_FAIL` macro, which terminates the test and records a Fail result. Unlike `NP_PASS`, if a test says it fails then NovaProva believes it.

```c
static void test_always_fails(void)
{
    printf("Hi, I'm failing\n");
    NP_FAIL;                                    /* terminates the test */
    printf("Now I'm mourning my failure!\n");   /* never happens */
}
```

Note that NovaProva provides a number of declarative *Assert Macros* which are much more useful than using `NP_FAIL` inside a conditional statement. Not only are they more concise, but if they cause a test failure they provide a more useful error message which helps with diagnosis. For example, this test code

```
static void test_dont_do_it_this_way(void)
{
    if (atoi("42") != 3)
        NP_FAIL;
}

static void test_do_it_this_way_instead(void)
{
    NP_ASSERT_EQUAL(atoi("42"), 3);
}
```

Will generate the following error messages

```
% ./testrunner

np: running: "mytests.dont_do_it_this_way"
EVENT EXFAIL NP_FAIL called
FAIL mytests.dont_do_it_this_way

np: running: "mytests.do_it_this_way_instead"
EVENT ASSERT NP_ASSERT_NOT_EQUAL(atoi("42")=42, 3=3)
FAIL mytests.do_it_this_way_instead
```

NovaProva also supports a third test result, Not Applicable, which is neither a Pass nor a Fail. A test which runs but decides that some preconditions are not met, can call the `NP_NOTAPPLICABLE` macro. Such tests are not counted as either passes or failures; it's as if they never existed.

### 1.6.4 Dependencies

Some unit test frameworks support a concept of test dependencies, i.e. the framework knows that some tests should not be run until after some other tests have been run. NovaProva does not support test dependencies.

In the opinion of the author, test dependencies are a terrible idea. They encourage a style of test writing where some tests are used to generate external state (e.g. rows in a database) which is then used as input to other tests. NovaProva is designed around a model where each test is isolated, repeatable, and stateless. This means that each test must trigger the same behaviour in the Code Under Test and give the same result, regardless of which order tests were run, or whether they were run in parallel, or whether any other tests were run at all, or whether the test had been run before.

The philosophy here is that the purpose of tests is to find bugs and to keep on finding bugs long after it's written. If a test is run nightly, fails roughly once a month, but nobody can figure out why, that test is useless. So a good test is conceptually simple, easy to run, and easy to diagnose when it fails. Deliberately sharing state between tests makes it harder to achieve all these ideals.

If you find yourself writing a test and you want to save some time by feeding the results of one test into another, please just stop and think about what you're doing.

If the Code Under Test needs to be in a particular state before the test can begin, you should consider it to be the job of the test to achieve that state from an initial null state. You can use *Fixtures* to pull out common code which sets up such state so that you don't have to repeat it in every test. You can also use coding techniques which allow to save and restore the state of the Code Under Test (e.g. a database dump), and check the saved state into version control along with your test code.

## 1.7 Failure Modes

This chapter lists the various failure modes that NovaProva automatically detects in Code Under Test, in addition to the explicit assert macro calls that you write in test functions, and describes how they are reported.

### 1.7.1 Isolate, Detect, Report

The purpose of a test framework is to discover bugs in code at the earliest possible time, with the least amount of work by developers. To achieve this, NovaProva takes an "Isolate, Detect, Report" approach.

- *Isolate*: failure modes are isolated to the test that caused them, and thus do not affect other tests. This helps mitigate the problem of cascading spurious test failures, which it harder for you to track down which test failures are directly due to bugs in the Code Under Test.
- *Detect*: failures are detected using the best automated debugging techniques possible. This reduces the time it takes you to find subtle bugs.
- *Report*: failures are reported with as much information as possible, in the normal test report. Ideally, many bugs can be diagnosed by examining the test report without re-running the test in a debugger. This reduces the time it takes you to diagnose test failures.

#### Process Per Test

NovaProva uses a strong model of test isolation. Each test is run in a separate process, with a central process co-ordinating the starting of tests and the gathering of results. This design eliminates a number of subtle failure modes where running one test can influence another, such as heap corruption, file descriptor leakage, and global variable leakage. The process-per-test model also has the advantage that tests can be run in parallel, and it allows for test timeouts to be handled reliably.

#### Valgrind

All tests are run using the Valgrind memory debugging tool, which enables detection of a great many subtle runtime errors not otherwise detectable.

The use of Valgrind is on by default and is handled silently by the NovaProva library. Normally, running a program under Valgrind requires the use of a wrapper script or special care in the Makefile, but with NovaProva all you have to do is to run the test executable.

NovaProva also detects when the test executable is being run under a debugger such as `gdb`, and avoids using Valgrind. This is because `gdb` and Valgrind usually interact poorly, and it's best to use only one or the other.

The use of Valgrind can also be manually disabled by using the `NOVAPROVA_VALGRIND` environment variable before running the test executable. This is not a recommended practice.

```
# NOT RECOMMENDED
export NOVAPROVA_VALGRIND=no
./testrunner
```

The downside of all this isolation and debugging is that tests can run quite slowly.

#### Stack Traces

NovaProva reports as much information as possible about each failure. In many cases this includes a stack trace showing the precise point at which the failure was detected. When the failure is detected by Valgrind, often even more information is provided. For example, when Valgrind detects that the Code Under Test written some bytes past the end of a struct allocated with `malloc()`, it will tell also give you the stack trace showing where that struct was allocated.

### 1.7.2 Call To exit()

NovaProva assumes that the Code Under Test is library code, and that therefore any call to `exit()` is inappropriate. Thus, any call to the libc `exit()` while running a test will cause the test to fail and print the exit code and a stack trace. Note that calls to the underlying `_exit()` system call are *not* detected.

Here's some example test output.

```
np: running: "mytest.exit"
About to call exit(37)
EVENT EXIT exit(37)
at 0x8056522: np::spiegel::describe_stacktrace
by 0x804BD9C: np::event_t::with_stack
by 0x804B0CE: exit
by 0x804AD42: test_exit
by 0x80561F0: np::spiegel::function_t::invoke
by 0x804C3A5: np::runner_t::run_function
by 0x804D28A: np::runner_t::run_test_code
by 0x804D4F7: np::runner_t::begin_job
by 0x804DD9A: np::runner_t::run_tests
by 0x804DEF2: np_run_tests
by 0x804ACF2: main
FAIL mytest.exit
np: 1 run 1 failed
```

### 1.7.3 Messages Emitted To syslog()

NovaProva assumes by default that messages emitted using the libc `syslog()` facility are error reports. Thus any call to `syslog()` while a test is running will cause the test to fail immediately, and the message and a stack trace will be printed.

Here's some example output.

```
np: running: "mytest.unexpected_syslog"
EVENT SLMATCH err: This message shouldn't happen
at 0x8059B22: np::spiegel::describe_stacktrace
by 0x804DD9C: np::event_t::with_stack
by 0x804B9FC: np::mock_syslog
by 0x807A519: np::spiegel::platform::intercept_tramp
by 0x804B009: test_unexpected_syslog
by 0x80597F0: np::spiegel::function_t::invoke
by 0x804FB99: np::runner_t::run_function
by 0x8050A7E: np::runner_t::run_test_code
by 0x8050CEB: np::runner_t::begin_job
by 0x805158E: np::runner_t::run_tests
by 0x80516E6: np_run_tests
by 0x804AFC3: main
FAIL mytest.unexpected_syslog
```

Sometimes the Code Under Test is actually expected to emit messages to `syslog()`. In these cases you can tell NovaProva to ignore the message and keep executing the test, using the `np_syslog_ignore()` call. This function takes a UNIX extended regular expression as an argument; any message which is emitted to `syslog()` from that point onwards in the test that matches the regular expression, will be silently ignored and will not cause the test to fail.

You can make multiple calls to `np_syslog_ignore()`, they accumulate until the end of the test. There's no need to remove these regular expressions, they're automatically removed at the end of the test.

Here's an example.

```
static void test_expected(void)
{
    /* tell NP that a syslog might happen */
    np_syslog_ignore("entirely expected");
    syslog(LOG_ERR, "This message was entirely expected");
}
```

When run, this test produces the following output. Note that the test passes and the message does not appear.

```
np: running: "mytest.expected"
PASS mytest.expected
```

This behavior changed in version 1.4 (commit 6234a2a). Before that, the call to `syslog()` would result in an `EVENT` and a stacktrace rather than being silently ignored.

You can achieve more subtle effects than just ignoring messages with `np_syslog_ignore()` by using it in combination with `np_syslog_fail()`. The latter function also takes a regular expression which is matched against messages emitted to `syslog()`, but it restores the default behavior where a match causes the test to fail. Sometimes this can be easier to do than trying to construct complicated regular expressions.

Finally, if the test depends on the Code Under Test generating (or not generating) specific messages, you can use `np_syslog_match()` which tells NovaProva to just count any matching messages, and `np_syslog_count()` to discover that count and assert on its value. The behavior of `np_syslog_match()` changed in version 1.4 (commit ef2f3b4). Before that, the call to `syslog()` would result in an `EVENT` and a stacktrace rather than being silently counted.

You can of course call any of `np_syslog_ignore()`, `np_syslog_fail()` and `np_syslog_match()` in a setup function (see *Fixtures* ).

### 1.7.4 Failed Calls To libc assert()

The standard library's `assert()` macro is sometimes used in the Code Under Test to check for conditions which must be true or the program is fatally flawed, e.g. preconditions, or the internal consistency of data structures. If the condition is false, the macro prints a message and exits the running process by calling `abort()`. NovaProva catches this occurrence, prints a more useful error message than the default (including a stack trace), and gracefully fails the test.

Here's an example.

```
static void test_assert(void)
{
    int white = 1;
    int black = 0;
    assert(white == black);
}
```

When run, this test produces the following output and the test fails.

```
np: running: "tnassert.assert"
EVENT ASSERT white == black
at 0x41827F: np::spiegel::describe_stacktrace (np/spiegel/spiegel.cxx)
by 0x40555C: np::event_t::with_stack (np/event.cxx)
```

(continues on next page)

(continued from previous page)

```
by 0x404CCD: __assert_fail (iassert.c)
by 0x4049F2: test_assert (tnassert.c)
by 0x417E0B: np::spiegel::function_t::invoke (np/spiegel/spiegel.cxx)
by 0x409E04: np::runner_t::run_function (np/runner.cxx)
by 0x40A83D: np::runner_t::run_test_code (np/runner.cxx)
by 0x40AB06: np::runner_t::begin_job (np/runner.cxx)
by 0x408DD6: np::runner_t::run_tests (np/runner.cxx)
by 0x40AD16: np_run_tests (np/runner.cxx)
by 0x40529A: main (main.c)

FAIL tnassert.assert
```

### 1.7.5 Invalid Memory Accesses

One of the plague spots of coding in C is the ease with which the Code Under Test can accidentally perform invalid memory accesses such as following a null pointer.

When this happens under NovaProva, Valgrind detects it first and emits a useful analysis containing:

- a stack trace,
- line numbers,
- the fault address, and
- information about where the fault address points.

NovaProva then gracefully fails the test. Here's an example:

```
np: running: "mytest.segv"
About to follow a NULL pointer
==32587== Invalid write of size 1
==32587==
...
==32587==
==32587==
==32587== Address 0x0 is not stack'd, malloc'd or (recently) free'd
==32587== Process terminating with default action of signal 11 (SIGSEGV)
==32587== Access not within mapped region at address 0x0
EVENT SIGNAL child process 32587 died on signal 11
at 0x804AD40: test_segv (mytest.c:22)
by 0x804DEF6: np_run_tests (runner.cxx:665)
by 0x804ACF6: main (testrunner.c:31)
FAIL mytest.crash_and_burn
np: 1 run 1 failed
```

### 1.7.6 Buffer Overruns

Buffer overruns are when C code accidentally walks off the end of a buffer, corrupting memory beyond the buffer. This is a classic security vulnerability and an important class of errors in C programs.

When this happens under NovaProva, Valgrind detects it first and emits a useful analysis. Depending on the exact failure mode, Valgrind might either just print the analysis or it might deliver a SEGV to the program. In either case, NovaProva catches it and gracefully fails the test. Here's an example:

```
np: running: "tnoverrun.heap_overrun_small"
about to overrun a buffer by a small amount
==6986== Invalid write of size 1
==6986==    at 0x4C29D28: memcpy (mc_replace_strmem.c:882)
==6986==    by 0x4049E8: do_a_small_overrun (tnoverrun.c:22)
==6986==    by 0x404A8E: test_heap_overrun_small (tnoverrun.c:39)
==6986==    by 0x4164A5: np::spiegel::function_t::invoke(std::vector
↪<np::spiegel::value_t, std::allocator<np::spiegel::value_t> >) const (spiegel.
↪cxx:606)
==6986==    by 0x4085F0: np::runner_t::run_function(np::functype_t,␣
↪np::spiegel::function_t*) (runner.cxx:526)
==6986==    by 0x409029: np::runner_t::run_test_code(np::job_t*) (runner.cxx:650)
==6986==    by 0x4092F2: np::runner_t::begin_job(np::job_t*) (runner.cxx:710)
==6986==    by 0x4075B6: np::runner_t::run_tests(np::plan_t*) (runner.cxx:147)
==6986==    by 0x409502: np_run_tests (runner.cxx:822)
==6986==    by 0x404CDE: main (main.c:102)
==6986==  Address 0x6b58801 is 1 bytes after a block of size 32 alloc'd
==6986==    at 0x4C27A2E: malloc (vg_replace_malloc.c:270)
==6986==    by 0x404A5A: test_heap_overrun_small (tnoverrun.c:36)
==6986==    by 0x4164A5: np::spiegel::function_t::invoke(std::vector
↪<np::spiegel::value_t, std::allocator<np::spiegel::value_t> >) const (spiegel.
↪cxx:606)
==6986==    by 0x4085F0: np::runner_t::run_function(np::functype_t,␣
↪np::spiegel::function_t*) (runner.cxx:526)
==6986==    by 0x409029: np::runner_t::run_test_code(np::job_t*) (runner.cxx:650)
==6986==    by 0x4092F2: np::runner_t::begin_job(np::job_t*) (runner.cxx:710)
==6986==    by 0x4075B6: np::runner_t::run_tests(np::plan_t*) (runner.cxx:147)
==6986==    by 0x409502: np_run_tests (runner.cxx:822)
==6986==    by 0x404CDE: main (main.c:102)
==6986==
overran
EVENT VALGRIND 2 unsuppressed errors found by valgrind
FAIL tnoverrun.heap_overrun_small
```

### 1.7.7 Use Of Uninitialized Variables

The accidental use of uninitialised variables is yet another of C's awful failure modes.

When this happens under NovaProva, Valgrind detects it first and emits a useful analysis. Then NovaProva catches it and gracefully fails the test. Here's an example:

```
np: running: "tnuninit.uninitialized_int"
==6020== Conditional jump or move depends on uninitialised value(s)
==6020==    at 0x404A07: test_uninitialized_int (tnuninit.c:27)
==6020==    by 0x4175C9: np::spiegel::function_t::invoke(std::vector
↪<np::spiegel::value_t, std::allocator<np::spiegel::value_t> >) const (spiegel.
↪cxx:606)
==6020==    by 0x40983C: np::runner_t::run_function(np::functype_t,␣
↪np::spiegel::function_t*) (runner.cxx:526)
==6020==    by 0x40A275: np::runner_t::run_test_code(np::job_t*) (runner.cxx:650)
==6020==    by 0x40A53E: np::runner_t::begin_job(np::job_t*) (runner.cxx:710)
==6020==    by 0x408802: np::runner_t::run_tests(np::plan_t*) (runner.cxx:147)
==6020==    by 0x40A74E: np_run_tests (runner.cxx:822)
==6020==    by 0x405172: main (main.c:102)
==6020==
EVENT VALGRIND 1 unsuppressed errors found by valgrind
```

(continues on next page)

(continued from previous page)

```
FAIL tnuninit.uninitialized_int
np: 1 run 1 failed
```

### 1.7.8 Memory Leaks

The accidental leaking of memory which is allocated but never freed, is yet another of C's awful failure modes.

NovaProva asks Valgrind to do an explicit memory leak check after each test finishes; Valgrind will print a report showing how much memory was leaked and the stack trace of where each leak was allocated. If the test caused memory leaks, NovaProva fails the test. Here's an example:

```
np: running: "mytest.memleak"
Valgrind report
About to do leak 32 bytes from malloc()
Test ends
==779== 32 bytes in 1 blocks are definitely lost in loss record 9 of 54
==779==    at 0x4026FDE: malloc ...
==779==    by 0x804AD46: test_memleak (mytest.c:23)
...
==779==    by 0x804DEFA: np_run_tests (runner.cxx:665)
==779==    by 0x804ACF6: main (testrunner.c:31)
EVENT VALGRIND 32 bytes of memory leaked
FAIL mytest.memleak
np: 1 run 1 failed
```

### 1.7.9 File Descriptor Leaks

A more subtle kind of resource leak is a file descriptor leak. This typically happens in code which reads a file, encounters an error condition, and while handling the error forgets to `fclose()` the file. This kind of problem can be very insidious in long-running server code.

NovaProva detects file descriptor leaks by scanning the test child process' file descriptor table before and after each test and looking for leaks. If the test (or any of the fixture code) caused a file descriptor leak, NovaProva fails the test. Here's an example:

```
np: running: "tnfdleak.leaky_test"
MSG leaking fd for .leaky_test.dat
EVENT FDLEAK test leaked file descriptor 5 -> /build/novaprova/tests/.leaky_test.dat
FAIL tnfdleak.leaky_test
np: running: "tnfdleak.leaky_fixture"
MSG leaking fd for .leaky_fixture.dat
EVENT FDLEAK test leaked file descriptor 3 -> /build/novaprova/tests/.leaky_fixture.
↪dat
FAIL tnfdleak.leaky_fixture
np: 2 run 2 failed
```

### 1.7.10 Looping, Deadlocked, Or Slow Tests

Sometimes the Code Under Test enters an infinite loop, or causes a deadlock between two or more threads. NovaProva uses a per-test timeout to detect these cases; if the test runs longer than the timeout NovaProva will kill the child test process with `SIGTERM` and gracefully fail the test.

The basic test timeout is 30 seconds. NovaProva automatically detects and adjusts the timeout in certain situations. When the test executable is being run under gdb, NovaProva disables the test timeout. When the test executable is being run under Valgrind (the default behavior), NovaProva triples the timeout.

```
np: running: "mytest.slow"
Test runs for 100 seconds
Have been running for 0 sec
Have been running for 10 sec
Have been running for 20 sec
Have been running for 30 sec
Have been running for 40 sec
Have been running for 50 sec
Have been running for 60 sec
Have been running for 70 sec
Have been running for 80 sec
EVENT TIMEOUT Child process 2294 timed out, killing
EVENT SIGNAL child process 2294 died on signal 15
FAIL mytest.slow
np: 1 run 1 failed
```

### 1.7.11 C++ Exceptions

A failure mode unique to C++ is the uncaught exception. NovaProva catches all uncaught exceptions, by setting up a global default terminate handler. If the Code Under Test throws an exception which is not caught, NovaProva will print a message including the exception type, the result of `e.what()` if the exception is of a subclass of `std::exception`, and the stacktrace of the `throw` statement. NovaProva will then gracefully fail the test.

```
np: running: "mytest.slow"
np: running: "tnexcept.uncaught_exception"
MSG before call to bar
EVENT EXCEPTION terminate called with exception foo::exception: Oh that went badly
at 0x416C2D: np::spiegel::describe_stacktrace (np/spiegel/spiegel.cxx)
by 0x426FCA: np::event_t::with_stack (np/event.cxx)
by 0x426A7B: __np_terminate_handler (iexcept.c)
...
FAIL tnexcept.uncaught_exception
```

## 1.8 Output Formats

NovaProva supports two different test result output formats. You can select between these using the `--format` option to the test executable, or by calling `np_set_output_format()` if you write your own `main()` routine. If multiple formats are required then provide a comma separated list when using the `--format` option or call `np_set_output_format()` multiple times.

**`text`** A simple text output format, designed to be read by humans. Output goes to stderr. Each completed test shows a single line, beginning with one of the key words `PASS`, `FAIL` or `N/A`, immediately when the test completes. These lines are interspersed with whatever output to stdout or stderr the tests themselves may emit. After all tests are complete a 1-line summary describes how many tests were run and how many failed. This is the default output format.

**`junit`** An XML format, designed to emulate the test report emitted by the JUnit library and read by many other tools, such as Jenkins CI. This output format creates a directory called `reports` containing multiple XML files called `TEST-filename.xml`, one for each test source file name. Each test's pass/fail status, elapsed run time, and any output to stdout or stderr are stored in the XML file.

## 1.9 Fixtures

When you write a lot of tests you will sooner or later find yourself pasting the same code over and over into multiple tests. Typically this is code which initializes the Code Under Test and the environment it runs in, getting it into a state where the tests can usefully be run. Examples include creating files, setting environment variables, and populating a database. Nobody likes repeated code, so we have a problem.

It's tempting to solve the problem by doing one of two things:

1. Pull the initialization code out of the tests entirely, say into a separate shell script, and let the tests just assume that the environment is initialized.
2. Write a test which does the initialization, and then make the other tests depend on it. See *Dependencies*.

Both of these are bad ideas because they clash with the principles that tests should be *isolated* and *self-contained*.

A better approach would be to write a function which does all the setup (a *setup function*) and call it from each test function. In general there will also be some other code to undo the effects of setup function and cleanup the environment again, this could go into a *teardown function* which is called at the end of every test.

This is much better, but there are still some issues. Most pressingly, if a test fails then the call from the end of the test function to the teardown function will never happen. If the setup function creates a resource that has a lifetime outside of it's process, like a file or a shared memory segment, that resource will be leaked.

NovaProva, like the xUnit family of test frameworks, provides a feature called fixtures, to make this easy and correct.

A fixture is simply a matched pair of setup and teardown functions attached to a test node, with the following automatic behavior implemented in the framework:

- The setup function on a test node is automatically called before any test function at or below that test node is called.
- The teardown function on a test node is automatically called after any test function at or below that test node is called.
- If the setup function fails, the test function and the teardown function are not run.
- If the test function is run, the teardown function will always be run, regardless of whether the test function succeeded or finished.
- If either the setup or teardown function fails the test is marked FAILed.
- Either the setup or teardown functions may be missing.
- If setup functions are defined for multiple ancestor test nodes of a test function, the setup functions are run from the rootmost to the leafmost, i.e. from the least specific to the most specific.
- Multiple teardown functions are run in the reverse order, i.e. from leafmost to rootmost.

Like test functions, setup and teardown functions are discovered at runtime using Reflection. They are functions which take no arguments and return an integer, with any of the following names.

- `setup`
- `Setup`
- `set_up`
- `init`
- `Init`
- `teardown`
- `tearDown`

- `Teardown`
- `TearDown`
- `tear_down`
- `cleanup`
- `Cleanup`

Here's an example of defining fixture functions.

```
static int set_up(void)
{
    fprintf(stderr, "Setting up fixtures\n");
    return 0;
}

static int tear_down(void)
{
    fprintf(stderr, "Tearing down fixtures\n");
    return 0;
}
```

Here's an example of fixtures running.

```
np: running
np: running: "mytest.one"
Setting up fixtures
Running test_one
Finished test_one
Tearing down fixtures
PASS mytest.one
np: running: "mytest.two"
Setting up fixtures
Running test_two
Finished test_two
Tearing down fixtures
PASS mytest.two
np: 2 run 0 failed
```

## 1.10 Mocking

One of the big problems when writing unit tests is how to pick apart the morass of interconnections between modules to find a subset of your program which can be tested by itself. To construct an example, let's say you want to test the `foo` module, but it makes calls to the `bar` module, which in turn does a REST call to the `baz` server. How do you unit test `foo`?

One answer is to test from the edges of the dependency graph inward, i.e. first test the `baz` server, then test the `bar` module using a live `baz` server, then finally test the `foo` module using live `bar` and `baz`. This can work, but it has a number of problems:

- the program may have loops in the dependency graph, e.g. the `baz` server calls into the `foo` module.
- it may be difficult or impossible to get a component into a reproducible initial state, e.g. `baz` is an Oracle database.
- you may need to test a module's response to unusual behaviors in it's downstream modules, e.g. you have to test that `foo` handles `bar` returning a particular error code.

The better answer is: mocking.

### 1.10.1 What Is Mocking?

Mocking is a technique used only in test builds, to replace modules we don't want to be testing with fake (*mock*) versions which present the same interfaces but have simpler and more controllable behavior. The process of replacing is called *mocking* and the replaced functions are *mocked*.

Let's construct an example. Let's say one of the functions in our `foo` module looks like this:

```c
void foo_mustache(int x)
{
    BarTxn *txn = bar_begin();
    int r = bar_shoreditch(x);
    if (r < 0)
    {
        fprintf(stderr, "shoreditch failed: %d", -r);
        return;
    }
    bar_commit(txn);
}
```

There's a common type of bug in this code. If the `bar_shoreditch()` function returns an error, the `BarTxn` object created by the earlier call to `bar_begin()` is leaked. But the bug only appears if `bar_shoreditch()` returns with an error, and this hardly ever happens with the real `bar` module.

To write a unit test that tickles this bug, we need to mock the `bar_shoreditch()` function. We want to create a version of that function which always returns an error, and arrange for it to be called whenever the Code Under Test tries to call the real `bar_shoreditch()`. Here's a way to do that using NovaProva

```c
int mock_bar_shoreditch(int x)
{
    if (x == 42)
        return -ENOENT;
    return 0;
}
```

### 1.10.2 How It Works

When NovaProva finds a function whose name starts with the letters `mock_`, it automatically adds that function as a mock to all the tests defined in the same source file (more precisely, the mock is attached to the testnode corresponding to the file). Mocks are automatically installed before their tests start and are automatically uninstalled again when their tests finish. While those tests are running, any attempt to call `bar_shoreditch()` from any part of the test function or the Code Under Test, will instead call the mocked function `mock_bar_shoreditch()` which then simulates whatever behavior we want for the test (in this case, return an error if the input is 42). Any other tests are unaffected by this behavior; if they call `bar_shoreditch()` that will call the real `bar_shoreditch()`.

The mechanism that NovaProva uses operates at runtime, not at link time like some C mocking libraries. Nor does it rely on C++ virtual functions, like some other mocking libraries. Instead it uses a very platform-specific mechanism similar to a debugger breakpoint, which causes any call to the original function to trap into some special NovaProva handler code which calls the mock function. This mechanism is incredibly powerful and has a number of implications.

### 1.10.3 Adding Mocks Dynamically

Mocks can be inserted and removed partway through a test. This allows mocking only some of the calls to a particular function, and letting other calls go through to the real implementation. For example

```
static int fail_with_enoent(int x)
{
    return -ENOENT;
}

void test_some_ditches(void)
{
    foo_mustache(0);     /* calls the real bar_shoreditch() */
    foo_mustache(1);
    np_mock(bar_shoreditch, fail_with_enoent);
    foo_mustache(2);     /* calls the mock, leaks BarTxn */
    np_unmock(bar_shoreditch);
    foo_mustache(3);     /* calls the real bar_shoreditch() again */
}
```

### 1.10.4 Using Many Simple Mocks

Because mocks can be added partway through a test, you can write a test which uses different mock functions in different parts of the test. This usually means that each mock function can be simple and easy to understand, instead of trying to write a single big complicated mock function with lots of logic designed to handle all the different tests you will ever use. For example

```
static int fail_with_enoent(int x)
{
    return -ENOENT;
}
static int fail_with_econnrefused(int x)
{
    return -ECONNREFUSED;
}
static int fail_with_eaccess(int x)
{
    return -EACCESS;
}

void test_failing_all_over(void)
{
    np_mock(bar_shoreditch, fail_with_enoent);
    foo_mustache(4);
    np_mock(bar_shoreditch, fail_with_econnrefused);
    foo_mustache(5);
    np_mock(bar_shoreditch, fail_with_eaccess);
    foo_mustache(6);
}
```

Note that you don't need to explicitly call `np_unmock()` - any mocks installed dynamically during the test are automatically uninstalled after the test finishes.

### 1.10.5 Failure Injection

You can use mocks to implement simple forms of failure injection. For example, there's a second common type of bug in the `foo_mustache()` implementation above: it doesn't check for a NULL return from `bar_begin()`, which might happen in rare cases like a memory allocation failure. Here's how you would test for that bug:

```
static void *returns_null(size_t sz)
{
    return NULL;
}
void test_malloc_failure(void)
{
    foo_mustache(7);
    np_mock(malloc, returns_null);
    foo_mustache(8);     /* malloc() call in bar_begin() fails */
    np_unmock(malloc)
    foo_mustache(9);
}
```

### 1.10.6 Mocking By Name

In the above examples you saw how you can mock a function using the function's address. NovaProva will also let you mock a function using the function's name. You don't even need to be able to call function normally, so you can mock static functions in other modules (as long as that function has a known and unique name). For example:

```
/* this function is not visible outside the bar module
 * and is called from bar_begin() */
static BarTxn *bar_txn_alloc(void)
{
    BarTxn *txn = (BarTxn *)malloc(sizeof(BarTxn));
    if (!txn)
        return NULL;
    txn->id = nextid++;
    txn->state = FETAL;
    txn->items = NULL;
    return txn;
}

/* in test code */
void test_txn_alloc_failure(void)
{
    np_mock_by_name("bar_txn_alloc", returns_null);
    foo_mustache(10);       /* bar_txn_alloc() returns NULL */
}
```

## 1.11 Parameters

Parameterized tests are the solution for when you want to run the same test code several times with just a little bit different each time.

In NovaProva, test parameters are a `static const char*` variable attached to a test node, which is set to one of a number of fixed values by the NovaProva library. Parameters are declared with the `NP_PARAMETER()` macro. This macro takes two arguments: the name of the variable and a string containing all the possible values, separated by commas or whitespace.

The parameter is attached to the level of the test node tree where it's declared, and applies to all the tests at that level or below, e.g. all the tests declared in the same `.c` file. Those tests are run once for each value of the parameter, with the parameter's variable set to a different value each time. In test results, the name and value of the parameter are shown inside `[]` appended to the full test name.

For example, this parameter declaration

```
NP_PARAMETER(pastry, "donut,bearclaw,danish");
static void test_munchies(void)
{
    fprintf(stderr, "I'd love a %s\n", pastry);
}
```

will result in the `test_munchies()` function being called three times, like this

```
np: running: "mytest.munchies[pastry=donut]"
I'd love a donut
PASS mytest.munchies[pastry=donut]

np: running: "mytest.munchies[pastry=bearclaw]"
I'd love a bearclaw
PASS mytest.munchies[pastry=bearclaw]

np: running: "mytest.munchies[pastry=danish]"
I'd love a danish
PASS mytest.munchies[pastry=danish]

np: 3 run 0 failed
```

When multiple parameters apply, the test functions are called once for each of the combinations of the parameters. For example, this pair of parameters

```
NP_PARAMETER(pastry, "donut,bearclaw,danish");
NP_PARAMETER(beverage, "tea,coffee");
static void test_munchies(void)
{
    fprintf(stderr, "I'd love a %s with my %s\n", pastry, beverage);
}
```

will result in the `test_munchies()` function being called six times, like this

```
np: running: "mytest.munchies[pastry=donut][beverage=tea]"
I'd love a donut with my tea
PASS mytest.munchies[pastry=donut][beverage=tea]

np: running: "mytest.munchies[pastry=bearclaw][beverage=tea]"
I'd love a bearclaw with my tea
PASS mytest.munchies[pastry=bearclaw][beverage=tea]

np: running: "mytest.munchies[pastry=danish][beverage=tea]"
I'd love a danish with my tea
PASS mytest.munchies[pastry=danish][beverage=tea]

np: running: "mytest.munchies[pastry=donut][beverage=coffee]"
I'd love a donut with my coffee
PASS mytest.munchies[pastry=donut][beverage=coffee]

np: running: "mytest.munchies[pastry=bearclaw][beverage=coffee]"
```

(continues on next page)

(continued from previous page)

```
I'd love a bearclaw with my coffee
PASS mytest.munchies[pastry=bearclaw][beverage=coffee]

np: running: "mytest.munchies[pastry=danish][beverage=coffee]"
I'd love a danish with my coffee
PASS mytest.munchies[pastry=danish][beverage=coffee]

np: 6 run 0 failed
```

## 1.12 C API Reference

### 1.12.1 Result Macros

These macros can be used in test functions to indicate a particular test result.

**NP_PASS**
Causes the running test to terminate immediately with a PASS result.

You will probably never need to call this, as merely reaching the end of a test function without FAILing is considered a PASS result.

**NP_FAIL**
Causes the running test to terminate immediately with a FAIL result.

**NP_NOTAPPLICABLE**
Causes the running test to terminate immediately with a NOTAPPLICABLE result.

The NOTAPPLICABLE result is not counted towards either failures or successes and is useful for tests whose preconditions are not satisfied and have thus not actually run.

### 1.12.2 Assert Macros

These macros can be used in test functions to check a particular condition, and if the check fails print a helpful message and FAIL the test. Treat them as you would the standard `assert` macro.

**NP_ASSERT**(cc)
Test that a given boolean condition is true, otherwise FAIL the test.

**NP_ASSERT_TRUE**(a)
Test that a given boolean condition is true, otherwise FAIL the test.

This is the same as `NP_ASSERT` except that the message printed on failure is slightly more helpful.

**NP_ASSERT_FALSE**(a)
Test that a given boolean condition is false, otherwise FAIL the test.

**NP_ASSERT_EQUAL**(a, b)
Test that two signed integers are equal, otherwise FAIL the test.

**NP_ASSERT_NOT_EQUAL**(a, b)
Test that two signed integers are not equal, otherwise FAIL the test.

**NP_ASSERT_PTR_EQUAL**(a, b)
Test that two pointers are equal, otherwise FAIL the test.

**NP_ASSERT_PTR_NOT_EQUAL**(a, b)
Test that two pointers are not equal, otherwise FAIL the test.

**`NP_ASSERT_NULL`**(a)
Test that a pointer is NULL, otherwise FAIL the test.

**`NP_ASSERT_NOT_NULL`**(a)
Test that a pointer is not NULL, otherwise FAIL the test.

**`NP_ASSERT_STR_EQUAL`**(a, b)
Test that two strings are equal, otherwise FAIL the test.

Either string can be NULL; NULL compares like the empty string.

**`NP_ASSERT_STR_NOT_EQUAL`**(a, b)
Test that two strings are not equal, otherwise FAIL the test.

Either string can be NULL, it compares like the empty string.

### 1.12.3 Syslog Matching

These functions can be used in a test function to control the how the test behaves if the Code Under Test attempts to emit messages to `syslog`. See *Messages Emitted To syslog()* for more information.

void **`np_syslog_fail`**(**const** char **re*)
Set up to FAIL the test on syslog messages matching a regexp.

From this point until the end of the test, if any code emits a message to `syslog` whose text matches the given regular expression, the test will FAIL immediately as if `NP_FAIL` had been called from inside `syslog`.

**Parameters**

- `re` - POSIX extended regular expression to match

void **`np_syslog_ignore`**(**const** char **re*)
Set up to ignore syslog messages matching a regexp.

From this point until the end of the test function, if any code emits a message to `syslog` whose text matches the given regular expression, nothing will happen. Note that this is the default behaviour, so this call is only useful in complex cases where there are multiple overlapping regexps being used for syslog matching.

**Parameters**

- `re` - POSIX extended regular expression to match

void **`np_syslog_match`**(**const** char **re*, int *tag*)
Set up to count syslog messages matching a regexp.

From this point until the end of the test function, if any code emits a message to `syslog` whose text matches the given regular expression, a counter will be incremented and no other action will be taken. The counts can be retrieved by calling `np_syslog_count`. Note that *tag* does not need to be unique; in fact always passing 0 is reasonable.

**Parameters**

- `re` - POSIX extended regular expression to match
- `tag` - tag for later matching of counts

unsigned int **`np_syslog_count`**(int *tag*)
Return the number of syslog matches for the given tag.

Calculate and return the number of messages emitted to `syslog` which matched a regexp set up earlier using `np_syslog_match`. If *tag* is less than zero, all match counts will be returned, otherwise only the match counts for regexps registered with the same tag will be returned.

**Return**

count of matched messages

**Parameters**

- `tag` - tag to choose which matches to count, or -1 for all

### 1.12.4 Parameters

These functions can be used to set up parameterized tests. See *Parameters* for more information.

**`NP_PARAMETER`**(nm, vals)

Statically define a test parameter and its values.

Define a `static char*` variable called *nm*, and declare it as a test parameter on the testnode corresponding to the source file in which it appears, with a set of values defined by splitting up the string literal *vals* on whitespace and commas. For example: Declares a variable called `db_backend` in the current file, and at runtime every test function in this file will be run twice, once with the variable `db_backend` set to `"mysql"` and once with it set to `"postgres"`.

```
NP_PARAMETER(db_backend, "mysql,postgres");
```

**Parameters**

- `nm` - C identifier of the variable to be declared
- `vals` - string literal with the set of values to apply

### 1.12.5 Dynamic Mocking

These functions can be used in a test function to dynamically add and remove mocks. See *Mocking* for more information.

void **`np_unmock_by_name`**(**`const`** char **fname*)

Uninstall a dynamic mock by function name.

Uninstall any dynamic mocks installed earlier by `np_mock_by_name` for function *fname*. Note that dynamic mocks will be automatically uninstalled at the end of the test, so calling `np_unmock_by_name()` might not even be necessary in your tests.

**Parameters**

- `fname` - the name of the function to mock

**`np_mock`**(fn, to)

Install a dynamic mock by function pointer.

Installs a temporary dynamic function mock. The mock can be removed with `np_unmock()` or it can be left in place to be automatically uninstalled when the test finishes.

**Parameters**

- `fn` - the function to mock
- `to` - the function to call instead

Note that if `np_mock()` may be called in a fixture setup routine to install the mock for every test in a test source file.

**np_unmock**(fn)

Uninstall a dynamic mock by function pointer.

Uninstall any dynamic mocks installed earlier by `np_mock` for function *fn*. Note that dynamic mocks will be automatically uninstalled at the end of the test, so calling `np_unmock()` might not even be necessary in your tests.

**Parameters**

- `fn` - the address of the function to mock

**np_mock_by_name**(fname, to)

Install a dynamic mock by function name.

Installs a temporary dynamic function mock. The mock can be removed with `np_unmock_by_name()` or it can be left in place to be automatically uninstalled when the test finishes.

**Parameters**

- `fname` - the name of the function to mock
- `to` - the function to call instead

Note that if `np_mock_by_name()` may be called in a fixture setup routine to install the mock for every test in a test source file.

### 1.12.6 Main Routine

These functions are for writing your own `main()` routine. You probably won't need to use these, see *Main Routine*.

np_plan_t *`np::`**`np_plan_new`**(void)

Create a new plan object.

A plan object can be used to configure a `np_runner_t` object to run (or list to stdout) a subset of all the discovered tests. Note that if you want to run all tests, you do not need to create a plan at all; passing NULL to `np_run_tests` has that effect.

**Return**

a new plan object

void `np::`**`np_plan_delete`**(np_plan_t **plan*)

Destroys a plan object.

**Parameters**

- `plan` - the plan object to destroy

bool `np::`**`np_plan_add_specs`**(np_plan_t **plan*, int *nspec*, **const** char ***spec*)

Add a sequence of test specifications to the plan object.

Each test specification is a string which matches a testnode in the discovered testnode hierarchy, and will cause that node (plus all of its descendant nodes) to be added to the plan. The interface is designed to take command-line arguments from your test runner program after options have been parsed with `getopt`. Alternately you can call `np_plan_add_specs` multiple times.

**Return**

false if any of the test specifications could not be found, true on success.

**Parameters**

- `plan` - the plan object

- `nspec` - number of specification strings
- `spec` - array of specification strings

void **`np_set_concurrency`**(np_runner_t **runner*, int *n*)

Set the limit on test job parallelism.

Set the maximum number of test jobs which will be run at the same time, to *n*. The default value is 1, meaning tests will be run serially. A value of 0 is shorthand for one job per online CPU in the system, which is likely to be the most efficient use of the system.

**Parameters**

- `runner` - the runner object
- `n` - concurrency value to set

void **`np_list_tests`**(np_runner_t **runner*, np_plan_t **plan*)

Print the names of the tests in the plan to stdout.

If *plan* is NULL, all the discovered tests will be listed in testnode tree order.

**Parameters**

- `runner` - the runner object
- `plan` - optional plan object

bool **`np_set_output_format`**(np_runner_t **runner*, **`const`** char **fmt*)

Set the format in which test results will be emitted.

Available formats are:

- **“junit”** a directory called `reports/` will be created with XML files in jUnit format, suitable for use with upstream processors which accept jUnit files, such as the Jenkins CI server.
- **”text”** a stream of tests and events is emitted to stdout, co-mingled with anything emitted to stdout by the test code. This is the default if `np_set_output_format` is not called.

Note that the function is a misnomer, it actually **adds** an output format, so if you call it twice you will get two sets of output.

Returns true if `fmt` is a valid format, or false on error.

**Parameters**

- `runner` - the runner object
- `fmt` - string naming the output format

int **`np_run_tests`**(np_runner_t **runner*, np_plan_t **plan*)

Runs all the tests described in the *plan* object.

If *plan* is NULL, all the discovered tests will be run in testnode tree order.

**Return**

0 on success or non-zero if any tests failed.

**Parameters**

- `runner` - the runner object
- `plan` - optional plan object

np_runner_t ***np_init** (void)

Initialise the NovaProva library.

You should call `np_init` to initialise NovaProva before running any tests. It discovers tests in the current executable, and returns a pointer to a `np_runner_t` object which you can pass to `np_run_tests` to actually run the tests.

The first thing the function does is to ensure that the calling executable is running under Valgrind, which involves re-running the process. So be aware that any code between the start of `main` and the call to `np_init` will be run twice in two different processes, the second time under Valgrind.

The function also sets a C++ terminate handler using `std::set_terminate()` which handles any uncaught C++ exceptions, generates a useful error message, and fails the running test.

**Return**

a new runner object

void **np_done** (np_runner_t **runner*)

Shut down the NovaProva library.

Destroys the given runner object and shuts down the library.

**Parameters**

- `runner` - The runner object to destroy

### 1.12.7 Miscellany

int **np_get_timeout** ()

Get the timeout for the currently running test.

If called outside of a running test, returns 0. Note that the timeout for a test can vary depending on how it's run. For example, if the test executable is run under a debugger the timeout is disabled, and if it's run under Valgrind (which is the default) the timeout is tripled.

**Return**

timeout in seconds of currently running test

## 1.13 Porting NovaProva

This chapter is intended as a resource for developers wanting to undertake a port of NovaProva to another platform.

### 1.13.1 Level of Difficulty

The NovaProva library contains significant levels of platform dependency, so porting it is a non-trivial task. To undertake a port you will need to have a working knowledge of details of the following components of the target platform.

- the assembly language
- the C ABI (i.e. function calling conventions and the like)
- some details of the C runtime library (e.g. where process arguments are stashed)
- some details of the kernel ABI (e.g. the shape of the stack frame used to deliver signals to user processes).

Many of these details are not documented; you will have to discover them by reading the system source, or for closed source systems applying reverse engineering techniques. Some of these details do not form part of the system ABI and may be subject to unexpected change.

As a rough guide, porting NovaProva to a new platform is more complex than porting any C library (except perhaps media codecs) and less complex than porting Valgrind or the Linux kernel.

### 1.13.2 Executable File Format

NovaProva uses the BFD library from the GNU binutils package to handle details of the executable file format (e.g. ELF on modern Linux systems). NovaProva uses only the abstract (i.e. format-independant) part of the BFD API, and only for a stricly limited set of tasks (such as discovering segment boundaries and types). Hopefully this will require little porting to other executable file formats (e.g. COFF or Mach objects).

### 1.13.3 Debugging Information

NovaProva depends deeply on the DWARF debugging format. There is a considerable body of code which parses DWARF and depends on DWARF formats and semantics. If your platform does not use DWARF natively, or cannot be convinced to by the use of compiler flags such as `-gdwarf2`, then porting NovaProva will be very much harder and you should contact the mailing list for advice.

### 1.13.4 Valgrind

Valgrind is an advanced memory debugging tool (actually, it's a program simulator which happens debug memory as a side effect). NovaProva is designed to make use of Valgrind's powerful bug discovery features. If your platform doesn't support Valgrind you're going to get much less value out of NovaProva than if you had Valgrind, and NovaProva is going to be missing a great many test failure cases that you really want to be caught.

For this reason, NovaProva will fail to build without Valgrind.

### 1.13.5 Platform Specific Code

NovaProva is written to isolate the platform dependent code to a small set of files with a well-defined interface. Partly this is good program practice to set the scene for future ports (we shall see how well this succeeded when the first new port is done). But partly it is due to current necessity, as NovaProva is sufficiently sensitive to platform details that 32 bit and 64 bit x86 Linux platforms need different code.

#### Code Layout

The platform specific code is contained in the C++ namespace `np::spiegel::platform` and is implemented in `.cxx` files in the directory `np/spiegel/platform/`. This follows the usual NovaProva convention where namespaces and directories have exactly the same shape.

Generally there will be two platform specific files, one containing code which depends on the OS alone (e.g. `linux.cxx`) and the other containing code which depends on the combination of the OS the machine architecture (e.g. `linux_x86.cxx`).

### Build Infrastructure

The `configure.ac` script decides which platform specific source files are built. It can also add compiler flags, so you can use `#ifdef` if you really feel the need.

Your first step is to add detection of your platform to `configure.ac`. Find the code that begins

```
case "$target_os" in
```

and add a new case for your platform operating system.

At this point you need to set the `$os` variable to the short name of the platform operating system, e.g. `x86`. This is going to be used to choose a filename `$os.cxx`, so the name must be short and contain no spaces or / characters. Ideally it will be entirely lowercase, to match the NovaProva conventions.

Optionally, you can append to the `$platform_CFLAGS` variable if there are some compiler flags (e.g. `-DGNU_SOURCE`) that should be set for that platform only. These flags are applied to every C++ file not just the platform specific one.

Your next step is to find the code that begins

```
case "$target_cpu" in
```

and add a new case for your platform hardware architecture.

At this point you need to set three variables.

- `$arch` is the short name of the platform hardware architecture, e.g. `x86`. This is going to be used to construct a filename `${os}_${arch}.cxx` and to add a compile flag `-D_NP_$arch` so it must contain only alphanumerics and the underscore character. Ideally it will be entirely lowercase, to match the NovaProva conventions.
- `$addrsize` is a decimal literal indicating the size of a platform virtual address in bytes, e.g. `4` for 32-bit platforms.
- `$maxaddr` is a C unsigned integer literal indicating the maximum value of a virtual address, e.g. `0xffffffffUL` on 32-bit platforms.

Optionally, you can also append to the `$platform_CFLAGS` variable here.

Finally you should ensure that the following two C++ source files exist.

- `np/spiegel/platform/${os}.cxx`
- `np/spiegel/platform/${os}_${arch}.cxx`

### Platform Specific Functions

Your next step is to add your implementations of the platform specific functions to one of those two platform specific files. Generally you should add a function to the most general of the two files in which it can be implemented without using `#ifdef`. For example, the function `get_argv()` works identically on all Linux platforms so it's implemented in `linux.cxx`, whereas `install_intercept()` varies widely between 32-bit x86 and 64-bit x86 so it's implemented twice in `linux_x86.cxx` and `linux_x86_64.cxx`.

The remainder of this section will describe the various platform specific functions, their purpose and the requirements placed upon them.

### Get Commandline

```
bool get_argv(int *argcp, char ***argvp)
```

Returns in `*argcp` and `*argvp` the original commandline argument vector for the process, and `true` on success. Modern C runtime libraries will store the commandline argument vector values passed to `main()` in global variables in the C library before calling `main()`. This method retrieves those values so that NovaProva can use them when forking itself to run Valgrind. Because no standard or convention describes these variables, their names are platform specific; it is also possible on some platforms that no such variables might exist and the argument vector might need to be deduced by looking in the kernel aux vector or a filesystem like /proc.

### Get Executable Name

```
char *self_exe()
```

Returns a newly allocated string representing the absolute pathname of the process' executable. This is used when NovaProva forks itself to run Valgrind. The Linux code uses a `readlink()` call on the symlink `/proc/self/exe`.

### List Loaded Libraries

```
vector<linkobj_t> get_linkobjs()
```

Returns an STL vector of `linkobj_t` structures which collectively describe all the objects dynamically linked into the current executable. Typically this means the first `linkobj_t` describes the program itself and this is followed by one `linkobj_t` for each dynamically linked library. This information can be extracted with a platform specific call into the runtime linker. For Linux glibc systems that call is `dl_iterate_phdr()`.

### Normalise an Address

```
np::spiegel::addr_t normalise_address(np::spiegel::addr_t addr)
```

Takes a virtual address and returns a possibly different virtual address which is normalized. Normalized addresses can be used for comparison, i.e. if two normalized addresses are the same they refer to the same C function. This apparently obvious property is not true of function addresses in a dynamically linked object where the function whose address is being taken is linked from another dynamic object; the address used actually points into the Procedure Linkage Table in the calling object.

In order to implement this, the platform specific code needs to know where the various PLTs are linked into the address space. The platform specific function

```
void add_plt(const np::spiegel::mapping_t &m)
```

is called from the object handling code to indicate the boundaries of the PLT in each object.

### Remap Text

```
int text_map_writable(addr_t addr, size_t len)

int text_restore(addr_t addr, size_t len)
```

These functions are used when inserting intercepts to ensure that some code in a `.text` or similar segment is mapped writable (modern OSes will map all code read-only by default for security reasons). The Linux implementation uses the `mprotect()` system call and reference counts pages to allow for the case where multiple intercepts are installed in the same page. This code should also work on most platforms that support the `mprotect()` call.

### Get A Stacktrace

```
vector<np::spiegel::addr_t> get_stacktrace()
```

Returns a stacktrace as a vector of code addresses (`%eip` samples in x86) of the calling address, in order from the innermost to the outermost. The current (somewhat disappointing) implementation walks stack frames using the frame pointer, which is somewhat fragile on x86 platforms (where libraries are often shipped built with the `-fomit-frame-pointer` flag, which breaks this technique). This function is used only to generate error reports that are read by humans, so it really should be implemented in a way which emphasizes accuracy over speed, e.g. using the DWARF2 unwind information to pick apart stack frames accurately.

### Detect Debuggers

```
bool is_running_under_debugger()
```

Returns `true` if and only if the current process is running under a debugger such as gdb. This is needed on some architectures to change the way that intercepts are implemented; different instructions need to be inserted to avoid interfering with debugger breakpoints. Also, some features like test timeouts are disabled when running under a debugger if they would do more harm than good. The Linux implementation digs around in the `/proc` filesystem to discover whether the current process is running under `ptrace()` and if so compares the commandline of the tracing process against a whitelist.

### Describe File Descriptors

```
vector<string> get_file_descriptors()
```

Returns an STL vector of STL strings in which the *fd*-th entry is a human-readable English text description of file descriptor *fd*, or an empty string if file descriptor *fd* is closed. This function is called before and after each test is run to discover file descriptor leaks in test code, so the returned descriptions should be consistent between calls. File descriptors used by Valgrind should not be reported. The Linux implementation uses the `/proc/self/fd` directory.

### Install Intercept

```
int install_intercept(np::spiegel::addr_t addr, intstate_t &state, std::string &err)

int uninstall_intercept(np::spiegel::addr_t addr, intstate_t &state, std::string &err)
```

These functions are the most difficult but most rewarding part of porting NovaProva. Intercepts are the key technology that drives advanced NovaProva features like mocks, redirects, and failpoints. An intercept is basically a breakpoint inserted into code, similar to what a debugger uses, but instead of waking another process when triggered an intercept calls code in the same process.

These two functions are called to respectively install an intercept at a given address and remove it again. The caller normalizes the address and takes care to only install one intercept at a given address, so for example `install_intercept` will not be called twice for the same address without a call to `uninstall_intercept`. The `intstate_t` type is defined in the header file `np/spiegel/platform/common.hxx` for all ports (using `#ifdef`) and contains any state which might be useful for uninstalling the intercept, e.g. the original instructions which were replaced at install time. The install function can assume that no NovaProva intercept is already installed at the given address, but it should take care to handle the case where a debugger like gdb has independently inserted it's own breakpoint.

Unlike debugger breakpoints, intercepts are always inserted at the first byte of an instruction, at the beginning of the function prologue. This can be a useful simplifying assumption; for example on x86 the first instruction in most functions is `pushl %ebp` whose binary form is the byte 0x55.

The install function will presumably be modifying 1 or more bytes in the instruction stream to contain some kind of breakpoint instruction; it should call `text_map_writable()` before modifying the bytes to ensure the byte range is mapped writable. Similarly the uninstall function should call `text_restore()` after restoring the original instruction, to potentially map the bytes read-only again. Both functions should call the Valgrind macro `VALGRIND_DISCARD_TRANSLATIONS()` after modifying the instruction stream; Valgrind uses a JIT-like mechanism for caching translated native instructions and it is important that this cache not contain stale translations.

Both functions return `0` on success. On error they set `err` to a human-readable English error string and return `-1`.

While an intercept is installed, any attempt to execute the code at `addr` should not execute the original code but instead cause a special function called the trampoline to be called (e.g. via a Unix signal handler). The trampoline has the following responsibilities.

1. Extract (from registers, the exception frame on the stack, or the calling function's stack frame) the arguments to the intercepted function, and store them in an instance of a platform-specific class derived from `np::spiegel::call_t`, which implements the `get_arg()` and `set_arg()` methods.
2. Call the static method `intercept_t::dispatch_before()` with the intercepted address (typically the faulting PC in the exception stack frame) and a reference to the `call_t` object.
3. Handle any of the possible side effects of `dispatch_before()`
    1. If `call_t.skip_` is `true`, arrange to immediately return `call_t.retval_` to the calling function, without executing the intercepted function and without calling `dispatch_after()`.
    2. If the redirect function `call_t.redirect_` is non-zero, arrange to call that instead of the intercepted function.
    3. Arrange for the intercepted (or redirect) function to be called with the arguments in the `call_t` object.
4. Call the intercepted (or redirect) function.
5. Store the return value of the intercepted (or redirect) function in `call_t.retval_`.
6. Call the static method `intercept_t::dispatch_after()` with the same arguments as `dispatch_before()`.
7. Arrange to return `call_t.retval_` (which may have been changed as a side effect of calling `dispatch_after()`) to the calling function.

Currently NovaProva intercepts are not required to be thread-safe. This means that the signal handler and trampoline function can use global state if necessary.

#### Exception Handling

```
char *current_exception_type()
```

Returns a new string describing the C++ type name of the exception currently being handled, or `0` if no exception is being handled.

```
void cleanup_current_exception()
```

Frees any storage associated with the exception currently being handled. If this function does nothing, uncaught C++ exceptions reported by NovaProva will also result in a Valgrind memory leak report.

### 1.13.6 Utility Functions

Some of NovaProva's utility functions have platform-specific features which need to be considered when porting NovaProva.

#### POSIX Clocks

The timestamp code in `np/util/common.cxx` relies on the POSIX `clock_gettime()` function call, with both the `CLOCK_MONOTONIC` and `CLOCK_REALTIME` clocks being used. If your platform does not supply `clock_gettime()` then you should write a compatible replacement. If your platform does not support a monotonic clock, returning the realtime clock is good enough.

#### Page Size

The memory mapping routines in `np/util/common.cxx` use call `sysconf(_SC_PAGESIZE)` to retrieve the system page size from the kernel. This may require a platform-specific replacement.

### 1.13.7 Static Library Intercepts

NovaProva also contains a number of functions which are designed to intercept and change behavior of the standard C library, usually to provide more complete and graceful detection of test failures. Some of these functions permanently replace functions in the standard C library with new versions by defining functions of the same signature and relying on link order. Some are runtime intercepts using the NovaProva intercept mechanism. Many of these functions are undocumented or platform-specific, and need to be considered when porting NovaProva.

#### __assert

```
void __assert(const char *condition, const char *filename, int lineno)
```

This function is called to handle the failure case in the standard `assert()` macro. If it's called, the calling code has decided that an unrecoverable internal error has occurred. Usually it prints a message and terminates the process in such a way that the kernel writes a coredump. NovaProva defines it's own version of this routine in `iassert.c`, which fails the running test gracefully (including a stack trace message). The function name and signature are not defined by any standard. Systems based on GNU libc also define two related functions `__assert_fail()` and `__assert_perror_fail()`.

### syslog

NovaProva catches messages sent to the system logging facility and allows test cases to assert whether specific messages have or have not been emitted. This is particularly useful for testing server code. This is done via a runtime intercept on the libc `syslog()` function. On GNU libc based systems, the system `syslog.h` header sometimes defines `syslog()` as an inline function that calls the library function `__syslog_chk()`, so that also needs to be intercepted. Similar issues may exist on other platforms.

# CHAPTER 2

# Legal Notice

# CHAPTER 3

# Indices and tables

- genindex
- search

# Index