Power Matters.™

Microsemi

SW/FW Automated Test Framework and Debug Toolkit for System Testing

Horace Chan
Brian Vandegriend

Motivation

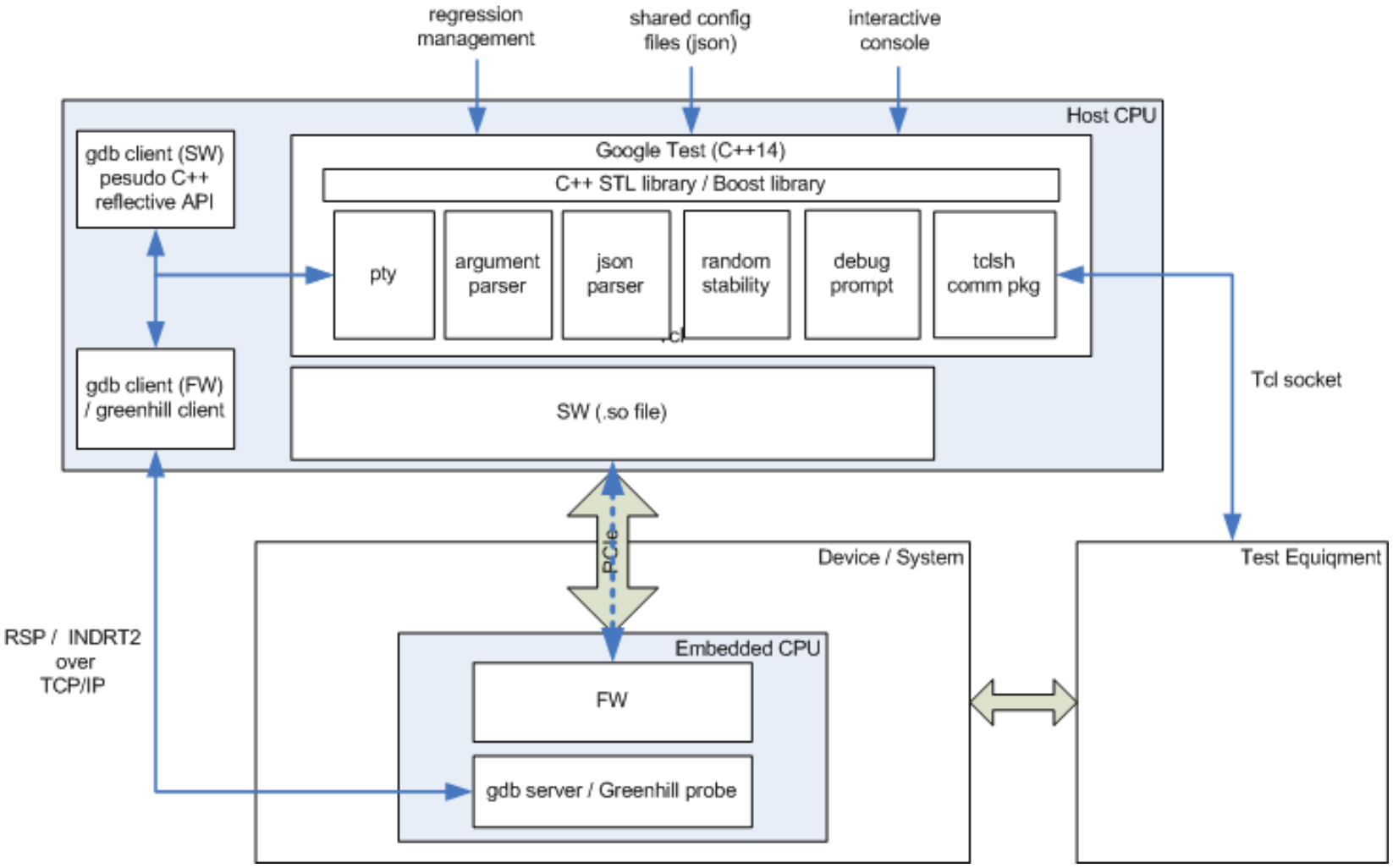
BAD: Write SW/FW system test in plain old C code

- Traditional approach to test networking SoCs
- C is **tedious** and **inefficient** for writing system tests
- Lots of **manual** test equipment configuration
- Poor system level **debug** support.
- Time wasted in **recompile** during debug

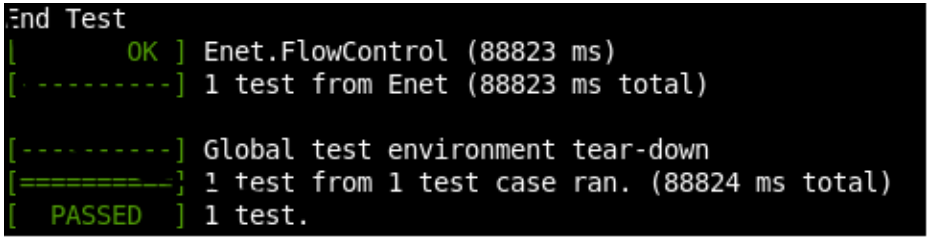
GOOD: Modernize the SW/FW system testcase

- Standardize **test framework** for automated testing
- Use powerful language syntax of **C++11/14**
- **Interactive** debug
- Leverage resources from the **open source** community
  - don’t reinvent the wheel

Main Idea



Google Test framework

BEFORE - BAD	AFTER - GOOD
• hundreds of tests binary executable	`./testsuite --test_filter="*ENET40G*" --gtest_repeat=100 --gtest_shuffle`
• copy reuse code to multiple test files • reuse code in include files and a makefile nightmare	• Object Oriented Programming (OOP) techniques • class inheritance, virtual function, dynamic cast • well defined inheritance structure in the test framework
• grep FAIL or ERROR in the log to check if the test failed • Differing styles in error messages, hard to understand why a test failed	• assertion macro output standard error message • XML report summarize the failures and errors 

C++, C++11/14, BOOST library

BEFORE - BAD	AFTER - GOOD
• C array manipulation	• C++ Standard Template Library (STL) containers
• C array for loop, while loop	• C++ STL iterator `for (auto it = vector.begin() ; it != vector.end(); ++it)`
• declare data type explicitly	• auto data type, let the compiler deduce the type
• C pointer `Segmentation fault (core dumped)`	• C++ smart pointer
• Write your own data structure or algorithms	• Use the BOOST library

Command Line Interface

BAD: ./test 1 y n 10 1 y y n y n n

GOOD: ./test –verbosity=HIGH –datapth=4 –noloopback

- Parse command line arguments with GNU C argp.h library

```
$./argex
Usage: argex [-v?V] [-a STRING1] [-b STRING2] [-o OUTFILE] [--alpha=STRING1] [-
-bravo=STRING2] [--output=OUTFILE] [--verbose] [--help] [--usage] [--version]
ARG1 ARG2


$./argex --help
Usage: argex [OPTION...] ARG1 ARG
argex -- A program to demonstrate how to code command-line options and
arguments. C
-a, --alpha=STRING1 Do something with STRING1 related to the letter A
-b, --bravo=STRING2 Do something with STRING2 related to the letter B
-o, --output=OUTFILE Output to OUTFILE instead of to standard output
-v, --verbose Produce verbose output
-?, --help Give this help list
--usage Give a short usage message
-V, --version Print program version
```

Configuration Files

BAD: hard code test config in testcase

GOOD: store test config in JSON format

```
{
    "comment" : "2xOTU4 100GE GMP",
    "testname" : "2xotu4_100ge_gmp",
    "testopts" : "",
    "dev_id" : "2x100G_LINE_ALL_10G_SERDES",
    "otn_datapaths" : [
        {
            "serdes_port" : 0,
            "otn_mode" : "OTN_SERVER_OTU4",
            "channel_rate" : 80,
            "mapotn_mode" : "MAPOTN_ODU4P_PKT_100_GE_GMP",
            "duplicate" : 1
        }
    ]
}
```

- JSON is a text file, allow edit without recompile
- Easy to read and audit
- Easy to share among multiple testcases

- Parse JSON files with Opensource Jansson C library

Tcl

BAD: Setup test equipment manually

GOOD: The testcase sets up test equipment automatically

- Remote control the test equipment with directly from the testcase with Tcl_Eval() and tcl ::comm package

```
#include <tcl.h>
#include <tk.h>
…
// instantiate tclsh inside C++
Tcl_Interp *g_tcl_interp = Tcl_CreateInterp();
Tcl_Eval(g_tcl_interp, "package require comm");
…
// Wrapper function for Tcl_Eval
char *tcl_eval(const char * format, ...)
…
// Connect to remote test equipment
tcl_eval("::comm::comm connect -port %d", port);
…
// Send tcl commands to remote test equipment
tcl_eval("::comm::comm send %d %s", port, cmd);
```

Shared Object

BAD: Check out and recompile to try different SW version

GOOD: Compile SW into shared object (.so file)

- Don’t compile the SW with the testcase
- Testcase can load different SW version at run time

```
// compile SW into shared object
$gcc –shared -fPIC –Wl,-soname,libsw.so <other gcc flags>
// copy the .so file of each version into different directory


// compile the testcase and link the SW shared object
$gcc –L/proj/swlib/latest –lsw <other gcc flags>

// set the library path to chose which SW version to test
$export LD_LIBRRARY_PATH=/proj/swlib/v5
```

NOTE: SW API must be backward compatible between versions

Interactive Debug Prompt

BAD: Edit and recompile to try new scenario in debug

GOOD: Try new scenario interactively

- Interactive debug prompt built into the testcase
- Store and recall command history with readline.h GNU C library

```
#include <readline/readline.h>
#include <readline/history.h>
…
//  register functions as debug commands, call the functions from prompt
void debug_util_s::add_command(string syntax, std::function<void()> f, string
description, int catch_all);
void debug_util_s::prompt(int debug_level, int ignore_level);
…
// registers testcase functions or lambda functions
add_command("traffic_run", debug_traffic_run, "start ilkn fpga traffic");
add_command("lamda", []() { /* do something */ }, "lamda function");
…
// call the debug prompt in the testcase with debug verbosity level
debug_util.prompt(1);
```

PTY (Pseudo Terminal) and GDB tricks

BAD: Print enum type to log as integer

Manually create enum to string conversion functions

GOOD: Use gdb as a C++ reflective API

```
#include <termios.h>
…
// create pty and run gdb in pty
fdm = posix_openpt(O_RDWR);
rc = grantpt(fdm);
rc = unlockpt(fdm);
fds = open(ptsname(fdm), O_RDWR);
char *gdb_argv[] = {"gdb", "-q", "-i", "mi", (char *)argv0.c_str(), NULL};
execv("/usr/bin/gdb", gdb_argv);
…
// enum <> string conversion function
template <typename T> T get_enum(string enum_value);
template <typename T> string get_string_from_enum(T enum_value);
// send “ptype <enum type>” to gdb, parse the name-value pairs output
```

BAD: Control FW via gdb server manually

GOOD: Testcase talks to the FW gdb server via pty

Summary / Benefits

- High productivity
  - Less lines of code (over 75% code reduction)
  - Less logical errors in the code
  - Easier code reuse with OOP
- Fully automated regression test
  - No need to baby-sit the test runs
  - Track regression status in Jenkins and Jira
  - Save 1 man-day per week
- Interactive debug toolkit
  - Faster debug turn around time
  - seconds vs minutes