In Proceedings of the Workshop on Embedded Systems Education,
Taipei, Taiwan, October, 2011.

Teaching Cross-Platform Design and Testing Methods for Embedded Systems using DICE

Shuvra S. Bhattacharyya
Dept. of ECE, and
UMIACS
University of Maryland,
College Park, USA
ssb@umd.edu

William Plishker
Dept. of ECE, and
UMIACS
University of Maryland,
College Park, USA
plishker@umd.edu

Chung-Ching Shen
Dept. of ECE, and
UMIACS
University of Maryland,
College Park, USA
ccshen@umd.edu

Ayush Gupta
Dept. of Physics
University of Maryland,
College Park, USA
ayush@umd.edu

ABSTRACT

DICE (the *DSPCAD Integrative Command Line Environment*) is a package of utilities that facilitates efficient management of software projects. Key areas of emphasis in DICE are cross-platform operation, support for projects that integrate heterogeneous programming languages, and support for applying and integrating different kinds of design and testing methodologies. The package is being developed at the University of Maryland to facilitate the research and teaching of methods for implementation, testing, evolution, and revision of engineering software.

The platform- and language-independent focus of DICE makes it an effective vehicle for teaching high-productivity, high-reliability methods for design and implementation of embedded systems for a variety of courses. In this paper, we provide an overview of features of DICE — particularly as they relate to *testing driven design practices* — that are useful in embedded systems education, and discuss examples and experiences of applying the tool in courses at the University of Maryland aimed at diverse groups of students — undergraduate programming concepts for engineers, graduate VLSI architectures (aimed at research-oriented students), and graduate FPGA system design (aimed at professional Master's students).

1. INTRODUCTION

In this paper, we present a motivation for deep integration into embedded systems education of testing-driven design and methods for cross-platform and language-independent test suite development. To support these objectives, we introduce instructional applications of DICE (the *DSPCAD Integrative Command Line Environment*), which is a package of utilities that facilitates efficient management of software projects [2]. Key areas of emphasis in DICE are cross-platform operation, support for projects that integrate heterogeneous programming languages, and support for applying and integrating different kinds of design and testing methodologies. The package is being developed at the University of Maryland to facilitate the research and teaching of methods for implementation, testing, evolution, and revision of engineering software. The package is also being developed as a foundation for developing experimental research software for techniques and tools in the area of computer-aided design (CAD) of digital signal processing (DSP) systems.

The platform- and language-independent focus of DICE makes it an effective vehicle for teaching high-productivity, high-reliability methods for design and implementation embedded systems. The platform- and language-independent aspects are especially useful for embedded systems, where a wide variety of platforms and programming languages are relevant (e.g., MATLAB, C, C++, CUDA, Verilog, and VHDL are some of the widely used languages), and heterogeneous languages are often involved in the same project — both at different stages of the design flow (e.g., simulation vs, implementation) as well as at horizontally, at the same level (e.g., cooperating hardware description language and embedded C code for hardware/software implementation). Furthermore, the lightweight conventions of DICE are useful in instructional settings to help focus attention on core design and testing principles, practices, and methodologies, rather than idiosyncrasies of specialized tools.

Using DICE, instructors can incorporate systematic, command line integration of arbitrary collections of design tools; expose students to rigorous testing-driven design practices; and retarget project modules to different languages and platforms (e.g., for migration from software to hardware, or from functional prototypes to optimized implementations on specialized engines for signal processing, graphics processing, etc.).

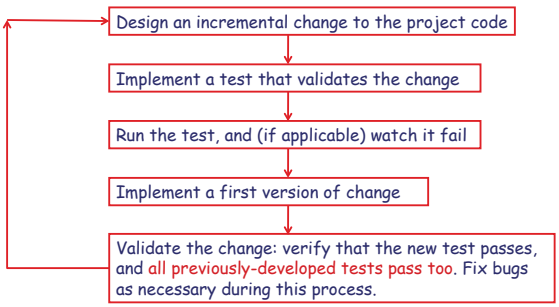

Figure 1: An illustration of a design flow for testing-driven software development.

The DICE package and related instructional modules that we have been developing at the University of Maryland (UMD) support the testing driven design flow illustrated in Figure 1. When teaching testing-driven design concepts, we introduce this design flow early in the course, and emphasize and assess its use throughout the semester. One method that we have found useful in assessing this use is to ask students to keep track of and document their incremental design process as they work on a project. We find this useful as it helps reinforce the importance of testing-driven design, and also helps us to assess the students' application of the associated methods.

Key concepts behind the design flow of Figure 1 are the development of code in incremental units or "steps", where each step is accompanied by the addition of one or more new tests that exercise the step, along with validation of any new tests along with all previously developed tests (i.e., validation against the entire test suite). This helps to give confidence that the new code works and that previously developed code has not been "broken" before proceeding to a new code development step. This *testing driven development methodology* is widely regarded as improving software productivity and reliability; however, instructional approaches to programming often focus more on a code-driven approach, where testing is treated more as an afterthought than as an integral part of the development process. For more details on testing-driven development practices, there are various useful references (e.g., see [9, 12, 21]). Lecture notes that elaborate on testing driven design concepts and provide an integrated overview of DICE are available on the online supplement to the DICE User's Guide (see [3]).

When applying testing-driven and cross-platform design practices, it is easy for students and instructors within instructional environments to get "lost" in the idiosyncrasies of various specialized tools (e.g., specialized unit testing environments, and platform-specific development environments, as well as specialized UNIX-based utilities such as `make` and different scripting environments). It is important that students understand distinctions between design *methods* and *tools* that assist in applying those methods. Through careful application of the lightweight conventions and interfaces in DICE, our instructional approaches are geared towards making these distinctions more clear. At the same time, DICE is designed for integration with arbitrary collections of specialized tools so that designers can work with the tools that are most relevant for their projects or their targeted educational experiences.

2. NOTATIONAL CONVENTIONS

In-line code fragments within sentences are shown using `this font`. In-line code fragments can be hyphenated across line boundaries, so care should be taken to "filter out" any line-ending hyphens when applying such fragments.

Line-by-line code fragments are shown

```
using one or more lines that have
this font.
```

A backslash at the end of a code fragment line indicates that the text on the following line is a continuation of the command on the previous line. For example:

```
gcc -Wall -pedantic -o x.exe \
        a.c b.c c.c d.c e.c f.c
```

For a command that spans multiple lines, we typically indent the code on the continuation lines relative to the code on the first line of the command, as shown above.

3. TEST SUITE CONSTRUCTION

To promote compatibility with a variety of platforms, the DICE engine consists of a collection of utilities implemented as Bash scripts, C programs, and Python scripts. Because DICE is developed based on these open-source, command line interfaces and languages, the package operates on and can easily be set up for different platforms, including Linux, OS-X, Solaris, and Windows (equipped with Cygwin). This gives DICE a wide base from which to integrate specific design flows. From this base, we have architected a testing approach that is applicable across design flows.

To implement a unit testing suite, the developer provides a test reference for the functional behavior of the *module under test* (*MUT*). This reference consists in general of a set of inputs and the set of outputs that are produced as a result of correct processing of those inputs. Unit testing is performed by executing the MUT and comparing the actual output with the correct expected behavior. The event of a module handling an anticipated error scenario can be added to the test reference as well — e.g., for use when a designer is trying to verify that when an input condition is violated, the application detects the error and reports an appropriate diagnostic message.

The basic unit of the DICE unit testing framework is a directory referred to as an *individual test subdirectory* (*ITS*). A test suite consists of any number of ITSs that test the different behaviors of the MUT. Every ITS name must start with the prefix "test" (e.g., `test01`, `test02`, `test-square-matrix-1`, `test-square-matrix-2`, etc.). Based on this convention, changing the ITS prefix to any word other than "test" will effectively exclude it from the test suite.

An ITS consists of the following required files:

- A `README.txt` file that contains an explanation of what part of the MUT functionality this ITS tests.
- A `makeme` script that contains all compilation steps required before running the test. It is important to note that the `makeme` script does not compile the source code of the MUT, rather it compiles any additional code required for the test — e.g., a *driver* (diagnostic) program that supplies the MUT with inputs and prints its outputs.
- A `runme` script that runs the test. The contents of the `runme` script may vary depending on the type of the MUT. For example, when testing a C program, one may need to just execute an object file, but for a Verilog module, a hardware simulator such as ModelSim may need to be invoked to exercise the MUT. Also the `runme` script may contain a call to other executables that perform different post processing functions on the MUT output before it is compared with the associated `correct-output.txt` and `expected-errors.txt` files, which are described next.
- A `correct-output.txt` file that contains the correct standard output that has to be produced by the test (i.e, after executing the `runme` file).
- An `expected-errors.txt` file that contains the error messages that the test is expected to produce on the standard error output. This file is useful when the ITS checks for the errors that the MUT has to catch. If no errors are expected, then the file needs to be present in the ITS as an empty file.

Students who are familiar with UNIX sometimes question the need for `makeme` scripts when the UNIX utility `make` already exists. Indeed, our use of `makeme` scripts is largely orthogonal to the `make` utility — a `makeme` script can include a call to `make` (e.g., if one wants to use features of make such as dependency tracking and update-driven compilation). Such orthogonality highlights the utility of DICE and its lightweight conventions to help distinguish between specific tools for project development (e.g., `make`), and structured interfaces, such as the use of `makeme` and `runme` files in DICE, that facilitate systematic application of higher level design methodologies, such as testing-driven and cross-platform design.

The basic DICE utility that makes use of the required files and exercises test suites is called `dxtest`. When `dxtest` is executed from a certain directory d, it recursively traverses all subdirectories of d that begin with the prefix "test", executes all of the ITSs that it finds throughout this recursive traversal, and summarizes the results, including the number of test failures encountered and the specific directory locations for each failed test.

During the recursive traversal of `dxtest`, any subdirectory that contains a `runme` file is considered as an ITS. When `dxtest` visits an ITS, it first executes the `makeme` script, followed by the `runme` script. It then compares the actual output generated after running `runme` with the contents of the `correct-output.txt` file, and the actual standard error output with the contents of the `expected-errors.txt` file.

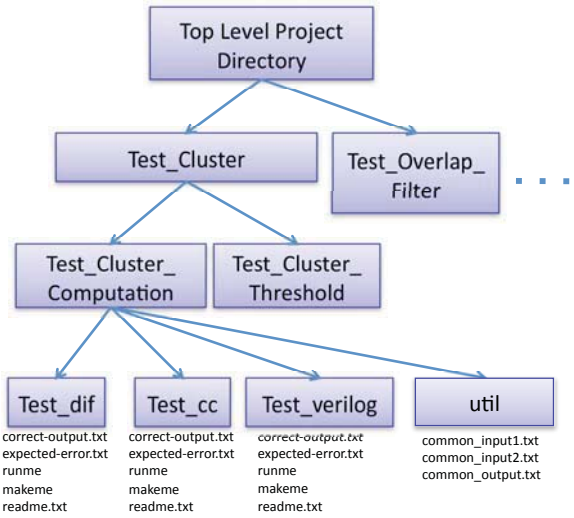

Figure 2: The file and directory structure of a cross-platform project using the DICE testing approach.

After traversing all subdirectories, a summary of successful and failed tests is produced, as described above.

In the example in Figure 2, there are three implementations under test for the same functional module: these are written in the dataflow interchange format (DIF) language for formal modeling [11], C++ for software implementation, and Verilog for hardware implementation. The input and output patterns are common to each of these tests and reside in a `util` directory that is common to all of the tests. While the implementation in each language has customized build and simulation scripts in `makeme` and `runme` files, and `correct-output.txt` and `expected-errors.txt` files that are tailored to their simulation environments, the fundamental inputs and outputs are directly shared between these platforms.

By using such a framework that automates the process of test verification, any change to the basic MUT can be verified not only for the new functional correctness, but also to ensure that it does not ruin a previous correct behavior. This also facilitates incremental code development, and reduces development and verification time by allowing bugs to be discovered earlier, before they get entangled with other bugs, and become more difficult to diagnose and fix.

4. RELATIONSHIP TO OTHER TESTING FRAMEWORKS

Typically, when software designers employ unit testing, they use frameworks that are language specific [9, 25]. More than just syntactic customizations, such frameworks are often tied to fundamental constructs of the language. For example, in CppUnit a unit test inherits from a base class defined by CppUnit [9]. A test writer then overloads various methods of the base class to put the specific unit test in this framework. Tests requiring the specific features that leverage the constructs of a language (e.g., in an object oriented language, checking that the method exhibits the proper form of polymorphism) are well served by these approaches. Furthermore, these language-specific approaches

work well when designers are using only a single language or a single platform for their final implementation. But when designers must move between languages with different constructs (such as between C++ and Verilog), the existing tests must be rewritten. This requires extra design effort and creates a new verification challenge to ensure that unit tests between the two languages are in fact performing the same test.

Embedded and high performance software must often utilize multiple languages and multiple platforms (e.g., see [5]), and transcoding between an initial application specification and software for the final implementation. DICE is language-agnostic to support this design need. By simplifying and streamlining the processes of testbench design and implementation, the same test fixture can be used in a variety of scenarios. DICE encourages that tests be written in a language-agnostic way, prompting designers to provide input and expected output streams using primitive data types. DICE tests are simpler to write (even non-language-experts can write them), easier to maintain, and much more portable.

Perhaps the most related framework to DICE is the Test Anything Protocol (TAP) [4]. TAP is language-agnostic by defining the protocol that manages the communication between unit tests and a testing harness. Individual tests (TAP producers) communicate test results to the testing harness (TAP consumers). TAP enables multi-platform, multi-language design, but only at the communication boundary. Unit tests need only adhere to the communication design, leaving test writers with no specific language-agnostic mechanism for writing the tests themselves. Indeed, many language specific unit tests have TAP compatible outputs so they may be hooked into a larger multi-language testing environment.

Note that the testing features provided in DICE are oriented towards test implementation, test execution, and general practices of test-driven project development — they are not developed as a framework oriented towards any particular methodology for test design or test generation, such as those discussed in the extensive survey by Hierons et al. [10]. DICE allows one to apply different methods for test suite design, while providing features of cross-platform operation; cross-language testing; efficient test retargetability; automated test suite execution and test status reporting; and seamless integration as part of the overall feature set of DICE, which provides a variety of other lightweight utilities for efficient cross-platform project development [2]. Exploring ways to integrate DICE-based project and test development with systematic approaches to test design and generation is a useful direction for further study.

5. INSTRUCTIONAL EVOLUTION AND CASE STUDIES

5.1 Programming Concepts for Engineers

In Spring 2007 at UMD, we initiated a course entitled *Intermediate Programming Concepts for Engineers* (*IPCE*) , which was intended to strengthen the exposure to programming concepts for Electrical Engineering (EE) majors, who, unlike our Computer Engineering majors in the same Department (ECE), have relatively few courses dedicated to programming and software development. The course was given in pilot form in Spring 2007 and Spring 2008, and then became a requirement for EE majors from Fall 2008. It is from the pilot version of this course that we began integrating testing-driven design practices in the curriculum — indeed, incorporation of concepts and methods of testing-driven design was one of the key aspects in the original syllabus for the course.

In initial offerings of the new programming concepts course, we introduced testing methods after approximately 5 weeks of lectures and initial assignments on programming techniques. We then switched to a format where the importance of integrated testing methods, and incremental design and testing approaches were emphasized from the beginning of the course, and initial programming assignments focused on writing tests rather than exercising new programming features or challenging students in terms of design complexity. Based on student feedback and our impressions about the rigor and consistency with which students were applying testing-driven design practices, we retained the latter model, where testing is covered and experience with testing and DICE is acquired by students from the beginning of the course.

The cross-platform aspect of DICE is not exercised directly in our IPCE course; however, we have found that the language-independent test suite structure provided by DICE, as described in Section 3, allows instructors to cleanly and naturally separate testing concepts and methods from their language-specific implementations. In our presentation of testing methods, we emphasized this separation explicitly during lectures, and demonstrated these concepts through the lightweight conventions of design — i.e., the basic requirements for a test can be encapsulated within readily retargetable `makeme`, and `runme` files, along with text files (`correct-output.txt` and `expected-errors.txt`) that provide inputs and establish "golden outputs" for the tests.

Our experience developing and delivering the IPCE course established a foundation for our instructional approaches to testing-driven design concepts and application of DICE for exercising these concepts. Students learning these concepts go on to learn different domain-specific languages (e.g., MATLAB, LabVIEW and Simulink) as they study EE topics in signal processing and control, as well as the Verilog HDL for digital system design. We anticipate that the emphasis in IPCE on testing and testing-driven design as general design methodologies demonstrated and visualized through an easily retargetable processes allows students to more easily connect testing concepts to other languages and design environments as they encounter them in other EE courses related to embedded system design.

The instructional foundation established in our work on the IPCE course is also useful in the development of graduate courses that are relevant to embedded systems. For example, it provides a testing- and cross-platform-design-oriented complement to available books in embedded systems, advanced digital system design, and cyber-physical systems (e.g., see [19, 20, 24, 26, 27]). Building on our experience in the development of the IPCE course, we have developed two

graduate courses at the University of Maryland that rigorously incorporate principles and practices of testing-driven design and cross-platform methodologies into relevant contexts in advanced embedded systems education. We discuss these courses next in Section 5.2 and Section 5.3.

5.2 VLSI Architecture

In Spring 2010, we delivered *VLSI Architecture* as a graduate course. The course had been developed and delivered previously, and we delivered it in Spring 2010 with a revised syllabus that incorporated a new focus on hands-on design experiences. Since designs for modern digital systems are increasingly based on customized programmable platforms such as digital signal processors, graphics processors (GPUs), and field programmable gate arrays (FPGAs), there is a variety of programming approaches, which presents developers with new, often difficult design decisions. For example, GPU implementations require applications to be described as sets of lightweight threads (e.g., see [13]).

Our revised VLSI Architecture course covers design flows and tools available for arriving at efficient implementations on customized programmable platforms, along with covering research related to new programming models for multicore platforms and advances in the simulation and synthesis capabilities provided by electronic design automation (EDA) tools. Reinforcement of the diversity of established and emerging platforms, and their underlying trade-offs is a key consideration throughout the course.

While the overall goal of the course is to present common challenges and solutions to utilizing platform specific programming approaches, along with coverage of their underlying VLSI architectures, it is also an implementation focused course and students code, build, and debug implementations on two such platforms — an FPGA and a CUDA-enabled GPU.

DICE facilitated this course by providing a single umbrella under which other design environments were used, giving students a more unified programing experience than a conventional approach would provide. For example, consider the `makeme` file shown in Program 1. This script compiles a *golden model* (the functional prototype reference from which specialized platform-based implementations are to be derived) of a Gaussian filter (`gfilter`) design project, where the golden model is to be developed in C. Development of this `makeme` script requires students to build a driver file for a test, and link it against a library of object files.

Program 1 An example of a `makeme` script for a C-based Gaussian filter project.

```
gcc -c -Wall -pedantic -I$UXSRC driver.c
gcc -Wall -pg -pedantic -o driver driver.o \
        $UXBIN/bmp_file_write.o \
        $UXBIN/gfilter.o \
        ...
```

Here, `gcc` is used to invoke the GNU C compiler with a selection of compiler options, and `bmp_file_write.o` and `gfilter.o` are a few selected object files taken from a designated object file directory `$UXBIN`. These files, respectively, provide code for writing the image output as a bitmap file and performing the actual Gaussian filtering on the image. Other object files include reading bitmap images and applying other image processing kernels, all of which may be linked into the driver with this `makeme` script. These object files are provided to the students as part of the instructional environment so that they can build on them to develop a full Gaussian filtering application for bitmap images.

A corresponding `runme` script for an ITS based on the `makeme` script shown above is:

```
./driver coeffs.txt ../util/lena512.bmp out.bmp > \
        diagnostics.txt
```

This script exercises the driver program with a specific set of Gaussian filter coefficients (taken from the file `coeffs.txt`), and a specific bitmap image (taken from the file `lena512.bmp`), and stores the resulting image in the file `out.bmp`. Diagnostic text output generated by the driver program to the display is redirected by the `runme` script to the file `diagnostics.txt`.

In a later portion of the course, students return to the same example to utilize GPU acceleration with CUDA. For this purpose, the build process in DICE is altered slightly to accommodate CUDA, by using NVIDIA's CUDA compiler, `nvcc`:

```
nvcc -I$UXSRC driver.c -o driver.o
nvcc -o driver \
        $UXBIN/driver.o $UXBIN/gfilter_cuda.o \
        ...
```

This `makeme` script for a CUDA-accelerated driver is similar to the corresponding C-based `makeme` script shown earlier, except that instead of using the GNU C Compiler (`gcc`), the script now invokes NVIDIA's CUDA Compiler (`nvcc`) to create and link the object files. Before applying this ITS `makeme` script, both GPU accelerated and unaccelerated actors are compiled into object files by `nvcc` — as part of the project development process — and placed in the same object file directory `$UXBIN`. Here, the driver is compiled with header files in `$UXSRC` and linked against the library of object files to create the executable to be run. One of the object files linked (`gfilter_cuda.o`) is a CUDA accelerated instance of the Gaussian filtering kernel.

In this version of the project, the method of compiling has changed from its unaccelerated C version, but the overall project interfaces did not change. Thus, the overall functionality is the same as the golden model — a Gaussian filtering application for bitmap images, and input, `correct-output.txt`, and `expected-error.txt` files used to validate the golden model can be reused to validate the CUDA-accelerated implementation.

Furthermore, by adhering to the same interface as the C example, there is no change needed to the `runme` script. By

virtue of respecting DICE conventions, there is no change needed to the testbench, including input files, `correct-output.txt`, and `expected-errors.txt`, as described above. The ability to reuse the testbench allows students to focus on fundamental CUDA programming issues while retaining the ability to easily run and test their own code versus a reference example, and concretely reinforcing the value of golden models for functional validation.

5.3 Design and Implementation of Digital Systems

In Spring 2010, we offered a new course entitled *Design and Implementation of Digital Systems* (*DIDS*) for professional Master's students who were registered in the UMD Professional Master of Engineering (ENPM) Program. The objective of this course is to introduce students to efficient verification and test planning as well as advanced HDL-based design methods for embedded systems design targeted to FPGAs.

Design methodology for embedded systems targeted on FPGAs is based heavily on use of HDLs, such as Verilog and VHDL, and use of automated synthesis from HDL programs into final implementations (e.g., see [27]). This trend towards HDL-based FPGA system design has been driven by the complexity of modern digital integrated circuits, and advances in the simulation and synthesis capabilities provided by electronic design automation (EDA) tools. Like the VLSI Architecture course discussed in Section 5.2, this course is designed to be project-oriented. Additionally, the course covers in more depth the design and implementation of digital systems using the Verilog HDL.

In the initial segment of this course, we introduce testing-driven design concepts and the DICE framework as a concrete environment for learning and experimenting with these concepts. We also emphasize the importance of cross-platform design methodologies in FPGA system design, as modern FPGA design flows employ a variety of languages — e.g., C for rapid prototyping, functional validation, and embedded control; Python for data format manipulations and other scripting operations; and Verilog and VHDL for the core digital processing functionality.

In order to help students understand systematic practices for testing of embedded systems, we first assign a C programming project that is based on an embedded audio processing application for 44.1 kHz to 48 kHz sampling rate conversion — i.e., for conversion between compact disc and digital audio tape formats. For this project, students use the lightweight dataflow (LWDF) programming approach. LWDF provides an abstract application programming interface (API) for model based design and implementation of signal processing systems that can be targeted to arbitrary simulation- and platform-oriented languages, such as C, CUDA, MATLAB, Verilog, and VHDL [23, 22]. LWDF is not part of DICE — it is a separate package that provides complementary features to DICE for the domain of signal processing systems. This first project in our DIDS course guides students in developing a practical embedded system using the DICE environment, where students follow the approaches explained in Section 3 to construct test suites for testing individual components and the overall application.

After introducing testing-integrated design practices using C and DICE, our coverage moves to the Verilog language, and to concepts for design and synthesis of digital hardware structures for efficient FPGA implementation. We assign three projects that are based on the development of different types of embedded signal processing subsystems, and are structured in terms of interacting LWDF components using the Verilog HDL. These projects are 1) a fixed-point digital filter (`fir`); 2) a first-in first-out (FIFO) buffer, which is a fundamental component for providing a concrete realization of the LWDF API; 3) and a polynomial evaluation accelerator (`pea`).

To use DICE for Verilog-based projects in our DIDS course, students need to customize their `makeme` scripts in order to compile Verilog source code, including source code for application components and testbenches for driving simulations of the applications. We use Mentor Graphics ModelSim as the core tool for verification and simulation. In the DICE-based HDL design environment for the course, `makeme` files effectively serve as standard wrappers for applying ModelSim in conjunction with course conventions for decomposing projects into modules and organizing project modules and test structures. These files help to provide a uniform design process, and also reinforce the general role of the embedded commands (in this case, calls to ModelSim, post-processing of output results, etc.) in the context of the overall design process.

Example code for a `makeme` file that compiles Verilog source code for application components is shown in Program 2. Here, the `vlib` command creates a working library for the targeted design, the `vmap` command defines a mapping between a logical library name and a working directory, and the `vlog` command compiles the Verilog source code into a specified working directory.

Program 2 An example of a `makeme` script for a Verilog project.

```
function makeone {
    if ! [ -d ../bin ]; then
        mkdir ../bin
    fi
    vlib ../bin
    vmap work ../bin
    vlog -work ../bin \$1.v
}

makeone fifo
makeone fir
makeone pea
```

Similarly, example code for a `makeme` file that is customized to compile Verilog source code for an HDL driver subsystem (e.g., a testbench) is shown in Program 3. Such a `makeme` script can be used as part of one or more ITSs in a project test suite.

To simulate the behavior of an application, and validate the results using DICE, students need to make appropriate customizations to their `runme` files. An example of a `runme` file that simulates an HDL implementation and redirects the

Program 3 An example of a `makeme` script for a Verilog testbench.

```
if ! [ -d work ]; then
    vlib work
    vmap work work
fi

vlog testbench.v
```

results to standard output is shown in Program 4. Here, the `vsim` command invokes the ModelSim simulator on the compiled design library.

Program 4 An example of a `runme` script for a DICE ITS associated with a Verilog project.

```
UXLIB=../../../bin

if [ -f out.txt ]; then
    rm -r out.txt
fi

vsim -c -L \$UXLIB -do "run -all" testbench > \
        transcript
cat out.txt 2> err.txt
```

5.4 Case Studies Summary

In summary, we have developed three significantly different courses related to embedded system design, and in these courses we have extensively applied our proposed instructional methods for testing-driven and cross-platform project development. These courses target diverse groups of students — from first- and second-year undergraduate students to professional masters students to research-oriented masters and Ph.D. students. DICE provides a lightweight environment for the courses that facilitates integration of different software tools in a manner that promotes testing-driven and cross-platform design practices. Building on the instructional foundations developed in these regular University of Maryland courses, we have also applied DICE in intensive short courses on embedded signal processing systems for industry, and also for summer schools targeted to Ph.D. students and early-career industry researchers. Key themes in our development of instructional modules based on DICE have been teaching students about testing-driven design, the importance of experimenting with different kinds of platforms and programming languages, and distinctions between design *processes* and specialized *tools* that are used to implement those processes.

6. OTHER APPROACHES IN TESTING-INTEGRATED PROGRAMMING EDUCATION

Testing is an aspect of programming that often receives relatively little attention in programming courses, even though it is a crucial aspect of almost all stages of professional programming. This creates a vast gap between practical embedded systems programming and students' classroom experiences in programming courses. Attending to this issue, some researches and curriculum developers have started to introduce test-driven development in programming and software engineering courses (e.g., see [1, 7, 8, 16]).

Some researchers have introduced testing in advanced courses on software engineering or courses that focus on development of large-scale projects [1, 15]. However, when testing is taught to students in a separate software-engineering course, it can end up being disconnected from their other programming experiences, and fail to become an integrated part of students' regular programming practices [7, 15, 6, 14]. Thus, there is a need to introduce testing early in the sequence of programming courses, and integrate and continue reinforcing it across all programming courses, rather than delegating it to a single course. The DICE framework lends itself particularly well to integration across a a variety of programming-related courses, as discussed in Section 5.

The effectiveness of a test-driven approach to teaching programming concepts can be hindered by limitations in students' fluency with programming. The DICE framework, which is language agnostic, and not encumbered by the syntactic regulations of a single language, simplifies and streamlines the process of software/code testing, making it much easier for students to learn testing at the same time as they are applying basic, intermediate or advanced programming skills.

The language-agnostic feature of DICE provides another advantage over other testing systems used for instruction, such as JUnit [12], TestLab [14], and BlueJ [18, 17]: since DICE is not tied to any particular language, students can continue to use the same testing framework over an entire sequence of programming courses — allowing for exposure to different programming languages — from introductory programming courses to advanced design courses — without the distraction of having to learn different testing environments. This can help reinforce the role of testing-driven design practices as common theme in complex embedded systems, while keeping the curriculum streamlined and focused on targeted languages and design methods.

7. SUMMARY AND FUTURE DIRECTIONS

In this paper, we have motivated the deep integration into embedded systems education of testing-driven design and methods for cross-platform and language-independent, test suite development. We have also introduced a software package called DICE (the *DSPCAD Integrative Command Line Environment*), which is a package of utilities that facilitates efficient management of software projects, and facilitates teaching of methods in testing-driven design, and cross-platform project development. DICE is an open source project under active development that is publicly available.

While we have found that DICE allows us to integrate testing-driven design methods easily into a variety of courses, using different design languages and target platforms, systematic study involving the impact on student learning is needed to thoroughly assess the instructional utility of our proposed methods and to help guide their further development. Thus, important directions for future work are motivated by the need for systematic research on the instructional effectiveness of the test-driven approach using the DICE framework

across the various courses, as well as basic research on differences in how students approach embedded systems programming when taught using traditional instruction versus a testing-driven approach. The methods and experiences reported on in this paper contribute foundations that help to address these important and challenging issues in embedded systems education.

8. ACKNOWLEDGMENTS

This work was supported in part by the US National Science Foundation.

9. REFERENCES

[1] E. Allen, R. Cartwright, and C. Reis. Production programming in the classroom. In *Proceedings of the SIGCSE Technical Symposium on Computer Science Education*, February 2003.

[2] S. S. Bhattacharyya, W. Plishker, C. Shen, N. Sane, and G. Zaki. The DSPCAD integrative command line environment: Introduction to DICE version 1.1. Technical Report UMIACS-TR-2011-10, Institute for Advanced Computer Studies, University of Maryland at College Park, 2011. http://drum.lib.umd.edu/handle/1903/11422.

[3] S. S. Bhattacharyya, C. Shen, W. Plishker, N. Sane, and G. Zaki. Using the DSPCAD integrative command-line environment: User's guide for DICE version 1.1. Technical Report UMIACS-TR-2011-13, Institute for Advanced Computer Studies, University of Maryland at College Park, 2011. http://hdl.handle.net/1903/11804.

[4] S. Cozens. *Advanced perl programming*. O'Reilly & Associates, Inc., second edition, 2005.

[5] S. A. Edwards. *Languages for Digital Embedded Systems*. Kluwer Academic Publishers, 2000.

[6] S. H. Edwards. Improving student performance by evaluating how well students test their own programs. *Journal on Educational Resources in Computing*, 3(3), September 2003.

[7] S. H. Edwards. Rethinking computer science education from a test-first perspective. In *Proceedings of the Conference on Object Oriented Programming, Systems, Languages, and Applications*, October 2003.

[8] M. H. Goldwasser. A gimmick to integrate software testing throughout the curriculum. In *Proceedings of the SIGCSE Technical Symposium on Computer Science Education*, February 2002.

[9] P. Hamill. *Unit Test Frameworks*. O'Reilly & Associates, Inc., 2004.

[10] R. M. Hierons et al. Using formal specifications to support testing. *ACM Computing Surveys*, 41(2), February 2009.

[11] C. Hsu, M. Ko, and S. S. Bhattacharyya. Software synthesis from the dataflow interchange format. In *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*, pages 37–49, Dallas, Texas, September 2005.

[12] A. Hunt and D. Thomas. *Pragmatic Unit Testing in Java with JUnit*. The Pragmatic Programmers, 2003.

[13] W. W. Hwu. *GPU Computing Gems*. Morgan Kaufmann Publishers Inc., 2011.

[14] E. L. Jones. Software testing in the computer science curriculum — a holistic approach. In *Proceedings of the Australasian Conference on Computing Education*, 2000.

[15] E. L. Jones. Integrating testing into the curriculum — arsenic in small doses. In *Proceedings of the SIGCSE Technical Symposium on Computer Science Education*, February 2001.

[16] E. L. Jones and C. L. Chatmon. A perspective on teaching software testing. *Journal of Computing Sciences in Colleges*, 16(3), 2001.

[17] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg. The BlueJ system and its pedagogy. *Computer Science Education*, 13(4):249–268, 2003.

[18] M. Kölling and J. Rosenberg. Guidelines for teaching object orientation with Java. In *Proceedings of the Annual Conference on Innovation and Technology in Computer Science Education*, 2001.

[19] E. A. Lee and S. A. Seshia. *Introduction to Embedded Systems, A Cyber-Physical Systems Approach*. 2011. `http://LeeSeshia.org`, ISBN 978-0-557-70857-4.

[20] P. Marwedel. *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems*. Springer, 2010.

[21] J. Reekie, S. Neuendorffer, C. Hylands, and E. A. Lee. Software practice in the Ptolemy project. Technical Report GSRC-TR-1999-01, Gigascale Semiconductor Research Center, University of California at Berkeley, April 1999.

[22] C. Shen, W. Plishker, and S. S. Bhattacharyya. Dataflow-based design and implementation of image processing applications. Technical Report UMIACS-TR-2011-11, Institute for Advanced Computer Studies, University of Maryland at College Park, 2011. http://drum.lib.umd.edu/handle/1903/11403.

[23] C. Shen, W. Plishker, H. Wu, and S. S. Bhattacharyya. A lightweight dataflow approach for design and implementation of SDR systems. In *Proceedings of the Wireless Innovation Conference and Product Exposition*, pages 640–645, Washington DC, USA, November 2010.

[24] S. Sriram and S. S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. CRC Press, second edition, 2009.

[25] H. G. T. Dohmke. Test-driven development of a PID controller. *IEEE Software*, 24(3):44–50, 2007.

[26] W. Wolf. *Computers as Components: Principles of Embedded Computer Systems Design*. Morgan Kaufmann Publishers Inc., 2000.

[27] W. Wolf. *FPGA-Based System Design*. Prentice Hall, 2004.