The DSPCAD Integrative Command Line Environment: Introduction to DICE Version 1.1

Shuvra S. Bhattacharyya, William Plishker, Chung-Ching Shen,
Nimish Sane, and George Zaki
Department of Electrical and Computer Engineering, and
Institute for Advanced Computer Studies
University of Maryland at College Park, USA
{ssb, plishker, ccshen, nsane, gzaki}@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 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 package is intended for cross-platform operation, and is currently being developed and used actively on the Linux, Mac OS, Solaris, and Windows (equipped with Cygwin) platforms.**

This report provides an introduction to DICE, and provides background on some of the key features in DICE Version 1.1. This report also gives a brief introduction to `dicelang`, which is a plug-in package for DICE that provides additional utilities, libraries, and tools for managing software projects in specific programming languages.

I. INTRODUCTION

This report provides an introduction to DICE, which stands for the *DSPCAD Integrative Command Line Environment*, and accompanies the latest release of DICE, which is Version 1.1. The objective of DICE is to provide a flexible, lightweight environment for the research, development, testing, and integration of software projects, particularly those that employ heterogeneous programming languages or models of computation. DICE is being developed by the Maryland DSPCAD Research Group, which focuses on computer-aided design (CAD) techniques for digital signal processing (DSP) systems. However, the features provided in DICE are generally not specific to DSP or CAD-for-DSP applications, and can be applied in other domains.

For clarity, we should note what DICE is *not*. DICE is not meant to replace existing software development tools. DICE is not a shell nor is it a compiler or synthesizer. It is not a debugger nor does it provide simulation capabilities. It does not provide automatic transcoding for porting between platforms and languages. DICE is instead a command line solution to utilize all of these existing kinds of tools more effectively, especially for cross-platform design.

II. SETTING UP DICE

DICE is implemented as a collection of utilities that are in the form of Bash scripts, C programs, and Python scripts. Therefore, facilities for interpreting/compiling these languages must be available to use all of the capabilities in DICE. DICE is developed with significant attention to cross-platform operation. Platforms on which DICE is used actively include Linux, Mac OS, Solaris, and Windows (equipped with Cygwin).

DICE can be downloaded from the DICE Project Website [1]. Details on setting up, using, and troubleshooting DICE can also be found through this website.

III. INTRODUCTION TO DICE UTILITIES

DICE includes a variety of utilities to help improve productivity while working in a command-line or shell-

Technical Report UMIACS-TR-2011-10, Institute for Advanced Computer Studies, University of Maryland at College Park, 2011.

based project development environment. This section provides a brief introduction to some of the basic utilities available in DICE. These utilities help improve the convenience with which one can do some common and fundamental tasks.

A. DIRECTORY NAVIGATION

For users of command line based development environments, directory navigation can be cumbersome and time consuming when done many times a day. To alleviate this, DICE provides a number of utilities for efficient navigation through directories. This group of utilities allows one to label directories with arbitrary, user-defined identifiers, and to move to (i.e., `cd` to) directories by simply referencing these identifiers. This is a much more convenient way of "jumping" from one directory to another compared to typing the complete directory path or explicitly changing directories through the relative path of the desired destination directory.

The primary DICE utility related to directory navigation is `dlk`, which stands for the "*d*irectory *l*in*k*ing utility". The following is the general usage format for the `dlk` command.

```
dlk <label>
```

Here, `<label>` is the string that is to be associated as the label for the current working directory. Such a label can be of arbitrary length, but should contain only alphanumeric characters (e.g., no spaces).

Once one runs the `dlk` command in a specific directory, the user can return to same directory at any future time (during the same shell session or a subsequent session) by running the DICE `g` command.

The command name `g` is derived from the word "*g*o". The general usage format for the `g` command is as follows.

```
g <label>
```

Here, `<label>` is a label that has been associated with a directory through prior use of the `dlk` command. Running the `g` command allows one to `cd` (change directory) to the directory that is currently associated with the given label.

As a simple example, consider the following sequence of commands, and assume that the directory paths referenced in the commands are valid.

```
cd ~/projects/proj1
dlk p1
cd ~/documents/doc1
g p1
```

After the above sequence of commands, the user will end up in `~/projects/proj1`.

If the `dlk` command is called with a label that is already associated with a different directory, then the previous association is silently overwritten, and the association is changed so that the label is linked to the current working directory.

The associations used by DICE between directory labels and absolute paths are maintained in a subdirectory called `g` in the `dice_user` directory. The `dice_user` directory is a directory in which user-specific files related to DICE are maintained. Information about setting up the `dice_user` directory is provided as part of the instructions for installing and setting up DICE.

After using the DICE navigation utilities for several weeks, it is natural to build up a large collection of directory labels, and such a collection can easily be backed up, along with other relevant, user-specific DICE settings and files, by backing up one's `dice_user` directory.

To remove a label-directory association, one can use the DICE `rlk` command. The name `rlk` is short for "*r*emove (directory) *l*in*k*". The usage format is as follows.

```
rlk <label>
```

Assuming that the given label is currently associated with a specific directory from a prior call to `dlk`, the `rlk` command removes the association between the label and directory. This has the side effect of removing a small text file, since each label-directory association is stored as a separate text file in `dice_user/g`, as described earlier. Thus, especially when large collections of labels are involved, `rlk` can be useful to conserve disk space or reduce clutter in the `dice_user/g` directory.

The set of DICE directory navigation commands includes two simple wrappers around the common UNIX commands `pushd` and `popd`, which manipulate the directory stack as they change the current working directory. The wrappers are called `dxpushd` and `dxpopd`, respectively. The `dxpushd` and `dxpopd` commands perform the same functions

as their standard UNIX counterparts, except that they do not produce any text to standard output. Instead, their standard output is redirected to the DICE user files `dice_user/tmp/dxpushd_discard.txt` and `dice_user/tmp/dxpopd_discard.txt`, respectively. By redirecting the output in this way, the output is available for diagnostic reference as needed without cluttering standard output. This is useful, for example, when "pushing" and "popping" directories in scripts as it helps to keep the output from the scripts more relevant to the direct functionality of the scripts (rather than their internal use of `pushd` and `popd` operations).

The usage formats for `dxpushd` and `dxpopd` are the same as their standard UNIX counterparts: any arguments provided to these wrapper versions are passed on directly to `pushd` and `popd`, respectively.

The naming of the `dxpushd` and `dxpopd` commands illustrates a naming convention that is used often (but not everywhere) in DICE: that of prefixing the name of a DICE utility with "`dx`." DICE plug-in packages, such as the `dicelang` package, which is discussed in Section V, will in general have similar utility-name prefix conventions (e.g., the core part of `dicelang` uses the prefix `dlx`, while sub-packages within `dicelang` in general have their own specific prefixes, with each sub-package-specific prefix starting with the two letters "`dl`" to indicate that they are sub-packages of *d*ice*l*ang.

The DICE command `plk` ("*p*ush directory to *l*in*k*") works like the `g` command, except that the new directory is pushed onto the directory stack (using `dxpushd`) so that one can return to the original directory with a `dxpopd` or `popd` command.

The usage format for `plk` is as follows.

```
plk <label>
```

Here, as in the usage format specification for the `g` command, `<label>` is a label that has been associated with a directory through prior use of the `dlk` command.

B. MOVING THINGS AROUND

Relocating files and directories inside or across complex project directory structures can be tedious and prone to errors. DICE provides a collection of utilities that help move and copy files and subdirectories across different directories. We refer to these utilities informally as the DICE utilities for MTA ("moving things around"). These utilities can be especially convenient when used in conjunction with the directory navigation utilities described in Section III-A, but they can also be used independently of any other utilities.

Users who are upgrading from the previous version of DICE should take note that the names of several MTA utilities have changed in the latest version. This section provides the updated names of the utilities.

The DICE MTA utilities reference a standard user subdirectory in DICE that we called the *DICE user clipboard*. The DICE user clipboard resides within another standard user directory called the *DICE temporary directory* or simply, the *temporary directory* when the DICE qualification is understood from context. The paths of the DICE user clipboard and temporary directory are stored in the DICE environment variables `UXCLIPBOARD` and `UXTMP`, respectively. The value of the `UXTMP` variable (i.e., the location of the temporary directory) is set by default upon startup of DICE to be the path to a subdirectory called `tmp` in the DICE user directory. So, for example, if the DICE user directory is located at `~/dice_user`, then the DICE temporary directory is located at `~/dice_user/tmp`, and the DICE user clipboard is located at `~/dice_user/tmp/clipboard`.

The DICE temporary directory and DICE user clipboard can be labeled for fast navigation by running the following command sequence.

```
cd $UXTMP
dlk tmp
cd $UXCLIPBOARD
dlk clip
```

Of course, it is not necessary to label the `dlk` directory codes with `tmp` and `clip` — any other alphanumeric strings can be used instead at the user's choosing.

The core set of MTA utilities is as follows.

- `dxcu`: *cu*t a specified file or directory by moving it to the DICE user clipboard.
- `dxco`: *co*py a specified file or directory to the DICE user clipboard.
- `dxpa`: *pa*ste (copy) a specified file or directory from the DICE user clipboard.
- `dxpar`: *pa*ste (copy) and *r*emove a file or directory from the DICE user clipboard.

- `dxparl`: *pas*te (copy) and *r*emove the *l*ast file or directory transferred (LFDT) from the DICE user clipboard.
- `dxpal`: *pas*te the *L*FDT from the DICE user clipboard.

The usage format for `dxcu` is as follows.

```
dxcu <arg>
```

where `<arg>` is the name of a file or directory. The command moves the specified file or directory to the DICE user clipboard.

Such a file can be retrieved subsequently from any directory by running the `dxpar` command, which moves the specified file or directory from the DICE user clipboard to the current working directory. More precisely, the usage format for `dxpar` is as follows.

```
dxpar <arg>
```

where `<arg>` is the name of a file or directory. The command moves the file or subdirectory named `<arg>` from the DICE user clipboard, assuming that such a file or subdirectory exists. The utilities `dxcu` and `dxpar` are often used in conjunction with one another, but this is not a requirement: a file or subdirectory moved using `dxpar` need not have been placed originally in the DICE user clipboard using `dxcu`.

The utilities `dxco` and `dxpa` work like their cousins, `dxcu` and `dxpar`, except that they *copy* rather than move the specified files or directories. Their usage formats are analogous to those for `dxcu` and `dxpar` — i.e., they each take a single argument, which gives the name of a file or directory.

The `dxparl` and `dxpal` utilities are variations of `dxpar` and `dxpa`, respectively, that implicitly reference the *last file or directory transferred* (*LFDT*) by `dxcu` or `dxco`. Each call to `dxcu` or `dxco` has the side-effect of updating a DICE internal variable that stores the name of the LFDT.

Neither `dxparl` nor `dxpal` takes any arguments. These commands transfer the LFDT from the DICE user clipboard to the current working directory. They differ in that the LFDT is *moved* out of the DICE user clipboard with `dxparl`, whereas it is *copied* with `dxpal`.

As described earlier, the DICE MTA utilities can be especially convenient to use in conjunction with the directory navigation utilities described in Section III-A. As an example of this kind of convenience, suppose that `proj1` and `proj2` are project directories that have previously been labeled as `pr1` and `pr2`, respectively, using `dlk`. Suppose also that there is a file called `utilities.c` in the `proj1` directory that one wants to use a copy of or expand on in the `proj2` directory. This file can be copied to the `proj2` directory using the following sequence of commands.

```
g pr1
dxco utilities.c
g pr2
dxparl
```

This is equivalent to a copy-and-paste using basic mouse-based file operations, but this kind of command-based sequence can be much more convenient and efficient once one gets used to it. It is even more convenient when used in conjunction with standard file name auto-completion features in the Bash shell (e.g., see [2]).

Note that there are better ways to reuse code than copying code files, so this example should not be taken as a model for project development practices, but rather as an illustration of the combined use of navigation-related and MTA utilities in DICE.

Another, perhaps more common, scenario in which this kind of MTA functionality can be useful is when selecting a template from a repository of document templates (e.g., templates for business letters, personal letters, project reports, oral presentations, etc.). One can quickly make a copy of and starting working with the appropriate template with just a few commands — for example:

```
plk templates
dxco letter-from-home.tex
popd
dxparl
```

As suggested earlier, the MTA utilities need not be used in pairs — e.g., a `dxco` command need not be coupled with a corresponding `dxpar` or `dxparl` command. The MTA utilities can be used for any pattern of moving and copying files to and from the DICE user clipboard. For example, moving or copying a file to the DICE user clipboard can be useful if one wants to move a file or directory out of the way from the current working directory, but is not completely sure yet whether or not the file or directory will be needed again in the future.

Such forms of usage of the MTA utilities can generally be useful to help fine-tune directory organization. However, after a while, they can lead to excessive disk space requirements for the DICE temporary directory (which contains the DICE user clipboard). Thus, the temporary directory should periodically be cleaned out or "reset". A good time to do this is when there are no pending `dxpal` or `dxparl` commands, and just after one has backed up the temporary directory (perhaps as part of a general user space backup routine). That way, in case one needs to refer back to a file or directory that was "semi-confidently" discarded into the temporary directory, there is a means to recover the file or directory.

Recall that the DICE user clipboard is contained within the DICE temporary directory. In addition to the DICE user clipboard, the temporary directory serves as a repository for miscellaneous intermediate files that are generated by various DICE utilities. Because of the importance, in terms of disk space usage, of periodically emptying-out the DICE temporary directory, DICE is equipped with a simple utility called `dxclntmp` to perform this task. The `dxclntmp` utility removes all of the files and subdirectories within the DICE temporary directory, and resets the directory so that it contains just a single file. This file is a small, single-line log file called `dxclntmp-log.txt`, which contains a record of the time stamp — as returned by the UNIX `date` command — of the most recent invocation of `dxclntmp`.

The name `dxclntmp` is derived from "*cl*e*an* *temp*orary (directory)". The `dx` prefix used here is based on the selectively-applied DICE utility naming convention described in Section III-A.

There is also a DICE convenience utility, called `dxrmcl`, for removing individually-specified files or directories from the DICE user clipboard. The name stands for "*rem*ove from *cl*ipboard." The usage format for `dxrmcl` is as follows.

```
dxrmcl <arg>
```

where `<arg>` is the name of a file or directory in the DICE user clipboard. If `<arg>` represents a file, then the file is removed from the DICE user clipboard. Conversely, if `<arg>` represents a subdirectory in the DICE user clipboard, then the entire subdirectory (i.e., the entire subdirectory along with all directories that are nested within it) is removed. Caution should be exercised before using `dxrmcl` since files and directories in the DICE user clipboard that are removed by this command are removed *permanently*.

It is useful to be aware of the `dxrmcl` utility when working with the MTA utilities — specifically, when working with `dxcu` and `dxco`. This is because by default, the `dxcu` and `dxco` commands are *safe* in the sense that the specified file or directory will not be moved if a file or directory with the same name already exists in the DICE user clipboard. In other words, one cannot accidentally overwrite a file or directory using `dxco` or `dxcu`. Thus, for example, if one is trying to move a file to the DICE user clipboard using `dxcu` but finds (through the error message reported from `dxcu`) that a file with the same name already exists in the DICE user clipboard, one must somehow get rid of the identically-named DICE user clipboard file. A convenient way to do this, assuming that the DICE user clipboard file is no longer needed, is with `dxrmcl`.

The default "safe" configuration of `dxco` and `dxcu` can be changed through the environment variable `DX_MTA_AGGRESSIVE`. This environment variable controls what happens when the file or directory specified to `dxco` or `dxcu` conflicts with an identically-named file or directory in the DICE user clipboard. If `DX_MTA_AGGRESSIVE` is set to a non-null value, then the identically-named file or directory in the DICE user clipboard is silently overwritten. Conversely, if `DX_MTA_AGGRESSIVE` is not defined or is set to a null (empty string) value, then an error message is displayed and the copy or move request is denied.

Note that some DICE utilities create or manipulate files in the DICE temporary directory. For example, some utilities place diagnostic output in the temporary directory that can be examined for debugging user-defined command sequences or scripts that invoke DICE utilities. Thus, if one browses the contents of the temporary directory, it is not unusual to find files there that one has not explicitly placed.

C. OTHER UTILITIES

The discussion of utilities in this section provides a brief illustration of some of the features available in DICE. Many more utilities are available in DICE. Documentation for these utilities is being systematically integrated into an online documentation system that is under development for DICE, and is planned for inclusion in a future release.

IV. UNIT TESTING

DICE includes a framework for implementation and execution of tests for software projects. Although the emphasis in this framework is on unit tests, and therefore, it is often referred to as the *DICE unit testing framework*, the framework can also be applied to testing at higher levels of abstraction, including subsystem- and system-level testing.

A major goal of the testing capabilities in DICE is to provide a lightweight and flexible unit testing environment. It is lightweight in that it requires minimal learning of new syntax or specialized languages, and flexible in that it can be used to test source code in any language, including C, Java, Verilog, and VHDL. This is useful in heterogeneous development environments so that a common framework can be used to test across all of the relevant platforms. It is also useful in instructional settings so that students can focus on the core principles and practices of unit testing while spending minimal effort learning framework-specific conventions and syntax.

Each specific test in a DICE-based test suite is implemented in a separate directory, which is referred to as an *individual test subdirectory* (ITS). To be processed by the DICE facilities for automated test suite execution, the name of an ITS must begin with "test" (e.g., `test01`, `test02`, `test-A`, `test-B`, `test_square_matrix_1`, `test_square_matrix_2`). To exclude a test from test suite evaluation, one can simply change its name so that it does not begin with "test".

A. Required components of an ITS

The following are required components of an ITS. Each of these components takes the form of a separate file.

- A `README.txt` file that provides a brief explanation of what is being tested by the test — that is, what is distinguishing about this test compared to the other tests in the test suite. Technically, a `README.txt` file is not *required* within an ITS, but we list it here among all of the actual requirements because it is of foremost importance to have tests properly documented in a place that is easy to find.
- An executable file (e.g., some sort of script) called `makeme` that performs all necessary compilation steps (e.g., compilation steps that are used to build a driver program) that are needed for the test. In DICE, we use the convention that files that do not have file name extensions are Bash scripts. Thus, tests that are developed as part of DICE have their `makeme` files implemented as Bash scripts. Note that the compilation steps performed in the `makeme` file for a test typically do *not* include compilation of the source code that is being tested. It is assumed that source code for a project under test is compiled separately before a test suite associated with the project is exercised. Thus, compilation functionality within the test suite specifies only compilation steps that are specific to the tests.

  Note also that for some kinds of tests, no compilation steps are needed apart from the compilation that is performed on the source code that is being tested. For example, if a test is invoked by running an executable program that is part of the project being tested, then it is likely that no test-specific compilation needs to be performed. In such a case, a `makeme` file should still be present in the ITS directory; however, its contents can be empty or can consist only of comments (e.g., comments that indicate that nothing needs to be done to build the test).
- An executable file called `runme` that runs the test associated with the enclosing ITS, and directs the normal output associated with the test to standard output, and the error output associated with the test to standard error. As with `makeme` files, `runme` files are typically scripts, and within DICE test suites, `runme` scripts are Bash scripts. If the test takes input from standard input, then the `runme` script should generally redirect that input to come from a file that is stored within the ITS.
- A file called `correct-output.txt` that contains the standard output text that should result from the test — i.e., the text that should appear if `runme` is executed, and the project functionality that is being tested has been implemented correctly. If the correct operation of the test does not produce any standard output text, then the `correct-output.txt` file should exist within the ITS as an empty file (i.e., a file that contains no characters).
- A file called `expected-errors.txt` that

contains the standard error text that should result from the test — i.e., the text that should appear on standard error if `runme` is executed, and the project error handling capability (if any) that is being tested has been implemented correctly. If the test does not exercise any error handling capability, then the `expected-errors.txt` file should exist within the ITS as an empty file.

B. Examples

As a supplement to the DICE User's Guide [3], one can find two simple project examples that involve program and function versions of vector inner product computation. These examples illustrate construction of ITSs and basic use of DICE project testing features. These examples are in the form of simple C-language projects with test suites that are implemented according to the DICE ITS conventions described in this section. These examples can easily be adapted to provide basic demonstrations of, and starting points for experimenting with DICE testing capabilities in other programming languages.

To try out these testing examples, it is recommended that users start with the program version rather than the function version, as this version is a little easier to understand. To try either version, one should first `cd` to the project `src` directory, and run `./makeme`, which compiles the project. Then one can `cd` to the project `test` directory, and run the DICE command `dxtest` to exercise all of the tests in the test suite.

C. The DICE `dxtest` *command*

The DICE `dxtest` command is the core utility available in DICE for exercising test suites. The usage format of this command is as follows.

```
dxtest [-v]
```

The `dxtest` utility recursively traverses the directory subtree located in the current working directory (the directory from which `dxtest` is called). During this traversal, only directories whose names begin with "test" are actually visited; all other directories are ignored. For any directory that is encountered with a name that does not begin with "test", the entire directory subtree rooted at that directory is ignored (even if the subtree contains subdirectories whose names start with "test").

Each time `dxtest` visits a new directory, the script checks whether or not the directory contains a file called `runme`. If a `runme` file is found, then the directory is treated as an ITS, and all of the required ITS files listed above are expected to exist, and are processed based on the descriptions given above. Specifically, the `makeme` file of the ITS is first executed to perform any necessary steps required to build the test. Then the `runme` file is executed to exercise the test. The output generated by `runme` is then compared with the `correct-output.txt` and `expected-errors.txt` files to determine whether or not the test succeeded.

After the entire recursive directory traversal is complete, `dxtest` produces a summary of how many tests (ITSs) were exercised, how many of these tests succeeded, and how many failed. Furthermore, if any test failures were encountered, a listing of the directory paths corresponding to the failed ITSs is provided in `autotest-output/test_summary.txt`.

Thus, with a high degree of automation, one can assess the overall success rate of a test suite and identify any specific tests that are failing.

D. Running tests in verbose mode

The `-v` option can be used with `dxtest` to provide *verbose output* as the test suite is exercised. This can be useful if the test suite is exhibiting some sort of unexpected behavior. For example, if the test suite is taking longer than expected to finish execution because one of the tests is "hanging" (e.g., due to an infinite loop), then verbose output can be enabled to locate the offending test.

It is useful, however, to run tests with verbose output "off" (i.e., by leaving out the `-v` option) before any sort of finalization of a testing pass (e.g., before committing changes to a shared code repository). This is because some errors that occur when running a test suite (e.g., problems executing a `makeme` or `runme` script) can be hard to notice amidst the normal verbose output. On the other hand, these kinds of problems are exposed clearly when verbose output is turned off.

E. More about `runme` *files*

As described earlier, the success or failure of an individual DICE test (ITS evaluation) is determined by comparing the standard output and standard error text generated from the associated `runme`

script with the given `correct-output.txt` and `expected-errors.txt` files, respectively.

This convention provides significant flexibility in how test output is actually defined and managed. In particular, it is not necessary for all of the output produced by the project code under test to be treated directly as test output (i.e., to be compared with `correct-output.txt` and `expected-errors.txt`) during each test evaluation. Instead, the `runme` file can serve as a wrapper to filter or reorganize the output generated by a test in a form that the user finds most efficient or convenient for test management.

For example, suppose that the project under test is a hardware description language (HDL) implementation in a language such as Verilog or VHDL, and the relevant output for one of the tests consists of three simulator output files `sim1.txt`, `sim2.txt` and `sim3.txt`. The "brute force" way to develop a `runme` script for this test would be to invoke the HDL simulator in the `runme` script, and then concatenate the files `sim1.txt`, `sim2.txt`, and `sim3.txt` to standard output (e.g., by using the UNIX `cat` command). The `correct-output.txt` file for such a test configuration would contain the concatenated contents of `sim1.txt`, `sim2.txt`, and `sim3.txt` that should result from a correct project implementation — i.e., the concatenated result of the three, pre-verified, "golden" simulation output files.

An alternative approach for this testing scenario, which may be preferable in many contexts to such a brute force approach, is to maintain the golden simulation output files in separate files — e.g., `correct-sim1.txt`, `correct-sim2.txt`, and `correct-sim3.txt` within the ITS. The `runme` script could then use the UNIX `diff` command to compare the files `sim1.txt`, `sim2.txt`, and `sim3.txt` produced from a test run with the corresponding golden output files — the trailing code in the `runme` script would then look something like:

```
diff sim1.txt correct-sim1.txt
diff sim2.txt correct-sim2.txt
diff sim3.txt correct-sim3.txt
```

The exact operation of the UNIX `diff` utility is not completely standardized across different platforms. However, a typical convention used in implementations of `diff` is to produce output to standard output if and only if the files being compared differ in at least one character. Under such a convention, the `correct-output.txt` file for our hardware description language test would simply be an empty file. This empty `correct-output.txt` file together with the three files `correct-sim1.txt`, `correct-sim2.txt`, and `correct-sim3.txt` comprise the normal (error-free) output verification files associated with this ITS.

In summary, it is the standard output produced from `runme` that is used by `dxtest` to validate an ITS against the associated `correct-output.txt` file. Through appropriate programming of the `runme` file, the standard output of `runme` is in general highly configurable by the person who develops the test. Creative design of `runme` files can help to make more powerful and convenient test organizations within the DICE testing framework.

V. DICELANG

The `dicelang` package, which can be viewed as a companion package of DICE, provides a collection of language-specific plug-ins that extend the features of DICE, and provide new features to facilitate efficient software project development, implementation management, and testing for selected programming languages. In contrast, the features in DICE emphasize generality, and applicability across different kinds of programming languages and development tools.

There is a benign, but possibly confusing (at first) circular dependence between DICE and `dicelang-C`, which is the C language plug-in within `dicelang`. This dependency arises because C, along with Bash and Python, is one of the languages in which DICE is implemented. Unlike programs in Bash and Python, C programs need to be compiled before they can be executed, and for the purpose of compiling the C-based components in DICE, we use some features in `dicelang-C` for building projects that are developed in C. Thus, `dicelang` must be installed and built before DICE can be built. However, DICE must be started up (without building the C-based components in DICE) before `dicelang-C` can be built. This is where the "circular dependency" described above arises.

The DICE User's Guide [3] includes instructions for handling this circular dependency, and getting started

using both DICE, including all of its C-based components, and `dicelang-C`.

More information about the `dicelang` package is available from the DICE User's Guide [3].

VI. RELATIONSHIP TO DIF

The dataflow interchange format (DIF), which includes The DIF Language (TDL) and The DIF Package (TDP), is another tool developed by the Maryland DSPCAD Research Group [4], [5]. TDL and TDP provide tools for model-based design and implementation of signal processing systems using dataflow graphs. DIF is orthogonal to DICE, so that one can use DICE without DIF and vice versa.

Even with best practices in industry, embedded software development involves an initial application description in a design environment, which is then manually transcoded and tuned to target the final design platform. Often separated by languages, tools, and even different teams, going from an initial application description to a final implementation tends to be a manual, error-prone, time-consuming problem. To improve quality and performance while reducing development time, a cross platform design environment is needed that accommodates both early design exploration and final implementation tuning.

DIF and DICE are complementary tools, which when used in tandem, enhance software development processes for dataflow-based design and implementation of signal processing systems. A designer starts with a platform-independent description of the application in DIF. This structured, formal application description is an effective starting point for capturing concurrency and optimizing and analyzing the application. After settling on the DIF description, a designer can refine this description to an optimized implementation by employing platform specific tools including compilers, debuggers, and simulators. Any transcoding or platform specific enhancements are accommodated by DICE via its flexible but standardized build and test framework. This allows designers to utilize the same design framework at inception as they do at final implementation.

Software developed jointly with DIF and DICE enjoys a single, cross platform software management framework, where verification of modules is handled consistently throughout each phase of development. By leveraging the reference DIF description, transcoding effort is saved by having a formal, unambiguous application description to base the implementation on. Quality is controlled with a high degree of automation through the direct reuse of unit tests in DICE.

VII. RELATIONSHIP TO OTHER UNIT TESTING FRAMEWORKS

Typically when software designers employ unit testing, they use frameworks that are language specific [6], [7]. More than just a syntactic customization, 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 [8]. 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 creates 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, 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 test bench 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.

Probably the most related framework to DICE is the Test Anything Protocol (TAP) [9]. 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.

DICE provides a language-agnostic approach to unit test writing by leveraging common dataflow constructs. Some unit test frameworks have data generators, but DICE encourages designers to think of module interfaces in terms of streaming data primitives into and out of them. DICE captures these input/output sequences in files and then ensures that the output files match with a structured build-and-run framework. These assumptions allow test writers in DICE to build more complete language-agnostic solutions than a test communication protocol alone.

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 (e.g., use of `dxco`, and `dxparl` to copy test directory templates from one place to another, and use of navigation utilities to jump back and forth between test directories and project source code). 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.

VIII. SUMMARY

This report has provided a motivation and overview of DICE, the DSPCAD integrative command-line environment. DICE is geared towards promoting productivity and efficiency in the management of software implementations. Significant emphasis has been placed in DICE on features that can be used directly or easily adapted across different programming languages. This report has also provided brief introductions to some of the general utility commands in DICE; the unit testing features in DICE; and the companion package, `dicelang`, which provides programming-language-specific plug-ins that extend the capabilities of DICE in ways that are customized to specific languages.

For detailed information on DICE, including more comprehensive documentation, as well as information on downloading and setting up DICE, we refer the reader to the DICE Project Website [1], and the DICE User's Guide [3]. More detailed discussions are also available on applications of DICE to specific kinds of design problems. For example, the application of DICE to design and implementation of high performance field-programmable gate array systems is discussed in [11], and applications to analysis and optimization of precision in signal processors are discussed in [12].

IX. ACKNOWLEDGMENTS

This work is sponsored in part by the U. S. National Science Foundation under grant NSF-ECCS0823989, the Laboratory for Telecommunication Science, and the US Air Force Research Laboratory.

We are grateful also to the following people who have made valuable contributions to DICE and `dicelang`, and to earlier software components that have evolved into parts of DICE and `dicelang`: Bishnupriya Bhattacharya, Nitin Chandrachoodan, Soujanya Kedilaya, and Robert Ricketts.

REFERENCES

[1] "DICE project website," http://www.ece.umd.edu/DSPCAD/projects/dice/dice.htm.

[2] C. Newham and B. Rosenblatt, *Learning the Bash shell*, O'Reilly & Associates, Inc., third edition, 2005.

[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," Tech. Rep. DSPCAD-TR-2011-13, Institute for Advanced Computer Studies, University of Maryland at College Park, 2011.

[4] 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*, Dallas, Texas, September 2005, pp. 37–49.

[5] W. Plishker, N. Sane, M. Kiemb, K. Anand, and S. S. Bhattacharyya, "Functional DIF for rapid prototyping," in *Proceedings of the International Symposium on Rapid System Prototyping*, Monterey, California, June 2008, pp. 17–23.

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

[7] H. Gollee T. Dohmke, "Test-driven development of a PID controller," *IEEE Software*, vol. 24, no. 3, pp. 44–50, 2007.

[8] P. Hamill, *Unit test frameworks*, chapter 7, O'Reilly & Associates, Inc., 2004.

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

[10] R. M. Hierons et al., "Using formal specifi cations to support testing," *ACM Computing Surveys*, vol. 41, no. 2, February 2009.

[11] W. Plishker, C. Shen, S. S. Bhattacharyya, G. Zaki, S. Kedilaya, N. Sane, K. Sudusinghe, T. Gregerson, J. Liu, and M. Schulte, "Model-based DSP implementation on FPGAs," in *Proceedings of the International Symposium on Rapid System Prototyping*, Fairfax, Virginia, June 2010, Invited paper, DOI 10.1109/RSP_2010.SS4, 7 pages.

[12] S. Kedilaya, W. Plishker, A. Purkovic, B. Johnson, and S. S. Bhattacharyya, "Model-based precision analysis and optimization for digital signal processors," in *Proceedings of the European Signal Processing Conference*, Barcelona, Spain, August 2011, To appear.