Using the DSPCAD Integrative Command-Line Environment: User's Guide for DICE Version 1.0 *

Shuvra S. Bhattacharyya, Soujanya Kedilaya, William Plishker,
Nimish Sane, Chung-Ching Shen, and George Zaki
Department of Electrical and Computer Engineering, and
Institute for Advanced Computer Studies
University of Maryland at College Park, USA
{ssb, soujanya, plishker, nsane, ccshen, gzaki}@umd.edu

This document provides a user's guide for DICE Version 1.0. The emphasis in this user's guide is on providing detailed specifications for setting up and using the various features in DICE. For a general overview of DICE and descriptions of some of its core features, see Reference [1].

This user's guide is supplemented by various materials that are available electronically from the DICE User's Guide Online Supplement [2].

1 What is DICE?

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 Windows (equipped with Cygwin), Solaris, and Linux platforms.

*Technical Report DSPCAD-TR-2009-01, Maryland DSPCAD Research Group, Department of Electrical and Computer Engineering, University of Maryland at College Park, 2009 (Revision: September 6, 2009).

1.1 What DICE is not

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.

1.2 Notational Conventions

The following notational conventions are used in this document.

- Many cross-references in this document are hyperlinked for convenient reference when one is viewing the electronic (PDF) version of this guide.
- 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.
- Items in command descriptions that are enclosed by angle brackets (`<...>`) indicate placeholders for command arguments or other text that needs to be customized based on the context of the command.
- Items in command descriptions that are enclosed by square brackets (`[...]`) indicate *optional* command arguments.

2 Setting up DICE

2.1 Downloading DICE

Download DICE (`dice.tar.gz`) from the Online Supplement [2], and extract the `dice/` directory from the `tar.gz` archive in which it is packaged. Move the `dice/` directory to the location in which you want the installation to reside (i.e., your *DICE installation directory*).

2.2 Setting up your `dice_user` Directory

A `dice_user` directory is a directory in which certain user-specific files associated with one's DICE environment are stored. This directory must be called `dice_user`. A typical location for this directory is `~/dice_user` (i.e., in one's home directory). To set up this directory, just create an empty directory called `dice_user` in the desired location, and then create a subdirectory of `dice_user` called `startup`.

For example:

```
cd ~
mkdir dice_user
cd dice_user
mkdir startup
```

This constructs the following directory structure:

```
~/dice_user/
    startup/
```

It is generally advised that users avoid creating their own *specialized* directories or files in `dice_user` or any subdirectory within `dice_user` except through DICE or `dicelang` commands that create those files or directories as side effects. For example, the `dxbuild` command creates as a side effect some files within `dice_user/dxgen`. Here, by "specialized," we mean files or directories that are not required as part of the standard set up process for DICE or one of its plug-in packages. The one exception to this advice is the `$UXTMP` directory, which is discussed in Section 6. The `$UXTMP` directory is intended to be a scratch area for arbitrary user work, including direct (not necessarily through DICE-related commands) creation, manipulation, and deletion of files and directories. Keeping the DICE user tree as "standard" as possible will help to avoid confusion between DICE-related and non-DICE-related user files and directories.

2.3 The Standard DICE Startup File

Download the standard DICE startup file (`dice_startup`) from the Online Supplement [2]. Place the `dice_startup` file in your `dice_user/startup` directory. Typically, this startup file is used "as is," and does not need any editing by the user.

2.4 Essential UX Definitions

Download the DICE template file `uxdefs_dice` for essential "UX definitions". This file is also available from the Online Supplement [2]. "UX" is a prefix used for DICE-related environment variables that correspond to user-specific customizations. Place the file `uxdefs_dice` in `dice_user/startup`.

Open the file `uxdefs_dice` with a text editor. Set to appropriate values the right hand sides of the two assignment statements at the bottom of the file. The meanings of the environment variable settings that are involved in these assignment statements are as follows.

- `UXARCH`: This is used to represent the host platform on which your installation of DICE is being used. Available options are: `lin` (Linux), `sol` (Solaris), and `win` (Windows).
- `UXDICE`: This should be set to the (UNIX/Cygwin-format) directory path of your DICE installation directory.

For example, suppose you are running DICE on Windows, and you have placed your DICE installation in `/home/utils/`. Then your environment variable settings in `uxdefs_dice` would be as follows.

```
UXARCH=win
UXDICE=/home/utils/dice
```

Note that on some platforms, you may need to process the `uxdefs_dice` and `dice_startup` files with the standard `dos2unix` utility before they will execute properly. Such processing just needs to be done once during initial setup of DICE.

2.5 Starting up DICE

To start up DICE, which involves loading necessary environment settings so that you can use all of the features in DICE, follow these steps:

1. Start a bash shell.
2. `cd` to your DICE user directory (e.g., run `cd ~/dice_user`).
3. run

   ```
   source startup/dice_startup
   ```

IMPORTANT: The `dice_startup` file must be invoked from the `dice_user` directory — e.g., as opposed to running it as

```
source dice_startup
```

from `~/dice_user/startup` or running it from your home directory.

If you encounter difficulties starting up DICE, see Section 9.

2.6 Building DICE

To use all of the features available in DICE, you need to build certain executables from the corresponding source code. To do this, you need to first make sure that you have the DICE plug-in package `dicelang` installed and built — if not, then first install and build `dicelang` by following the instructions in Section 3.

After you have ensured that `dicelang` is built, simply startup DICE and `dicelang`, and then, from the same bash session, run the DICE utility `dxbuild` (with no arguments). This step of building DICE needs to be done only once (at installation/setup time) for a given version/update of DICE.

This benign circular dependency between DICE and `dicelang` can be a bit confusing at first. However, the dependency does not cause problems if the packages are built using the proper sequence of steps. To summarize, the following sequence should be followed to correctly build DICE and `dicelang`.

1. Start up DICE.
2. Start up `dicelang`.
3. Build `dicelang` (run `dlxbuild` as described in Section 3).
4. Build DICE (run `dxbuild`).

3 Setting up `dicelang`

3.1 Downloading `dicelang`

Download `dicelang` (`dicelang.tar.gz`) from the Online Supplement [2], and extract the `dicelang` directory from the `tar.gz` archive in which it is packaged.

Move the `dicelang` directory to the location in which you want the installation to reside (i.e., your ***`dicelang`*** *installation directory*).

3.2 Setting up your `dicelang_user` Directory

Set up your `dicelang_user` directory, and its startup subdirectory. The `dicelang_user` directory is a directory in which certain user-specific files associated with one's `dicelang` environment are stored. This directory must be called `dicelang_user`, and it must be placed in your `dice_user` directory. For more information about the `dice_user` directory, see Section 2.2.

For example, if your `dice_user` directory is in your home directory:

```
cd ~/dice_user
mkdir dicelang_user
cd dicelang_user
mkdir startup
```

This constructs the following directory structure:

```
~/dice_user/
    dicelang_user/
        startup/
```

3.3 The Standard `dicelang` Startup File

Download the standard `dicelang` startup file (`dicelang_startup`) from the Online Supplement [2], and place it in your `dicelang_user/startup` directory. Typically, this startup file is used "as is," and does not need any editing by the user.

3.4 Essential UX Definitions

Download the `dicelang` template file (`uxdefs_dicelang`) for essential "UX definitions" associated with `dicelang`. This file is also available from the Online Supplement [2]. Place the file `uxdefs_dicelang` in `dicelang_user/startup`.

Open the file `uxdefs_dicelang` with a text editor. Set to an appropriate value the right hand side of the assignment statement at the bottom of the file. The meaning of the environment variable setting that is involved in this assignment statement is as follows.

- `UXDICELANG`: This should be set to the (UNIX/Cygwin-format) directory path of your `dicelang` installation directory.

For example, suppose you are running `dicelang` on Windows, and you have placed your `dicelang` installation in `/home/utils/`. Then your environment variable setting in `uxdefs_dicelang` would be as follows.

```
UXDICELANG=/home/utils/dicelang
```

Note that on some platforms, you may need to process the `uxdefs_dicelang` and `dicelang_startup` files with the standard `dos2unix` utility before they will execute properly. Such processing just needs to be done once during initial setup of `dicelang`.

3.5 Starting up `dicelang`

To startup `dicelang`, you need to first make sure that you have installed (but not necessary built) and started up DICE. Then in the bash shell in which you have already started up DICE, run

```
dxsource dx_load_package dicelang <dice_user_dir>
```

Here, `<dice_user_dir>` represents the path to your DICE user directory. For example, if your DICE user directory is `~/dice_user` then the command to start up `dicelang` would be:

```
dxsource dx_load_package dicelang ~/dice_user
```

If you encounter difficulties starting up `dicelang`, see Section 9

3.6 Building `dicelang`

To use all of the features available in `dicelang`, you need to build certain executables from the corresponding source code. To do this, simply startup `dicelang`, and then, from the same bash session, run the `dicelang` utility `dlxbuild` (with no arguments). This step of building `dicelang` needs to be done only once for a given version/update of `dicelang`.

Note that DICE needs to be started before `dicelang` is started, but before all DICE utilities can be built, `dicelang` must be built with the instructions given above.

3.7 Starting up both DICE and `dicelang` from another bash Startup File

If you want to load DICE from within another bash startup file, then include the following commands in that startup file:

```
pushd <dice_user_dir>
source startup/dice_startup
popd
```

Here `<dice_user_dir>` represents the path to your DICE user directory.
For example:

```
pushd ~/dice_user
source startup/dice_startup
popd
```

To also startup `dicelang` from the same higher-level, bash startup file, insert the following command *after* the commands for starting up DICE.

```
dxsource dx_load_package dicelang <dice_user_dir>
```

For example:

```
dxsource dx_load_package dicelang ~/dice_user
```

4 Displaying Version Information

The `dxversion` command can be used to check the version number for the version of DICE that one is currently using. The `dxversion` command does not take any arguments and reports the relevant version number and revision date to standard output.

Similarly, the `dlxversion` command can be used to check the version number for the version of `dicelang` that one is currently using. Like `dxversion`, the `dlxversion` command does not take any arguments and reports the relevant version number and revision date to standard output.

5 Directory Navigation

This section motivates and describes utilities in DICE for efficient 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 gives 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 the 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 the directory

```
~/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 8, 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.

6 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 helps to 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 5, but they can also be used independently of any other utilities.

The DICE MTA utilities reference a standard user subdirectory in DICE that we call the *DICE temporary directory* or simply, the *temporary directory* when the DICE qualification is understood from context. The path of this directory is stored in the DICE environment variable `UXTMP`, and the value of this variable (i.e., the location of the DICE 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`.

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

```
cd $UXTMP
dlk tmp
```

Of course, it is not necessary to label the directory with *tmp* — any other alphanumeric string can be used instead at the user's choosing.

The core set of MTA utilities are `mvttmp`, `mvftmp`, `mvftmpl`, `cpttmp`, `cpftmp`, and `cpftmpl`. These names stand, respectively, for *mov*e *t*o DICE *temp*orary directory; *mov*e *f*rom DICE *temp*orary directory; *mov*e *f*rom DICE *temp*orary directory — *l*ast file; *cop*y *t*o DICE *temp*orary directory; *cop*y *f*rom DICE *temp*orary directory; and *cop*y *f*rom DICE *temp*orary directory — *l*ast file.

The usage format for `mvttmp` is as follows.

```
mvttmp <arg>
```

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

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

```
mvftmp <arg>
```

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

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

The mvftmpl and cpftmpl utilities are variations of mvftmp and cpftmp, respectively, that implicitly reference the last file or directory transferred (LFDT) by mvttmp or cpttmp. Each call to mvttmp or cpttmp has the side-effect of updating a DICE internal variable that stores the name of the LFDT.

Neither mvftmpl nor cpftmpl takes any arguments. These commands transfer the LFDT from the DICE temporary directory to the current working directory. They differ, as their names suggest, in that the LFDT is *moved* out of the temporary directory with mvftmpl, whereas it is *copied* with cpftmpl.

As described earlier, the DICE MTA utilities can be especially convenient to use in conjunction with the directory navigation utilities described in Section 5. 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
cpttmp utilities.c
g pr2
mvftmpl
```

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.

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
cpttmp letter-from-home.tex
popd
mvftmpl
```

As suggested earlier, the MTA utilities need not be used in pairs — e.g., a `cpttmp` command need not be coupled with a corresponding `mvftmp` or `mvftmpl` command. The MTA utilities can be used for any pattern of moving and copying files to and from the DICE temporary directory. For example, moving or copying a file to the temporary directory 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 increasing disk space requirements for the DICE temporary directory. Thus, the temporary directory should periodically be cleaned out or "reset". A good time to do this is when there are no pending `cpftmpl` or `mvftmpl` 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.

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*ea*n* *temp*orary (directory)". The `dx` prefix used here is based on the selectively-applied DICE utility naming convention described in Section 5.

There is also a DICE convenience utility, called `rmftmp`, for removing individually specified files or directories from the temporary directory. The name stands for "*r*e*m*ove *f*rom *temp*orary directory". The usage format for `rmftmp` is as follows.

`rmftmp <arg>`

where `<arg>` is the name of a file or directory in the DICE temporary directory. If `<arg>` represents a file, then the file is removed from the temporary directory. Conversely, if `<arg>` represents a subdirectory in the temporary directory, 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 `rmftmp` since files and directories in the temporary directory that are removed by this command are removed *permanently*.

It is useful to be aware of the `rmftmp` utility when working with the MTA utilities — specifically, when working with `mvttmp` and `cpttmp`. This is because the `mvttmp` and `cpttmp` 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 temporary directory. In other words, one cannot accidentally overwrite a file or directory using *cpttmp* or *mvttmp*. Thus, for example, if one is trying to move a file to the temporary directory using *mvttmp* but finds (through the error message reported from *mvttmp*) that a file with the same name already exists in the temporary directory, one must somehow get rid of the identically-named temporary directory file. A convenient way to do this, assuming that the temporary directory file is no longer needed, is with *rmftmp*.

The default "safe" configuration of `cpttmp` and `mvttmp` can be changed through the environment variable `DX_MTA_AGGRESSIVE`. This environment variable controls what happens when the file or directory specified to `cpttmp` or `mvttmp` conflicts with an identically-named file or directory in the DICE user temporary directory. If `DX_MTA_AGGRESSIVE` is set to a non-null value, then the identically-named file or directory in the temporary directory 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.

7 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".

7.1 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., steps for compiling any 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.

7.2 Examples

In the Online Supplement [2], 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.

7.3 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.

7.4 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.

7.5 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` 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.

7.6 Other DICE Features

The discussion of utilities in this user's guide provides coverage of some of the key features available in DICE. Many more utilities are available in DICE. Subsequent revisions of this user's guide will incorporate more thorough documentation on DICE utilities.

8 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.

More information about `dicelang` will be provided in subsequent revisions of this User's Guide.

9 Troubleshooting Guidelines

1. Directories and filenames with spaces. Unless otherwise specified, DICE generally does not work well with file or directory names and paths that contain spaces. This applies to arguments of utilities as well as DICE-related environment variable settings, and it often applies even if the arguments or settings are quoted. For example, a setting like the following one is likely to cause problems.

   ```
   UXDICE="/cygdrive/c/Documents and Settings/\
   ann/My Documents/programs/dice"
   ```

   On the other hand, because the right hand side does not contain any spaces, the following setting is fine (even if the full path to the user's home directory includes directory names that contain spaces).

   ```
   UXDICE=~/utils/dice
   ```

   Note that an important exception to the "space problem" is the family of navigation-related utilities, which is described in Section 5. In particular, `dlk` *can* be used to define labels for directories whose paths contain spaces. For example, the following sequence will work as long as the targets of the `cd` commands are valid.

   ```
   cd "/cygdrive/c/Documents and Settings/ann/My Documents"
   dlk mydocs
   cd ~/somewhere_else
   g mydocs
   ```

2. Windows-style paths. When using DICE under Windows, it is important to keep in mind that DICE generally requires UNIX-style paths. Such paths should not contain Windows drive specifiers such as `C:` or `D:`, nor should they contain the backward slash character (\) as a directory separator.

   For example, the following are likely to cause problems.

   ```
   UXDICE=C:/smith/downloaded/dice
   UXDICE=C:\\smith\\downloaded\\dice
   ```

   On the other hand, the following forms are acceptable.

   ```
   UXDICE=/cygdrive/c/smith/downloaded/dice
   UXDICE=~/smith/downloaded/dice
   ```

3. `DXVERBOSE`. A useful diagnostic feature to use when setting up DICE and DICE plug-ins is the `DXVERBOSE` environment variable. If this variable is set before starting up DICE to any non-empty-string value (e.g., with `DXVERBOSE=on`), then diagnostic comments will be displayed as DICE is started up. This is usually not needed, but can be useful to help diagnose or report a problem if DICE or a DICE plug-in fails to start up properly.

4. `DXVERBOSE` for DICE utilities. Some DICE utilities produce diagnostic output that is enabled by the `DXVERBOSE` environment variable, as described above. Thus, the `DXVERBOSE` variable can also be useful in diagnosing configuration or usage problems related to DICE that are not associated with the setup of DICE or the DICE startup process.

5. Exiting during startup. Another point that is useful to keep in mind when troubleshooting DICE setup issues is that the enclosing bash shell may exit if DICE or one of its plug-in packages fail to startup. If this happens, then enable verbose output, and redirect standard output to a file in your startup command.

   For example, to apply this method when starting up DICE, one can use the following sequence of commands.

   ```
   DXVERBOSE=on
   source startup/dice_startup > ~/startup-log.txt
   ```

   Then one can view the `~/startup-log.txt` file for a transcript of the failed startup session.

   Similarly, if the DICE user directory is `~/dice_user`, then to apply this troubleshooting approach when starting up `dicelang`, one can use the following command (from any directory).

   ```
   DXVERBOSE=on
   dxsource dx_load_package dicelang ~/dice_user > \
           ~/startup-log.txt
   ```

10 Acknowledgments

This work was supported in part by the U. S. National Science Foundation under grant NSF-ECCS0823989.

We are grateful also to the following people who have made valuable contributions to DICE and `dicelang`: Bishnupriya Bhattacharya, Nitin Chandrachoodan, Robert Ricketts.

References

[1] S. S. Bhattacharyya, S. Kedilaya, W. Plishker, N. Sane, C. Shen, and G. Zaki, "The DSPCAD integrative command line environment: Introduction to DICE version 1," Institute for Advanced Computer Studies, University of Maryland at College Park, Tech. Rep. UMIACS-TR-2009-13, August 2009.

[2] "DICE user's guide online supplement," http://www.ece.umd.edu/DSPCAD/projects/dice/guide/supplement.htm.