The DSPCAD Lightweight Dataflow Environment: Introduction to LIDE Version 0.1 *

Chung-Ching Shen, Lai-Huei Wang, Inkeun Cho, Scott Kim,
Stephen Won, William Plishker, and Shuvra S. Bhattacharyya
Department of Electrical and Computer Engineering, and
Institute for Advanced Computer Studies
University of Maryland at College Park, USA
{ccshen, laihuei, inkeun, sckim, swon, plishker, ssb}@umd.edu

November 21, 2011

Abstract

LIDE (the *DSPCAD Lightweight Dataflow Environment*) is a flexible, lightweight design environment that allows designers to experiment with dataflow-based approaches for design and implementation of digital signal processing (DSP) systems.

LIDE contains libraries of dataflow graph elements (primitive actors, hierarchical actors, and edges) and utilities that assist designers in modeling, simulating, and implementing DSP systems using formal dataflow techniques. The libraries of dataflow graph elements (mainly actors) contained in LIDE provide useful building blocks that can be used to construct signal processing applications, and that can be used as examples that designers can adapt to create their own, customized LIDE actors. Furthermore, by using LIDE along with the *DSPCAD Integrative Command Line Environment* (*DICE*), designers can efficiently create and execute unit tests for user-designed actors.

This report provides an introduction to LIDE. The report includes details on the process for setting up the LIDE environment, and covers methods for using pre-designed libraries of graph elements, as well as creating user-designed libraries and associated utilities using the C language. The report also gives an introduction to the C language plug-in for `dicelang`. This plug-in, called `dicelang-C`, provides features for efficient C-based project development and maintenance that are useful to apply when working with LIDE.

1 Introduction

This report provides an introduction to LIDE, which stands for the *DSPCAD Lightweight Dataflow Environment*. The objective of LIDE is to provide a flexible, lightweight design environment that allows designers to experiment with dataflow-based approaches for design and implementation of digital signal processing (DSP) systems. The package is intended for cross-platform usage, and is currently being developed and used actively on the Linux, MacOS, Solaris, and Windows (equipped with Cygwin) platforms.

LIDE contains a collection of pre-designed libraries of *dataflow graph elements* (components for implementing vertices and edges in dataflow graphs), including a variety of useful actors and FIFOs, for building DSP systems that are specified using dataflow semantics. LIDE also contains useful utility functions for conducting functional simulations for and implementing DSP systems. With the assistance of LIDE, designers can also create and integrate their own dataflow graph elements and schedulers to construct customized dataflow-based systems.

Actor design in LIDE is based on the semantics of a specific dataflow model of computation called *Enable-Invoke Dataflow* (EIDF) [1], and the *Lightweight Dataflow* (LWDF) programming methodology [2], which

*Technical Report UMIACS-TR-2011-17, Institute for Advanced Computer Studies, University of Maryland at College Park, 2011.

can be viewed as an abstract programming model for EIDF that can be targeted to different platform- or simulation-oriented languages. For general background on DSP-oriented dataflow models of computation, we refer the reader to [3], which provides an extensive review of fundamental and state-of-the-art methods.

In LIDE, actor design includes four interface functions — the *construct*, *enable*, *invoke*, and *terminate* functions. The `construct` function defines the topological connections between an actor and its input and output FIFO buffers (i.e., its incident dataflow graph edges), and performs any other pre-execution initialization associated with the actor. The `terminate` function performs any operations that are required for "closing out" the actor after the enclosing graph has finished executing (e.g., deallocation of actor-specific memory or closing of associated files). The `enable` function and `invoke` function implement the *enable method* and *invoke method* of EIDF. These methods check for sufficient input data, and execute individual invocations (actor firings) for the given actor, respectively.

The separation of enable and invoke functionality in EIDF and LIDE leads to a number of useful features, which are discussed in [1, 4]. One such feature is the natural provision for a *guarded execution* primitive, which can be used by schedulers. If A is an EIDF actor, then a guarded execution of A at time t has no effect if A is not enabled at time t, and has the effect of firing A once if A is enabled at time t. This concept of guarded execution provides a useful primitive for the design of EIDF scheduling techniques, and their representation in terms of intuitive *schedule tree* structures [5].

For further details on the EIDF model and the LWDF programming methodology, we refer the reader to [1, 2].

In this initial release of LIDE, we target only the C language. However, based on the lightweight orientation of LIDE and the ease of retargeting its underlying design methods to different languages, we expect that additional language targets will be added in subsequent releases. In particular, LIDE adaptations for MATLAB, Verilog, and CUDA are currently in experimental pre-release stages, and are expected to evolve into subsystems within future releases of LIDE.

LIDE can be applied with the *DSPCAD Integrative Command Line Environment* (*DICE*) to provide an integrated framework for the design, implementation, and testing of DSP systems and their associated system components. With the assistance of the DICE unit testing framework, DSP systems can be tested efficiently both at the component level and the system level.

LIDE is being developed by the Maryland DSPCAD Research Group, which focuses on computer-aided design (CAD) techniques for DSP systems and applications. The overall organization for the LIDE package and its installation process follow the *DICE organizational conventions* for software packages [6].

2 Notational Conventions

The following notational conventions are used in this document.

- 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. This kind of notation is also used to represent placeholders in other design contexts (e.g., function arguments or segments of strings that need to be customized in context-specific ways).
- The special `<no arguments>` placeholder is used to emphasize that the associated command or function does not take any arguments.
- Items in command descriptions that are enclosed by square brackets (`[...]`) indicate *optional* command arguments.

3 LIDE Setup

3.1 LIDE Package Specification

LIDE can be downloaded from the LIDE Project Website (`http://www.ece.umd.edu/DSPCAD/projects/lide/lide.htm`). The installation and setup process for LIDE follows the *DICE organizational conventions*, which provide general conventions for organizing and installing software packages that are parameterized for easy adaptation across different packages. For details on the DICE organizational conventions, and the parameterized installation and setup steps associated with these conventions, we refer the reader to [6].

In terms of the DICE organizational conventions, the package parameter settings for LIDE are summarized as follows.

- Package name `<PK_NAME>` = `lide`.
- Package user directory: `<PK_USER_DIR>` = `lide_user`.
- Package download site: `<PK_SITE>` = Online Supplement [7].
- Package download file: `<PK_FILE>` = `lide.tar.gz`.
- Package definitions file: `<PK_DEFS>` = `uxdefs_lide`.
- Package startup file: `<PK_STARTUP>` = `lide_startup`.
- Package version command: `<PK_VERSION>` = `lideversion`.
- Package build command: `<PK_BUILD>` = `lidebuild`.
- Startup dependency list: `<PK_STARTUP_DEPS>` = {`dicemin`, DICE, `dicelang`}.
- Build dependency list: `<PK_BUILD_DEPS>` = {`dicemin`, `dicelang`, DICE}.

Essential UX definitions for LIDE: the meaning of the environment variable setting that is involved in `uxdefs_lide` is as follows.

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

Since LIDE is developed using the DICE package, and employs language-specific utilities provided by the `dicelang` package, DICE and `dicelang` must be installed before LIDE is set up. DICE and `dicelang` in turn depend on a simple package called `dicemin`, so one must first install `dicemin` before installing DICE and `dicelang`. Instructions on installing and using `dicemin`, `dicelang`, and DICE are provided in [6].

After installing `dicemin`, `dicelang`, and DICE, one can install the LIDE package using the parameterized setup, startup, and build processes associated with the DICE organizational conventions, along with the package parameter settings for LIDE defined above. When LIDE is built, the associated libraries of graph elements and run-time utilities will be compiled and installed in the LIDE installation directory.

3.2 Testing the Setup

As a basic test of the startup process and its preceding setup process, one can run the `lideversion` command, which takes no arguments, from the enclosing Bash session after LIDE has been started up.

If the LIDE package has been properly set up and started up, the `lideversion` command should execute and produce a (typically brief) message on standard output that gives the version number and other basic background information for the corresponding installation of LIDE.

After the LIDE package has been set up, plug-ins associated with a specific targeted language can be found in `lide/<language>`, where `<language>` is the targeted language. For example, the LIDE plug-in associated with the C language can be found in

```
lide/c
```

We refer to the integration of LIDE and a language-targeted plug-in as *LIDE-*`<language>`. For example, the integration of LIDE and the C-targeted plug-in is called *LIDE-C*.

3.3 LIDE Directory Organization

Within a language-targeted plug-in, the `src` directory contains associated source code — i.e., the source code that implements the plug-in. In this `src` directory, the directories `gems`, `demo`, and `runtime` contain, respectively, the associated graph elements (actor and edge implementations), application demonstrations, and runtime code — i.e., code that is part of the runtime environment in which LIDE applications execute. The name `gems` stands for *g*raph *e*le*m*ent*s*.

4 Introduction to DICELANG-C Utilities

The `dicelang-C` package, which is the C language plug-in within `dicelang`, provides a collection of Bash scripts that facilitates efficient software project development, implementation management, and testing using the C programming language. This section provides a brief introduction to some of the basic utilities available in `dicelang-C`. These utilities improve the efficiency with which one can create and work with C-based projects in LIDE.

4.1 Project Build

The C compiler that is used in `dicelang-C` is the *GNU Compiler Collection* (`gcc`) [8]. To configure utilities that are provided within `dicelang-C` for building (compiling and linking) C projects, we employ — in each source code directory of the project — a Bash file called `dlcconfig`, which can be viewed as a `dicelang-C`-specific project configuration file. For a project that employs `dicelang-C`, each source code directory should contain a `dlcconfig` file that contains project-specific configuration settings associated with that project directory. These settings associate values with certain variables that are used by `dicelang-C`.

The core set of variables that is relevant to `dlcconfig` files is summarized as follows.

- `dlcincludepath`: specifies zero or more *include paths* that are to to be passed to the compiler for compiling C files in the project directory. An empty string (`""`) setting corresponds to no include paths.
- `dlclibpath`: specifies zero or more paths to system libraries that are to be linked when building the target executable or library file associated with the project directory. An empty string (`""`) setting corresponds to no system libraries needed to be linked.
- `dlcmklibs`: specifies zero or more paths to user-specified library files that are to be linked when building the target executable or library file associated with the project directory. An empty string (`""`) setting corresponds to no library files needed to be linked.
- `dlctargetfile`: specifies the name of the target executable file or library file that is built by the project directory.

- `dlcinstalldir`: specifies the *installation directory* — that is, the directory where the target file for the associated project directory is to be installed.
- `dlcobjs`: specifies one or more *object code files* that correspond to the source code files that need to be compiled in the project directory.
- `dlcverbose`: specifies whether or not to display the current `dicelang-C` project configuration settings during compilation. A non-empty setting (i.e., a string that contains one or more characters) should be used to specify that such *verbose output* is desired.

Example 1 and Example 2 show examples of `dlcconfig` files for C-based project directories associated with individual actors in LIDE.

Example 1 An example of a `dlcconfig` file for a project in LIDE-C.

```
#!/usr/bin/env bash

dlcincludepath="-I. -I../common -I../../runtime"
dlcmklibs=""
dlctargetfile="lide_c_basic.a"
dlcinstalldir="$LIDECGEN"
dlcobjs="lide_c_add.o lide_c_block_add.o lide_c_file_source.o \
        lide_c_file_sink.o lide_c_table_lookup.o lide_c_switch.o lide_c_fifo.o"
dlcverbose=""
```

Example 2 Another example of a `dlcconfig` file.

```
#!/usr/bin/env bash

dlcincludepath="-I. -I../common -I../basic -I../../runtime"
dlclibpath="-lm"
dlcmklibs="$LIDECGEN/lide_c_basic.a $LIDECGEN/lide_c_runtime.a"
dlctargetfile="lide_c_imaging.a"
dlcinstalldir="$LIDECGEN"
dlcobjs="lide_c_bmp_file_read_halo.o lide_c_bmp_file_write.o \
        lide_c_file_source_fcv.o lide_c_gfilter.o lide_c_invert.o lide_c_t.o"
dlcverbose="true"
```

In LIDE, we employ the DICE convention of implementing the build functionality for a project directory within a Bash script named `makeme`. Here, by the *build functionality* of a project directory D, we mean the functionality to compile the source code in D and perform any additional steps needed to update the executables, library files, auto-generated documentation, etc. that are produced from the source files in D; for more details about such DICE project organization conventions, we refer the reader to [6] .

To implement the build functionality for a LIDE-C project directory, one should first create a `dlcconfig` file, as described above, and then invoke the `dicelang-C` utility `dlcmakeme` file by using the Bash `source` command from the `makeme` file. A `makeme` file constructed in this way is shown in Program 1.

Note the use of the `set` command at the beginning of the script in Program 1, and note also that no arguments are passed to `dlcmakeme`.

4.2 Building a Project Tree or Subtree

To build the entire project tree or subtree rooted at the current working directory, one can use the `dxmaketree` command. This command name is prefixed with `dx` instead of `dlc` since it is a DICE command rather than

Program 1 A `makeme` file that is constructed by following DICE conventions.

```
#!/usr/bin/env bash
# Script to build this project

# Export variable definitions and modifications
set -a

source dlcmakeme
```

a `dicelang-C` command. Indeed, the command can be used across arbitrary projects that are organized using DICE organizational conventions — not just C-based projects.

The general format for the `dxmaketree` command is as follows.

```
dxmaketree [-b]
```

This command builds the entire project tree that is rooted at the current working directory. This build is performed by calling all `makeme` scripts in the tree. By default, the `makeme` scripts are called through a top-down traversal of the tree (higher level directories have their `makeme` scripts executed before their respective subdirectories. If the `-b` option is used, then a bottom-up directory traversal is performed instead.

Directories that do not have `makeme` files are silently ignored during the top-down or bottom-up traversal.

4.3 Project Installation

After the project components in a `dicelang-C` project directory have been built, a target executable file (e.g., a `.exe` file) or a target library file (e.g., a `.a` file) is generated. Then one can use the `dlcinstall` command to install the generated file to the installation directory that is referenced by the `dlcinstalldir` variable, as defined in the `dlcconfig` file for the project directory. The general format for the `dlcinstall` command is as follows.

```
dlcinstall <no arguments>
```

To install project target files throughout the entire directory tree rooted at the current working directory, one can use the `dlcinstalltree` command. This command recursively traverses all directories rooted at the current working directory, and processes all directories that contain `dlcconfig` files. In particular, such directories are processed by running the associated `dlcconfig` scripts to set project variables, and then installing the associated target executable or library files in the associated installation directories. Throughout this recursive traversal, directories that do not contain `dlcconfig` files are ignored. The general format for the `dlcinstalltree` command is as follows.

```
dlcinstalltree <no arguments>
```

To list the contents of the installation directory that is associated with the current project directory, as determined by the `dlcconfig` file in that directory, one can use the `dlcinlist` command. The general format for this command is as follows.

```
dlcinlist <no arguments>
```

4.4 Project Cleanup

To remove standard types of generated files from a `dicelang-C` project directory, one can use the `dlcclean` command. In particular, files from the current working directory with `.d`, `.o`, `.exe`, and `.a` extensions are removed. This utility is typically used to remove intermediate files associated with the project build process, as well as the local copy of the target executable or library file after `dlcinstall` has been called. The general format for the `dlcclean` command is as follows.

```
dlcclean <no arguments>
```

To remove standard types of generated files from an entire `dicelang-C` project directory tree, one can use the `dlccleantree` command. This command recursively traverses the directory tree rooted at the current working directory, and invokes the `dlcclean` command, described above, in each visited directory. The general format for the `dlccleantree` command is as follows.

```
dlccleantree <no arguments>
```

4.5 `dicelang-C` Version Information

To display the version number of and other background information about the `dicelang-C` installation that is being used, one can use the `dlcversion` command. This command produces a message on standard output that gives the version number and other basic background information for the given installation of `dicelang-C`. The general format for the `dlcversion` command is as follows.

```
dlcversion <no arguments>
```

5 Introduction to LIDE-C

LIDE-C is the C-targeted plug-in of LIDE. In LIDE-C, graph elements and associated utilities are implemented using the C language and managed using the `dicelang-C` package. LIDE-C allows designers to easily create their own C-based actors, and to integrate arbitrary collections of LIDE-C actors (including combinations of LIDE-C library actors and user-designed LIDE-C actors) into fully functional signal processing subsystems, and applications.

In this preliminary LIDE-C release, there are three key components — `gems` (`lide/c/src/gems`), `runtime` (`lide/c/src/runtime`), and `demo` (`lide/c/demo`). These provide graph elements, runtime utilities, and demonstration applications, respectively. Details of these package components are described in the following subsections.

In LIDE-C, we follow the project organization conventions of `dicelang-C`, which are described in Section 4. In particular, each source code directory in LIDE-C contains a Bash file called `dlcconfig`, which provides `dicelang-C`-specific project directory configuration settings. For example, the `dlcconfig` file for the `lide/c/src/gems/basic` directory is shown in Example 1. In this `dlcconfig` file, note that the `LIDECGEN` variable provides the path of the target file installation directory for LIDE-C, which is used to store executable files and library files that are generated from the LIDE-C source code files. The value of `LIDECGEN` is set automatically upon startup of the LIDE-C package.

5.1 Dataflow Graph Elements

Graph elements in LIDE-C are designed and implemented as abstract data types (ADTs) that provide C-based, object-oriented of actors and FIFOs. In this design style, C header files (`.h` files) are used to specify and document application programming interfaces (APIs), which provide definitions that are exported to application developers, and C implementation files (`.c` files) are used to provide implementations for these APIs.

As discussed in Section 1, four interface functions — `construct`, `enable`, `invoke`, and `terminate` — are required to create an actor in LIDE-C. The general forms of the prototypes for these functions are shown in Program 2, Program 3, Program 4, and Program 5, respectively.

Program 2 Prototype format for the `construct` function of a LIDE-C actor.

```
lide_c_<actor name>_context_type *lide_c_<actor name>_new(<FIFO pointer list>,
        [parameter list]);
```

Program 3 Prototype format for the `enable` function of a LIDE-C actor.

```
boolean lide_c_<actor name>_enable (lide_c_<actor name>_context_type *context);
```

Program 4 Prototype format for the `invoke` function of a LIDE-C actor.

```
void lide_c_<actor name>_invoke (lide_c_<actor name>_context_type *context);
```

Designers can develop their own actors by appropriately specializing these prototype function templates, and applying the LWDF programming methodology to implement each of the corresponding four core actor interface functions. For details on the LWDF programming methodology, we refer the reader to [2]. For further reference on actor implementation in LIDE-C, many examples of practical LIDE-C-based actor implementations can be found in the `gems` source code directory of the LIDE-C package.

An example of a project directory within LIDE-C is the the `basic` subdirectory within `gems` (i.e., `lide/c/src/gems/basic`). This `basic` directory contains `dicelang-C` projects that implement graph elements for basic operations, including actors for addition, switching, and table lookup, as well as for a generic edge (FIFO). The `basic` directory can be viewed as a project that integrates a collection of graph elements into a single library, called `lide_c_basic.a`. To compile this project, build the target library file (`lide_c_basic.a`), and install this file to the target file installation directory (`$LIDECGEN`), one can run the following command sequence.

```
./makeme
dlcinstall
```

This simple command sequence illustrates how a library (or other project) can be built and installed using `dicelang-C`. Users will typically not need to use this sequence for `lide_c_basic.a` (or for other libraries or executables in LIDE-C) since such libraries are built and installed as part of the LIDE-C setup and installation process. However, the structure of this example may be useful as a template for users to develop their own `dicelang-C`-based projects, such as LIDE-C projects that involve user-defined actors and applications.

The `common` directory within the `gems` directory stores header files that provide fundamental definitions used for implementing and working with dataflow graph elements. These definitions include APIs for `enable` and `invoke` functions of LIDE-C actors; definitions that provide standard run-time context information for actor executions; and definitions of various utility functions.

5.2 Run-time Utilities

The `runtime` project directory (`lide/c/src/runtime`) contains utility functions that are to be used by and linked into applications that are implemented using LIDE-C. These include simple functions that provide standardized interfaces for opening files and allocating memory, as well as a simple scheduling strategy for executing the actors in a dataflow graph. This scheduling strategy is an adaptation of the *canonical scheduling strategy* of the *functional DIF* environment [1]. The implementation of this scheduler in LIDE-C can also be used as a reference for developing and interfacing more sophisticated schedulers within LIDE-C applications.

5.3 Demo Applications

The `demo` project directory (`lide/c/demo`) contains demonstration applications for LIDE-C. These applications demonstrate the use of dataflow graph elements and run-time utility functions that have been developed

Program 5 Prototype format for the `terminate` function of a LIDE-C actor.

```
void lide_c_<actor name>_terminate (lide_c_<actor name>_context_type *context);
```

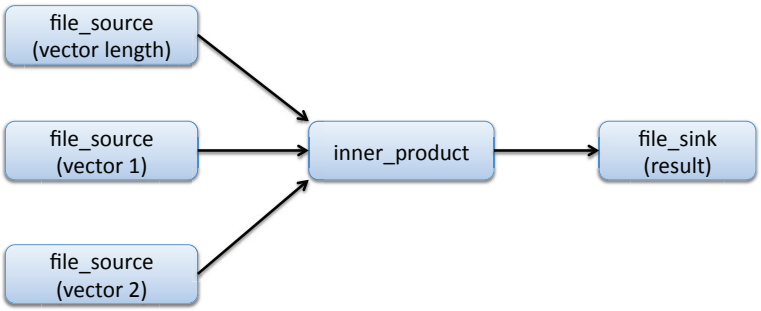

Figure 1: Dataflow graph for the inner product computation example.

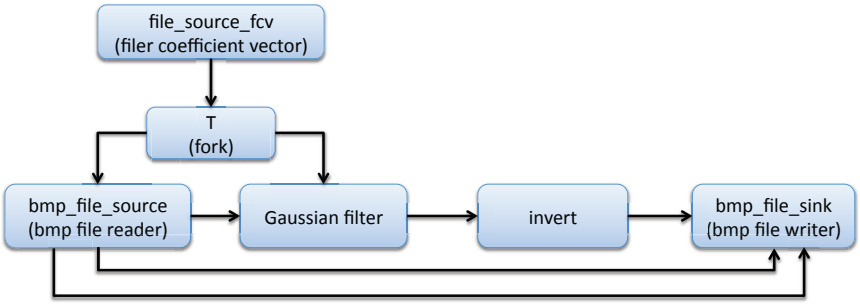

Figure 2: Dataflow graph for the Gaussian filtering example.

using LIDE-C. For each demo application, a driver program is provided. This driver program instantiates graph elements to construct the corresponding application graph, and invokes a scheduler to coordinate execution of the graph.

Three demo examples are provided. These include a basic *inner product computation* example —

`lide/c/demo/basic/inner_product`;

Gaussian filtering for image processing —

`lide/c/demo/imaging/gaussian_filtering`;

and *channel estimation* for wireless communication —

`lide/c/demo/communication/channel_estimation`.

The inner product example computes the inner product of two vectors, where the vector length can be configured at run-time. The Gaussian filtering example denoises an image based on a two-dimensional filter whose impulse response is a Gaussian curve. The channel estimation example estimates channel responses for data subcarriers in a wireless communication receiver.

Fig. 1, Fig. 2, and Fig. 3 show the dataflow graphs for the inner product computation, Gaussian filtering, and wireless channel estimation applications, respectively.

For each demo application, a driver program is provided in the associated `util` subdirectory to instantiate and execute the corresponding dataflow graph. The driver programs execute the applications based on the

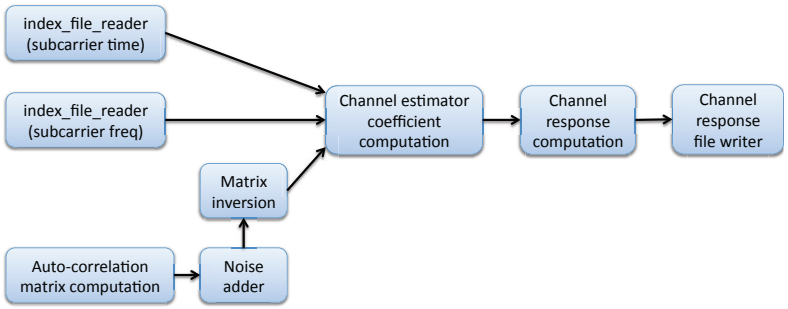

Figure 3: Dataflow graph for the wireless channel estimation example.

simple EIDF/CFDF canonical scheduling strategy [1]. An implementation of this scheduling technique is provided in the `runtime` directory.

For each demo application, several test subdirectories are provided. These subdirectories contain test inputs and expected outputs, which can be exercised automatically using the test suite management features of DICE [9]. One can enter a "test demo" subdirectory to compile an associated driver program that is stored in the `util` subdirectory, and build the executable (`.exe`) file by running the following command.

```
./makeme
```

To run a simulation of a demo application and examine the output results, one can run the following command.

```
./runme
```

An example of such a test demo directory is

`lide/c/demo/basic/inner_product/test_demo1`,

and the corresponding `util` directory is

`lide/c/demo/basic/inner_product/util`.

5.4 Creating Driver Programs

In this section, we describe a general design procedure for creating LIDE-C driver programs that construct and execute dataflow graphs based on the LWDF programming methodology. This design procedure is summarized by the following steps.

1. Declare an array of pointers, where these pointers are to point to the actor contexts for all of the actors in the application dataflow graph.

2. Declare pointer variables that are to point to the FIFOs (edges) in the dataflow graph.

3. Declare other required variables.
4. Create the dataflow graph FIFOs by allocating memory for them with the desired capacities and token sizes, and setting the FIFO pointer variables (see Step 2) to point to the corresponding blocks of allocated memory.
5. Create the dataflow graph actors by constructing them with appropriate parameter settings and connections to the appropriate FIFOs. Set the elements of the actor pointer array (see Step 1) to point to the corresponding blocks of allocated memory.
6. Execute the graph by calling a pre-defined or user-defined scheduler.

An example that follows and illustrates this design procedure can be found in

`$UXLIDEC/demo/basic/inner_product/util/lide_c_inner_product_driver.c`

5.5 Creating User-designed Schedulers

In addition to using the pre-defined scheduler that is provided within the `runtime` directory in LIDE-C, designers can design and implement their own schedulers and integrate them within driver programs to coordinate execution of LIDE-based dataflow graphs. General guidelines for implementing and interfacing such a user-defined scheduler are summarized as follows.

1. Use the actor pointers that have been defined for the enclosing driver program (See Section 5.4).
2. Create actor descriptors as strings for diagnostic output.
3. If an actor is to be executed by the user-defined scheduler with guarded execution (see Section 1), call the following LIDE-C runtime API function to perform the actor execution.

   `lide_c_util_guarded_execution(<actor pointer>, <actor descriptor>)`

   The function call should be made with the appropriate actor pointer and actor descriptor (string) pointer as arguments.
4. If an actor will be executed without guarded execution (i.e., an unconditional actor execution), then call the `invoke` function associated with the actor, and pass the actor pointer as an argument to this `invoke` function.

   When such unconditional actor execution is used, some form of scheduling analysis should be employed by the designer to ensure that sufficient data will always be available for the associated actor firing. Otherwise, the actor may execute on invalid data and result in incorrect application behavior.
5. Determine how actor executions are to be ordered by scheduler, and perform calls to the API functions for guarded execution (Guideline 3) or unconditional actor execution (Guideline 4) as this ordering process is carried out.

These guidelines for implementing user-defined schedulers can also be applied in software synthesis processes to automate the generation of scheduling code, and ensure proper linkage of such schedules with code for carrying out executions of specific actors.

An example of a hand-written, user-fined scheduler for a simple LIDE-C application example is shown in Program 6.

5.6 Summary of Graph Elements

The collection of graph elements (`gems`) that is provided in LIDE-C is summarized in Fig. 4. We are actively extending this collection with additional actors that we plan to include in future releases.

Users of LIDE-C can create application dataflow graphs by using the pre-defined graph elements in LIDE-C, by using their own, user-defined graph elements (see Section 5.1), or by using a combination of pre-defined and user-defined components.

Program 6 Example code for a simple, hand-written scheduler.

```
/* Header files that are included in the driver program */
#include "lide_c_down_sampler.h"
#include "lide_c_up_sampler.h"
#include "lide_c_util.h"

/* Actor pointers that are defined in the driver program. */
lide_c_actor_context_type* down_sampler;
lide_c_actor_context_type* up_sampler;

/* Actor descriptors. */
char* down_sample_descriptor = "Down Sampler";
char* up_sample_descriptor = "Up Sampler";

/* Call the API functions with or without guarded execution according to
   the desired scheduling order.
 */
lide_c_util_guarded_execution(down_sampler, down_sample_descriptor);
up_sampler->invoke(up_sampler);
```

Library Name	Brief Description
basic	Basic operations.
communication	Digital communication.
dsp	Fundamental signal processing operations.
hep	High-energy physics.
imaging	Image processing.

Figure 4: Summary of the pre-defined graph elements provided in LIDE-C.

6 Summary

This report has provided a motivation and overview of LIDE, the DSPCAD Lightweight Dataflow Environment. LIDE is a flexible, lightweight design environment that contains libraries of dataflow graph elements and utilities for dataflow-based application design. LIDE allows designers to experiment with dataflow-based approaches for design and implementation of DSP systems, including associated methods for application modeling, simulation, and implementation.

Actor design in LIDE is based on the semantics of the Enable-Invoke Dataflow (EIDF) model of computation, and the Lightweight Dataflow (LWDF) programming methodology, which provides a systematic and retargetable (language-independent) method to implement dataflow actors and graphs based on the EIDF model. This report has also provided introductions to some of the general utility commands available in `dicelang-C`, as well as to the unit testing features provided in DICE.

For more details on EIDF and LWDF, we refer the reader to [1, 4, 2]. 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 [10], and the DICE User's Guide [9].

7 Acknowledgment

This work was sponsored in part by the Laboratory for Physical Science, Laboratory for Telecommunication Sciences, and US National Science Foundation.

References

[1] 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.

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

[3] S. S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, Eds., *Handbook of Signal Processing Systems*. Springer, 2010.

[4] S. S. Bhattacharyya, W. Plishker, N. Sane, C. Shen, and H. Wu, "Modeling and optimization of dynamic signal processing in resource-aware sensor networks," in *Proceedings of the Workshop on Resources Aware Sensor and Surveillance Networks in conjunction with IEEE International Conference on Advanced Video and Signal-Based Surveillance*, Klagenfurt, Austria, August 2011, pp. 449–454.

[5] W. Plishker, N. Sane, and S. S. Bhattacharyya, "Mode grouping for more effective generalized scheduling of dynamic dataflow applications," in *Proceedings of the Design Automation Conference*, San Francisco, July 2009, pp. 923–926.

[6] 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," Institute for Advanced Computer Studies, University of Maryland at College Park, Tech. Rep. UMIACS-TR-2011-13, 2011, http://hdl.handle.net/1903/11804.

[7] "LIDE overview online supplement," http://www.ece.umd.edu/DSPCAD/projects/lide/guide/supplement.htm.

[8] M. Loukides and A. Oram, *Programming with GNU Software*. O'Reilly & Associates, Inc., 1997, iSBN 1565921127.

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

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