The DSPCAD Framework for Modeling and Synthesis of Signal Processing Systems

36

Shuoxin Lin, Yanzhou Liu, Kyunghun Lee, Lin Li, William Plishker, and Shuvra S. Bhattacharyya

Abstract

With domain-specific models of computation and widely-used hardware acceleration techniques, Hardware/Software Codesign (HSCD) has the potential of being as agile as traditional software design, while approaching the performance of custom hardware. However, due to increasing use of system heterogeneity, multi-core processors, and hardware accelerators, along with traditional software development challenges, codesign processes for complex systems are often slow and error prone. The purpose of this chapter is to discuss a Computer-Aided Design (CAD) framework, called the DSPCAD Framework, that addresses some of these key development issues for the broad domain of Digital Signal Processing (DSP) systems. The emphasis in the DSPCAD Framework on supporting cross-platform, domain-specific approaches enables designers to rapidly arrive at initial implementations for early feedback, and then systematically refine them towards functionally correct and efficient solutions. The DSPCAD Framework is centered on three complementary tools – the Data-flow Interchange Format (DIF), LIghtweight Data-flow Environment (LIDE) and DSPCAD Integrative Command Line Environment (DICE), which support flexible design experimentation and orthogonalization across three major dimensions in model-based DSP system design – abstract data-flow models, actor implementation languages, and integration with platform-specific design tools. We demonstrate the utility

S. Lin (✉) • Y. Liu • K. Lee • L. Li • W. Plishker
Department of Electrical and Computer Engineering, University of Maryland, College Park, MD, USA
e-mail: slin07@umd.edu; yzliu@umd.edu; leekh3@umd.edu; lli12311@umd.edu; plishker@umd.edu

S.S. Bhattacharyya
Department of Electrical and Computer Engineering and Institute for Advanced Computer Studies, University of Maryland, College Park, MD, USA

Department of Pervasive Computing, Tampere University of Technology, Tampere, Finland
e-mail: ssb@umd.edu

of the DSPCAD Framework through a case study involving the mapping of synchronous data-flow graphs onto hybrid CPU-GPU platforms.

Acronyms

ADT	Abstract Data Type
API	Application Programming Interface
BDF	Boolean Data Flow
BPSK	Binary PSK
CAD	Computer-Aided Design
CAL	Cal Actor Language
CFDF	Core Functional Data Flow
CPU	Central Processing Unit
CSDF	Cyclo-Static Data Flow
CUDA	Compute Unified Device Architecture
D2H	Device-to-Host
DICE	DSPCAD Integrative Command Line Environment
DIF	Data-flow Interchange Format
DSP	Digital Signal Processing
FCFS	First-Come First-Serve
FIFO	First-In First-Out
FIR	Finite Impulse Response
FPGA	Field-Programmable Gate Array
FSM	Finite-State Machine
GLV	Graph-Level Vectorization
GPU	Graphics Processing Unit
H2D	Host-to-Device
HDL	Hardware Description Language
HSCD	Hardware/Software Codesign
ITS	Individual Test Subdirectory
LIDE	LIghtweight Data-flow Environment
MDSDF	Multi-Dimensional Synchronous Data Flow
MILP	Mixed Integer Linear Programming
PREESM	Parallel and Real-time Embedded Executives Scheduling Method
PSDF	Parameterized Synchronous Data Flow
PSK	Phase Shift Keying
PSM	Parameterized Sets of Modes
QAM	Quadrature Amplitude Modulation
QPSK	Quadrature PSK
RVC	Reconfigurable Video Coding
SADF	Scenario-Aware Data Flow
SDF	Synchronous Data Flow
SDR	Software Defined Radio
SDTC	Scheduling and Data Transfer Configuration
SysteMoC	SystemC Models of Computation

VF	Vectorization Factor
WSDF	Windowed Synchronous Data Flow

Contents

36.1 Introduction

Software design processes have evolved rapidly over the past two decades. In many areas, agile programming [1] has shown how software development benefits from going to implementation quickly. By writing core functionality for key use cases, software engineers can gain early feedback from real implementations, and, thereby, features, performance, and platforms may be refined effectively and quickly. Hardware/Software Codesign (HSCD) stands to inherit these same benefits from agile design but in practice has not kept pace with traditional software development evolution. Domain-specific models and languages that support fast application descriptions already exist. However, compared to traditional software, Hardware/Software tools to translate those descriptions to implementations are inherently more complex. They must deal with traditional software development issues, as well as system heterogeneity, multiple cores, and hardware accelerators. Because of the diversity of applicable tools and approaches, many of the steps are manual, ad hoc, or platform specific.

The purpose of this chapter is to discuss a Computer-Aided Design (CAD) framework for Digital Signal Processing (DSP) applications, called the DSPCAD Framework, that addresses some of these key development issues for the broad domain of DSP. The DSPCAD Framework achieves this by establishing a cross-platform, domain-specific approach that enables designers to arrive at initial implementations quickly for early feedback, and then systematically refine them toward functionally correct and high-performance solutions. The keys to such an approach include (a) lightweight design principles, which can be applied relatively quickly and flexibly in the context of existing design processes and (b) software techniques and tools that are grounded in data-flow models of computation.

36.1.1 Data Flow

Data-flow models have proven invaluable for DSP system design. Their graph-based formalisms allow designers to describe applications in a natural yet semantically rigorous way. As a result, data-flow languages are increasingly popular. Their diversity, portability, and intuitive design have extended them to many application areas and platform types within the broad DSP domain (e.g., see [3]). Modeling applications through coarse-grain data-flow graphs is widespread in the DSP design community, and a variety of data-flow models of computation have been developed for DSP system design.

Common to each of these modeling paradigms is the representation of computational behavior in terms of data-flow graphs. In this context of DSP system design, a data-flow graph is a directed graph $G = (V, E)$ in which each vertex (*actor*) $v \in V$ represents a computational task, and each edge $e \in E$ represents First-In First-Out (FIFO) communication of data values (*tokens*) from the actor $src(e)$ at the source of e to the actor $snk(e)$ at the sink of e. Data-flow actors execute in terms of discrete units of execution, called *firings*, which produce and consume tokens from the incident edges. When data-flow graphs are used for behavioral modeling of DSP systems, the graph represents application functionality with minimal details pertaining to implementation. For example, how the FIFO communication associated with each edge is mapped into and carried out through physical storage, and how the execution of the actors is coordinated are implementation-related details that are not part of the data-flow graph representation. Such orthogonalization between behavioral aspects and key implementation aspects is an important feature of data-flow-based DSP system design that can be leveraged in support of agile design processes. For a detailed and rigorous treatment of general principles of data-flow modeling for DSP system design, we refer the reader to [31], and for discussion on the utility of orthogonalization in system-level design, we refer the reader to [28].

36.1.2 Data-Flow Modeling Variants

A distinguishing aspect of data-flow modeling for DSP system design is the emphasis on characterizing the rates at which actors produce and consume tokens from their incident edges, and the wide variety of different variants of data-flow models of computation that has evolved, due in large part to different assumptions and formulations involved in these *data-flow rates* (e.g., see [3, 49]). For example, Synchronous Data Flow (SDF) is a form of data flow in which each actor consumes a constant number of tokens from each input port and produces a constant number of tokens on each output port on every firing [30]. SDF can be viewed as an important common denominator that is supported in some fashion across most data-flow-based DSP design tools, and a wide variety of techniques for analyzing SDF graphs and deriving efficient implementations from them has been developed (e.g., see [3]). However, the restriction to constant-valued data-flow rates limits the applicability of the SDF model. This has led to the study of alternative data-flow models that provide more flexibility in specifying inter-actor communication. Examples of such models include Boolean Data Flow (BDF), Core Functional Data Flow (CFDF), Cyclo-Static Data Flow (CSDF), Multi-Dimensional Synchronous Data Flow (MDSDF), Parameterized Synchronous Data Flow (PSDF), Scenario-Aware Data Flow (SADF), and Windowed Synchronous Data Flow (WSDF) [6, 7, 9, 27, 34, 43, 51].

36.1.3 DSPCAD Framework

The DSPCAD Framework is a CAD framework that helps designers to apply the formalisms of the data-flow paradigm in DSP-oriented, HSCD processes. The DSP-CAD Framework is specifically oriented toward flexible and efficient exploration of interactions and optimizations across different signal processing application areas (e.g., speech processing, specific wireless communication standards, cognitive radio, and medical image processing), alternative data-flow models of computation (e.g., Boolean Data Flow (BDF), Core Functional Data Flow (CFDF), etc., as listed in Sect. 36.1.2), and alternative target platforms along with their associated platform-based tools (e.g., field programmable gate arrays, graphics processing units, programmable digital signal processors, and low-power microcontrollers).

The DSPCAD Framework is based on three complementary subsystems, which respectively provide a domain-specific modeling environment for experimenting with alternative, DSP-oriented data-flow modeling techniques; a lightweight, cross-platform environment for implementing DSP applications as data-flow graphs; and a flexible project development tool that facilitates DSP system integration and validation using different kinds of platform-based development tools. These subsystems of the DSPCAD Framework are called, respectively, the Data-flow Interchange Format (DIF), LIghtweight Data-flow Environment (LIDE) and DSPCAD Integrative Command Line Environment (DICE). While DIF, LIDE, and DICE can be used independently as stand-alone tools, they offer significant synergy when applied together for HSCD. The DSPCAD Framework is defined by such integrated use of these three complementary tools.

In the remainder of this section, we provide brief overviews of DIF, LIDE, and DICE. We cover these tools in more detail in Sects. 36.3, 36.4, and 36.5, respectively. Then in Sect. 36.6, we demonstrate their integrated use in the DSPCAD Framework to develop a platform-specific data-flow framework for mapping SDF graphs into Graphics Processing Unit (GPU) implementations. This case study is presented to concretely demonstrate the DSPCAD Framework and its capability to derive specialized data-flow tools based on specific data-flow modeling techniques and target platforms. In Sect. 36.7, we summarize the developments of this chapter and discuss ongoing directions of research in the DSPCAD Framework.

DIF – DIF provides application developers an approach to application specification and modeling that is founded in data-flow semantics, accommodates a wide range of specialized data-flow models of computation, and is tailored for DSP system design [21, 22].

DIF is comprised of a custom language that provides an integrated set of syntactic and semantic features that capture essential modeling information of DSP applications without over-specification. DIF also includes a software package for reading, analyzing, and optimizing applications described in the language. Additionally, DIF supports mixed-grain graph topologies and hierarchical design in specification of data-flow related, subsystem- and actor-specific information. The data-flow semantic specification is based on data-flow modeling theory and independent of any specialized design tool.

DIF serves as a natural design entry point for reasoning about a new application or class of applications and for experimenting with alternative approaches to modeling application functionality. LIDE and DICE complement these abstract modeling features of DIF by supporting data-flow-based implementations on specific platforms.

LIDE – LIDE is a flexible, lightweight design environment that allows designers to experiment with data-flow-based implementations directly on customized programmable platforms. LIDE is "lightweight" in the sense that it is based on a compact set of application programming interfaces that can be retargeted to different platforms and integrated into different design processes relatively easily.

LIDE contains libraries of data-flow graph elements ("gems"), as described in Sect. 36.1.1, and utilities that assist designers in modeling, simulating, and implementing DSP systems using formal data-flow techniques. Here, by gems, we mean actor and edge implementations. The libraries of data-flow gems (mostly actor implementations) 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.

Schedules for LIDE-based implementations can be created directly by designers using LIDE Application Programming Interfaces (APIs) or synthesized by DIF, decreasing the time to initial implementation. Refinements based on initial implementations may occur at the data-flow level (e.g., using DIF) or at the schedule implementation or gems level with LIDE, giving an application developer an opportunity to efficiently refine designs in terms of performance or functionality.

DICE – DICE is a package of utilities that facilitates efficient management of software projects. Key areas of emphasis in DICE are cross-platform operation, support for model-based design methodologies, support for projects that integrate heterogeneous programming languages, and support for applying and integrating different kinds of design and testing methodologies. The package facilitates research and teaching of methods for implementation, testing, evolution, and revision of engineering software. The package is a foundation for developing experimental research software for techniques and tools in the area of DSP systems. The package is cross-platform, supporting Linux, Mac OS, Solaris, and Windows (equipped with Cygwin) platforms. By using LIDE along with DICE, designers can efficiently create and execute unit tests for user-designed actors.

36.2 Related Work

In this section, we review a number of representative data-flow-based tools that are applied to modeling, simulation, and synthesis of DSP systems. The intent in this review is not to be comprehensive but rather to provide a sampling of representative, research-oriented data-flow-based tools that are relevant to DSP system design. We also summarize distinguishing aspects of the DSPCAD Framework in relation to the state of the art in data-flow research for DSP. For broader and deeper coverage of different data-flow-based design tools and methodologies, we refer the reader to [3].

36.2.1 Representative Tools

Parallel and Real-time Embedded Executives Scheduling Method (PREESM) is an Eclipse-based code generation tool for signal processing systems [37,41]. PREESM provides architecture modeling and scheduling techniques for multi-core digital signal processors. In PREESM, applications are modeled as a hierarchical extension of SDF called an algorithm graph, while the targeted architectures are modeled as architecture graphs, which contain interconnections of abstracted processor cores, hardware coprocessors, and communication media. PREESM then takes the algorithm graph, architecture graph, and application parameters and constraints as its inputs to automatically generate software implementations on multi-core programmable digital signal processors.

The multi-processor scheduler in PREESM is based on the List and Fast Scheduling methods described by Kwok [29]. A randomized version of the List Scheduling method is first applied to return the best solution observed during a designer-determined amount of time. The obtained best solution can be applied directly for software synthesis or be used to initialize the population of a genetic algorithm for further optimization. The capabilities of PREESM are demonstrated, for example, by the rapid prototyping of a state-of-the-art computer vision application in [38].

SystemC Models of Computation (SysteMoC) is a SystemC-based library that facilitates data-flow-based HSCD for DSP systems. Actor design in SysteMoC is based on a model that includes a set of functions and an actor Finite-State Machine (FSM). The set of functions is partitioned into *actions*, which are used for data processing and *guards*, which are used to check for enabled transitions in the actor FSM. In [19], an MPEG-4 decoder application is provided as a case study to demonstrate the capability of SysteMoC to support system synthesis as well as design space exploration for HSCD processes. For more details about SysteMoC, we refer the reader to ▶ Chap. 3, "SysteMoC: A Data-Flow Programming Language for Codesign".

Cal Actor Language (CAL) is a data-flow programming language that can be applied to develop hardware and software implementations [11]. Like designs in SysteMoC, CAL programs incorporate an integration of data flow and state machine semantics. Actor specification in CAL includes actions, guards, port patterns, priorities, and transitions between actions. Thus, data-flow actor design in CAL is similar to that in SysteMoC and (as we will see in Sect. 36.4) LIDE in terms of an underlying, state-machine-integrated, data-flow model of computation. A major advance provided by CAL has been through its use in a recent MPEG standard for Reconfigurable Video Coding (RVC) [25].

36.2.2 Distinguishing Aspects of the DSPCAD Framework

Perhaps the most unique aspects of the DSPCAD Framework compared to other data-flow tools such as PREESM, SysteMoC, and CAL are the (1) emphasis on orthogonalization across three major dimensions in model-based DSP system design – abstract data-flow models, actor implementation languages, and integration with platform-specific design tools – and (2) support for a wide variety of different data-flow modelings styles. Feature 1 here is achieved in the DSPCAD Framework through the complementary objectives of DIF, LIDE, and DICE, respectively.

Support for Feature 2 in the DSPCAD Framework is threefold. First, DIF is agnostic to any particular data-flow model of computation and is designed to support a large and easily extensible variety of models. Second, LIDE is based on a highly expressive form of data-flow CFDF, which is useful as a common model for working with and integrating heterogeneous data-flow models of computation. This is because various specialized forms of data flow can be formulated as special cases of CFDF (e.g., see [44]). More details about CFDF are discussed in Sect. 36.3.1. Third, LIDE contains flexible support for parameterizing data-flow actors and manipulating actor and graph parameters dynamically. This capability is useful for experimenting with various parametric data-flow concepts, such as PSDF, and parameterized and interfaced data-flow [9] meta model, and the hierarchical reconfiguration methodologies developed in the Ptolemy project [35].

The DSPCAD Framework can be used in complementary ways with other DSP design environments, such as those described above. The modularity and specialized areas of emphasis within DIF, LIDE, and DICE make each of these component tools useful for integration with other design environments. For example, DIF has been employed as an intermediate representation to analyze CAL programs and derive statically schedulable regions from within dynamic data-flow specifications [16], and, in the PREESM project, the CFDF model of computation employed by

LIDE has been used to represent dynamic data-flow behavior for applying novel architectural models during design space exploration [39].

Although the DSPCAD Framework is not limited to any specific domain of signal processing applications, the components of the framework have been applied and demonstrated most extensively to date in the areas of wireless communications, wireless sensor networks, and embedded computer vision. For elaboration on HSCD topics in these latter two domains, we refer the reader to ▸ Chap. 38, "Wireless Sensor Networks" and ▸ Chap. 40, "Embedded Computer Vision" respectively.

36.3 Data-Flow Interchange Format Overview

DIF provides a model-based design environment for representing, analyzing, simulating, and synthesizing DSP systems. DIF focuses on data-flow graph modeling and analysis methods where the details of actors and edges of a graph are abstracted in the form of arbitrary actor and edge attributes. In particular, implementation details of actors and edges are not specified as part of DIF representations.

The DIF environment is composed of the DIF language and the DIF package. The DIF language is a design language for specifying mixed-grain data-flow models for DSP systems. The DIF package, a software package that is built around the DIF language, contains a large variety of data-flow graph analysis and transformation tools for DSP application models that are represented in DIF. More specifically, the DIF package provides tools for (1) representing DSP applications using various types of data-flow models, (2) analyzing and optimizing system designs using data-flow models, and (3) synthesizing software from data-flow graphs. The software synthesis capabilities of DIF assume that actor implementations are developed separately (outside of the DIF environment) and linked to their associated actor models as synthesis-related attributes, such as the names of the files that contain the actor implementation code.

Unlike most data-flow-based design environments, which are based on some forms of static data-flow model or other specialized forms of data flow, DIF is designed specifically to facilitate formal representation, interchange, and analysis of different kinds of data-flow models and to support an extensible family of both static and dynamic data-flow models. Models supported in the current version of DIF include SDF [30], CSDF [6], MDSDF [34], BDF [7], PSDF [2], and CFDF. DIF also provides various analysis, simulation, and synthesis tools for CFDF models and its specialized forms. As motivated in Sect. 36.2, CFDF is useful as a common model for working with and integrating heterogeneous data-flow models of computation [44], which makes it especially useful for the purposes of the DIF environment. Examples of data-flow tools within the DIF package are tools for CFDF functional simulation [43], SDF software synthesis for programmable digital signal processors [23], and quasi-static scheduling from dynamic data-flow specifications [16, 42]. Due to the important role of CFDF in DIF, we introduce background on CFDF in the following section.

36.3.1 Core Functional Data Flow

CFDF is a dynamic data-flow model of computation in which the behavior of an actor A is decomposed into a set of modes $modes(A)$. Each firing of A is associated with a specific mode in $modes(A)$. For each mode $m \in modes(A)$, the data-flow rates (numbers of tokens produced or consumed) for all actor ports are fixed. However, these rates can vary across different modes, which allows for the modeling of dynamic data-flow behavior.

When a CFDF actor A fires in a particular mode m, it produces and consumes data from its incident ports based on the constant production and consumption rates associated with m, and it also determines the *next mode* $z \in modes(A)$ for the actor, which is the mode that will be active during the next firing of A. The next mode may be determined statically as a property of each mode or may be data dependent. Combinations of data-dependent next mode determination and heterogeneous data-flow rates across different modes can be used to specify actors that have different kinds of dynamic data-flow characteristics.

A CFDF actor has associated with it two computational functions, called the *enable* and *invoke* functions of the actor. These functions provide standard interfaces for working with the actor in the context of a schedule for the enclosing data-flow graph. The enable function for a given actor A returns a Boolean value that indicates whether or not there is sufficient data on the input edges and sufficient empty space on the output edges to accommodate the firing of A in its next mode.

The invoke function of an actor, on the other hand, executes the actor according to its designated next mode and does so without any use of blocking reads or writes on actor ports – that is, data is consumed and produced without checking for availability of data or empty space, respectively. It is assumed that these checks will be performed (a) either statically, dynamically (using the enable method), or using a combination of static and dynamic techniques and (b) before the associated firings are dispatched with the invoke function. Thus, overhead or reduced predictability due to such checking need not be incurred during execution of the invoke function. This decomposition of actor functionality into distinct enable and invoke functions can be viewed as a formal separation of concerns between the checking of an actor's fireability conditions and execution of the core processing associated with a firing.

Various existing data-flow modeling techniques, including SDF, CSDF, and BDF, can be formulated as special cases of CFDF [44]. For further details on CFDF semantics, we refer the reader to [43, 44].

36.3.2 Reconfigurable Modulator Example

Here, we present a practical application as an example of CFDF modeling. Figure 36.1a shows a dynamically reconfigurable modulator application (*RMOD*) that supports multiple source rates and multiple Phase Shift Keying (PSK) and Quadrature Amplitude Modulation (QAM) schemes. Actor C reads two run-time

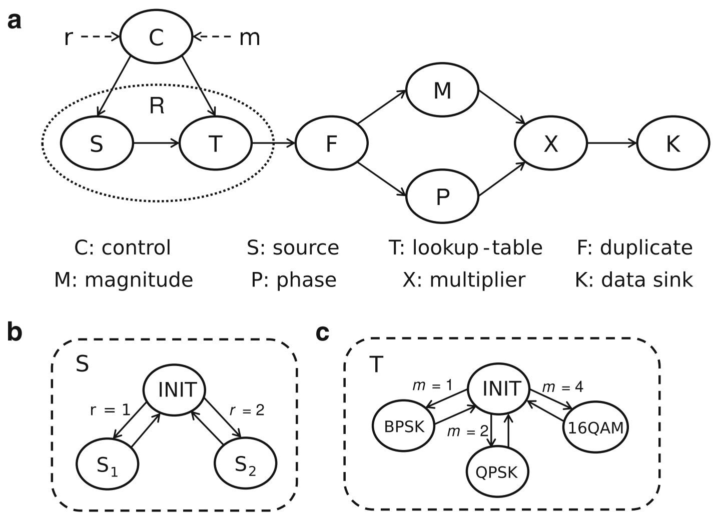


Fig. 36.1 CFDF modeling of a reconfigurable modulator (RMOD) application supporting multiple source data rate and modulation schemes. (**a**) CFDF model of the RMOD application (**b**) Mode transitions of actor S. (**c**) Mode transitions of actor T

a

Mode	Edge:C→S	Edge:S→T
INIT	−1	0
S_1	0	1
S_2	0	2

b

Mode	Edge:C→T	Edge:S→T	Edge:T→F
INIT	0	−1	0
BPSK	−1	0	1
QPSK	−2	0	1
16QAM	−4	0	1

Fig. 36.2 Data-flow tables. (**a**) Table for actor S. (**b**) Table for actor T

parameters, r and m, corresponding to the source data rate and modulation scheme, respectively, and sends these parameter values to the actors S and T. S and T in turn are two CFDF actors that each have multiple modes and data-dependent mode transitions, as illustrated in Figs. 36.1b and c, respectively.

Both S and T are initialized to begin execution in their respective INIT modes. In its INIT mode, S reads the source data rate r and switches to either S_1 or S_2 depending on the value of r. Similarly, in its INIT mode, T reads the modulation scheme index m and switches to one of the 3 modes, Binary PSK (BPSK), Quadrature PSK (QPSK), or 16-QAM, depending on m. S and T have different production and consumption rates in different modes.

Figure 36.2 shows the *data-flow tables* for actors S and T. A data-flow table Z for a CFDF actor A specifies the data-flow behavior for the available modes in the actor. Each entry $Z[\mu, p]$ corresponds to a mode $\mu \in \mathit{modes}(A)$ and input or output port p of A. If p is an output port of A, then $Z[\mu, p]$ gives the number of tokens produced on the edge connected to p during a firing of A in mode μ. Similarly, if

p is an input port, then $Z[\mu, p] = -c$, where c is the number of tokens consumed during mode μ from the edge connected to p.

In the column headings for the data-flow tables shown in Fig. 36.2, each port is represented by the edge that is connected to the port. If $m = 1$, then T executes in the BPSK mode and consumes only 1 token on its input edge. On the other hand, if $m = 4$, then T executes in the 16-QAM mode and consumes 4 tokens on its input edge. After firing in their respective BPSK or 16-QAM modes, S and T switch back to their INIT modes and await new values of r and m for the next round of computation. The remaining actors are SDF actors that consume/produce a single token on each of their input/output edges every time they fire.

36.3.3 Data-Flow Graph Specification in the DIF Language

As discussed above, the DIF language is a design language for specifying mixed-grain data-flow models in terms of a variety of different forms of data flow [22]. The DIF language provides a C-like, textual syntax for human-readable description of data-flow structure. An XML-based version of the DIF language, called *DIFML*, is also provided for structured exchange of data-flow graph information between different tools and formats [17]. DIF is based on a block-structured syntax and allows specifications to be modularized across multiple files through integration with the C preprocessor. As an example, a DIF specification of the RMOD application is shown in Listing 1.

Listing 1 DIF Language specification of the RMOD application

```
CFDF RMOD {
  topology {
    nodes = C, S, T, F, M, P, X, K;
    edges = e1(C, S), e2(C, T), e3(S, T), e4(T, F),
      e5(F, M), e6(F, P), e7(M, X), e8(P, X), e9(X, K);
  }
  actor C {
    name = "mod_ctrl";
    out_r = e1; out_m = e2; /* Assign edges to ports */
  }
  actor S {
    name = "mod_src";
    in_ctrl = e1; out_data = e3;
    mode_count = 3;
  }
  actor T {
    name = "mod_lut";
    in_ctrl = e1; in_bits = e3; out_symbol = e4;
    mode_count = 4;
  }
  /* Other actor definitions */
  /* ... */
}
```

In this example, the RMOD application is described using CFDF semantics, which is represented by the `cfdf` keyword in DIF. The `topology` block defines the actors (nodes) and edges of the data-flow graph and associates a unique identifier with each actor and each edge. Because data-flow graphs are directed graphs, each edge is represented as an ordered pair (u, v), where u is the source actor and v is the sink actor. Each actor can be associated with an optional `actor` block, where attributes associated with the actor are defined. The attributes can provide arbitrary information associated with the actor using a combination of *built-in* and *user-defined* attribute specifiers. In the example of Listing 1, the `actor` block specifies the following attributes: (1) the name of the implementation associated with the actor (to help differentiate between alternative implementations for the same abstract actor model), (2) input/output port connections with the incident edges, and (3) the number of CFDF modes for the actor.

In addition to the language features illustrated in Listing 1, DIF also supports a variety of other features for specifying information pertaining to data-flow-based application models. For example, DIF supports hierarchical specification, where an actor in one graph can be linked with a "nested" subgraph to promote top-down decomposition of complex graphical models and to help support different forms of semantic hierarchy, such as those involved in parameterized data-flow semantics [2]. Another feature in DIF is support for *topological patterns*, which enable compact, parameterized descriptions of various kinds of graphical patterns (e.g., chain, ring, and butterfly patterns) for instantiating and connecting actors and edges [46].

36.3.4 Model-Based Design and Integration Using DIF

The DIF package provides an integrated set of models and methods, illustrated in Fig. 36.3, for developing customized data-flow-model-based design flows targeted to different areas of signal processing, and different kinds of target platforms. As opposed to being developed primarily as a stand-alone data-flow tool, DIF is designed for flexibility in integrating established or novel data-flow capabilities into arbitrary model-based design environments for DSP. For example, Zaki presents a DIF-based tool for mapping Software Defined Radio (SDR) applications into GPU implementations, and integrating the derived mapping solutions into GNU Radio, which is a widely used environment for SDR system design [52]. As another example, DIF has been integrated to provide data-flow analysis and transformation capabilities for the popular data-flow language called CAL, which was discussed previously in Sect. 36.2.1. For details on this application of DIF to CAL, we refer the reader to [16, 17], while readers can find details about the CAL language in [10].

The DIF package consists of three major parts: the DIF representation, DIF-based graph analysis and transformation techniques, and tools for simulation and software synthesis.

DIF representation. The DIF package provides an extensible set of data structures that represent data-flow-based application models, as they are specified in the DIF language and as they are transformed into alternative models for the purposes of

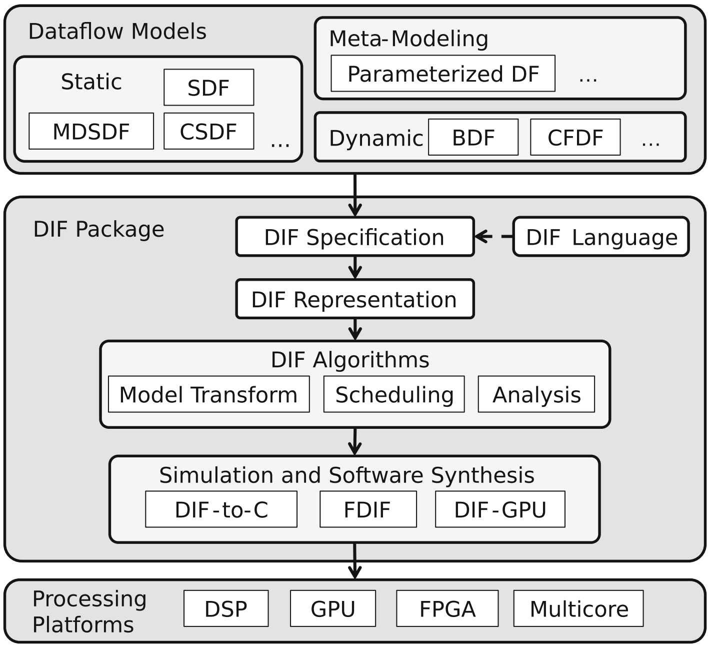


Fig. 36.3 Overview of the DIF package

analysis, optimization, or software synthesis. These graphical data structures are collectively referred to as the DIF intermediate representation or "DIF representation" for short. The initial DIF representation (before any transformations are applied) for a given DIF language specification is constructed by the DIF front-end tools, which are centered on a Java-based parser. This parser is developed using the SableCC compiler framework [13].

Analysis and Transformation Techniques. The DIF package provides implementations of a large set of methods for data-flow model analysis and transformation, including methods for scheduling, and buffer management. These methods operate on the graphical data structures within the DIF representation. The analysis and transformation techniques provided in DIF are useful in many aspects of data-flow-based design and implementation.

Simulation and Software Synthesis. DIF presently includes a number of tools for simulation and software synthesis from data-flow models. Functional DIF (FDIF) simulates CFDF-based models where actor functionality is programmed in terms of CFDF semantics using Java [43] along with CFDF-specific APIs. FDIF is designed especially to help designers to efficiently prototype and validate alternative kinds of static, dynamic, and quasi-static scheduling strategies. The DIF-to-C tool generates C code that is optimized for efficient execution on programmable digital signal processors [23]. The software synthesis capabilities in DIF-to-C are integrated with a variety of analysis and transformation techniques in DIF so that designers can apply different combinations of transformations to explore trade-offs among data memory requirements, code size, and execution speed. DIF-GPU is a newly developed software synthesis tool that is targeted to heterogeneous CPU–GPU platforms. Currently, DIF-GPU generates multi-threaded Compute Unified Device Architecture (CUDA) application code that can utilize both Central Processing Units (CPUs) and GPUs for implementation of high-performance DSP systems. Further details on DIF-GPU are discussed in Sect. 36.6.

36.4 Lightweight Data-Flow Environment

LIDE facilitates design and implementation of DSP actors and systems using a structured, CFDF-based data-flow approach that can be integrated with a wide variety of platform-oriented languages, such as C, CUDA, OpenCL, Verilog, and VHDL [47, 48]. LIDE is centered on a compact set of abstract APIs for developing data-flow actors and edges. These APIs are (1) defined in terms of fundamental data-flow principles, (2) independent of any specific programming language, and (3) readily retargetable across a wide variety of specific languages for DSP implementation, including the platform-oriented languages listed above.

LIDE is designed with a primary objective of allowing DSP system designers to apply and experiment with data-flow techniques relatively easily in the context of their existing design processes, language preferences, and target platforms. This objective is supported by the compact collection of retargetable, language-agnostic APIs that LIDE is based on. LIDE also provides collections of pre-designed data-flow gems, as described in Sect. 36.1.3.

When LIDE is integrated with a specific programming language XYZ for implementing gems, we refer to the resulting integrated design tool as LIDE-XYZ or in some cases as LIDE-X if X is used as an abbreviation for XYZ. Existing subsystems within LIDE include LIDE-C, LIDE-CUDA, LIDE-V, and LIDE-OCL, where the latter two represent the integration of LIDE with the Verilog Hardware Description Language (HDL) and OpenCL, respectively.

36.4.1 Actor Design in LIDE

As described previously, actor implementation in LIDE is based on the CFDF model of computation. This choice of CFDF as the modeling foundation for LIDE is motivated by the high expressive power of CFDF, and its utility in working with heterogeneous forms of data flow [44].

Actor design in LIDE includes four basic interface functions, which are referred to as the construct, enable, invoke, and terminate functions of an actor. The *construct* function instantiates an actor and performs pre-execution initialization of the actor, such as initializing values of actor parameters and allocating storage that is related to the state of the actor. Conversely, the *terminate* function performs any operations that are required for "closing out" the actor after the enclosing graph has finished executing. Such operations include freeing memory that has been allocated in the corresponding construct function.

The *enable* and *invoke* functions provide direct interfaces for key concepts of CFDF semantics, which were discussed in Sect. 36.3.1. As one would guess, the enable and invoke functions in LIDE are defined to provide implementations for the enable and invoke functions in CFDF semantics. We employ a minor abuse of terminology here, where this pair of functions is defined with the same names in both LIDE (a design tool) and CFDF (a model of computation). Where there may be confusion, one may qualify a reference to the function with an appropriate reference to the tool or model (e.g., "the LIDE enable function").

The enable and invoke functions in LIDE provide flexible interfaces for implementing arbitrary schedulers, including static, dynamic, and quasi-static schedulers, for executing data-flow graph implementations. The enable function is implemented by the actor programmer to check whether or not the actor has sufficient tokens on its input ports and enough empty space on its output ports to support a single firing in next CFDF mode of execution that is currently associated with the actor. Similarly, the invoke function is implemented to execute a single firing of the actor according to its next mode. The invoke function should also update the next mode of the actor, which in turn determines the conditions that will be checked by the enable function if it is called prior to the next actor firing.

When the invoke function is called, it is assumed that sufficient input tokens and output space are available (since there is a separate API function dedicated to checking these conditions). Thus, the actor programmer should not implement checks for these conditions within the invoke function. These conditions should be satisfied – as part of the design rules of any tool that implements CFDF semantics – before calling the invoke function to execute a given actor firing.

We emphasize that in a given scheduler for an enclosing data-flow graph, it is not always necessary to call the enable function of an actor before calling the invoke function. In particular, such calls to the enable function can be bypassed at run time if the corresponding conditions are guaranteed through other forms of analysis, including any combination of static, dynamic, and hybrid static/dynamic analysis. For example, when implementing the scheduler for a LIDE-based data-flow graph that consists only of SDF or CSDF actors, the use of the enable function can be avoided entirely if a static schedule is employed [6, 30]. This allows designers in LIDE to more effectively utilize the large collection of available static scheduling techniques for SDF and CSDF representations (e.g., see [3, 8, 12, 14, 36, 40, 45]).

For more details on actor implementation in LIDE, we refer the reader to [48].

36.4.2 Parameterized Sets of Modes

Actor design in LIDE naturally supports the concept of Parameterized Sets of Modes (PSM), which is a modeling enhancement to CFDF that enables designers to more concisely specify and work with actor behavior that involves groups of related modes [33].

For example, consider an actor A that has two input ports *in1* and *in2*, and a single output port *out*. The actor starts execution in a mode called *read_length*, which consumes a single, positive integer-valued token from *in1*, and stores this consumed value in a state variable N. The value of the input token consumed from *in1* is restricted to fall in the range $\{1, 2, \ldots, M\}$, where M is a parameter of A. In the next firing, the actor consumes a vector spanning N input tokens from *in2*, computes the maximum of these N values, outputs the result (as a single token) on *out*, and determines its next mode to be the *read_length* mode. Thus, intuitively, the actor executes through alternate firings where (a) a vector length is read and used to determine the consumption rate of a subsequent mode, and then in this subsequent mode, (b) a vector is read and processed to produce a single output token.

Using standard CFDF notation, we can represent this as an actor that has $(M + 1)$ distinct modes, i.e., as M different "vector processing modes" in addition to the *read_length* mode. However, such a representation can become unwieldy, especially if M is large. A PSM is a level of abstraction that allows us to group together a collection of related modes with one or more parameters that are used to select a unique mode from the collection at run time. These parameters can be determined statically or dynamically, allowing for significant flexibility in how PSMs are applied to actor design.

In this simple vector processing example, the M vector processing modes can be grouped together into a single PSM *vect_proc*, and with an associated parameter *vect_len* whose value corresponds to the value of the actor state variable N.

Technically, an actor *mode* in LIDE corresponds to a PSM rather than an individual CFDF actor mode. A LIDE actor can produce or consume different numbers of tokens in the same mode as long as the data-flow rates are all uniquely determined by the LIDE actor mode PSM and the values of the actor parameters that are associated with that PSM. Such unique determination of data-flow rates ensures that the underlying actor behavior corresponds to CFDF semantics, while allowing the code to be developed and the actor functionality to be reasoned about in terms of the higher-level PSM abstraction.

For a more formal and thorough development of PSM-based modeling, we refer the reader to [33].

36.4.3 Implementation in LIDE

In this section, we discuss details of design and implementation of data-flow components in LIDE using an example based on LIDE-C. In LIDE-C, data-flow gems are implemented in the C language. A collection of gems and utilities is provided as part of LIDE-C. These can be linked through various LIDE-C libraries into data-flow graph implementations, and they can also serve as useful templates or examples to help users develop new gems for their applications.

More specifically, LIDE-C contains a set of libraries called `gems`, and another library called `tools`. Basic actor and edge FIFO implementations are provided in gems, while basic utilities, including a simple scheduler, are accessible in tools. The scheduler provided in LIDE-C is a basic form of CFDF scheduler, called a *canonical scheduler* [44]. This form of scheduler can be applied to arbitrary data-flow graphs in LIDE-C. Because it is general and easy to use, it is useful for functional validation and rapid prototyping. However, it is relatively inefficient, as it is designed for simplicity and generality, rather than for efficiency.

More efficient schedulers can be implemented by LIDE-C designers using the core LIDE APIs, including the enable and invoke functions for the actors. Each LIDE-C actor, as a concrete form of LIDE actor, must have implementations of these functions. LIDE-C schedulers can also be generated automatically through software synthesis tools.

36.4.3.1 Data-Flow Graph Components

In LIDE-C, gems (actors and FIFOs) are implemented as Abstract Data Types (ADTs) in C. Such ADT-based implementation provides a C-based, object-oriented design approach for actors and FIFOs in LIDE-C. As we discussed in Sect. 36.4.1, each LIDE actor has four standard interface functions. The developer of an actor in LIDE-C must provide implementations of these functions as methods – referred to as the new, enable, invoke, and terminate methods – of the ADT for the actor.

An analogous process is followed for FIFO design in LIDE-C and in related targets of LIDE, including LIDE-CUDA and LIDE-OCL. In particular, users can define any number of different FIFO types (e.g., corresponding to different forms of physical implementation, such as mappings to different kinds of memories), where each FIFO type is designed as an ADT. For example, in LIDE-OCL, which is currently developed for hybrid CPU-GPU implementation platforms, two FIFO ADTs are available – one for implementation of the FIFO on a CPU and another for implementation on a GPU.

The abstract (language-agnostic) LIDE API contains a set of required interface functions for FIFOs that implement edges in LIDE programs. In LIDE-C, FIFOs are implemented as ADTs where the required interface functions are implemented as methods of these ADTs. Required interface functions for FIFOs in LIDE include functions for construction and termination (analogous to the construct and terminate functions for actors), reading (consuming) tokens, writing (producing) tokens, querying the number of tokens that currently reside in a FIFO, and querying the capacity of a FIFO. The capacity of a FIFO in LIDE is specified through an argument to the construct function of the FIFO.

Listing 2 shows the function prototypes for the new, enable, invoke, and terminate methods in LIDE-C. In addition to these interface functions, designers can add auxiliary functions in their actor implementations. For working with actor parameters, components in the LIDE-C libraries employ a common convention of using corresponding set and get methods associated with each parameter (e.g., set_tap_count, get_tap_count).

Listing 2 The format for function prototypes of the new, enable, invoke, and terminate methods of a LIDE-C actor

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

boolean lide_c_<actor_name>_enable(
    lide_c_<actor_name>_context_type *context)

void lide_c_<actor_name>_invoke(
    lide_c_<actor_name>_context_type *context)

void lide_c_<actor_name>_terminate(
    lide_c_<actor_name>_context_type *context)
```

Each function prototype shown in Listing 2 involves an argument that points to a data structure that is referred to as the *actor context* (or simply "context"). Each actor A in a LIDE-C data-flow graph implementation has an associated context, which encapsulates pointers to the FIFOs that are associated with the edges incident to A; function pointers to the `enable` and `invoke` methods of A; an integer variable that stores the index of the current CFDF mode or PSM of A; and parameters and state variables of A.

For purposes of data-flow graph analysis or transformation (e.g., as provided by DIF), a LIDE actor that employs one or more state variables can be represented by attaching a self-loop edge to the graph vertex associated with the actor. Here, by a *self-loop edge*, we mean an edge e for which $src(e) = snk(e)$. In general, one can also use such a self-loop edge to represent inter-firing dependencies for an actor that can transition across multiple CFDF modes at run time (here, the mode variable acts as an implicit state variable). On the other hand, if the mode is uniquely determined at graph configuration time and does not change dynamically, then this "CFDF-induced" self-loop edge can be omitted. Such an actor can, for example, be executed in a data parallel style (multiple firings of the actor executed simultaneously) if there are no state-induced self-loop edges or other kinds of cyclic paths in the data-flow graph that contain the actor.

36.4.3.2 Actor Implementation Example

As a concrete example of applying LIDE-C, we introduce in this section a LIDE-C implementation of a modulation selection actor. Recall that such an actor is employed as actor T in the RMOD application that was introduced in Sect. 36.3.2. This actor T is an example of CFDF semantics augmented with the concept of PSMs. Recall that the data-flow graph and data-flow tables for this actor are shown in Figs. 36.1 and 36.2.

Listing 3 and Listing 4 illustrate key code segments within the `enable` and `invoke` methods, respectively, for actor T in our LIDE-C implementation of the actor. These code segments involve carrying out the core computations for determining fireability and firing the actor, respectively, based on the actor mode or PSM that is active when the corresponding method is called. For conciseness, each of these two listings shows in detail the functionality corresponding to a single mode, along with the overall structure for selecting an appropriate action based on the current mode (using a `switch` statement in each case). Lines marked with "`......`" represent code that is omitted from the illustrations for conciseness. Interface functions whose names start with `lide_c_fifo` are methods of a basic FIFO ADT that is available as part of LIDE-C.

Listing 3 Code within the enable method for actor T in the RMOD application

```
/* context: structure that stores actor information. E.g.,
   context->mode stores the actor's current mode.
*/
switch (context->mode) {
    case LIDE_C_RMOD_T_MODE_INIT:
        result = (lide_c_fifo_population(context->fifo_ctrl_input)
                    >= 1);
        break;
    case LIDE_C_RMOD_T_MODE_BPSK:
        result = ......
        break;
    case LIDE_C_RMOD_T_MODE_QPSK:
        result = ......
        break;
     case LIDE_C_RMOD_T_MODE_QAM16:
        result = ......
        break;
    default:
        result = FALSE;
        break;
}
return results;
```

Listing 4 Code within the invoke method for actor T in the RMOD application

```
switch (context->mode) {
     case LIDE_C_RMOD_T_MODE_INIT:
        /* scheme: variable indicating BPSK, QPSK or QAM16 */
        lide_c_fifo_read(context->fifo_ctrl_input, &scheme);
        context->mode = scheme;
        /* nbits: number of bits to process for the given scheme
           rb: remaining bits before switching scheme */
        context->rb = context->nbits;
        break;
     case LIDE_C_RMOD_T_MODE_BPSK:
        lide_c_fifo_read_block(context->fifo_data_input,
                               &bits, 1);
        code.x = context->bpsk_table[bits].x;
        code.y = context->bpsk_table[bits].y;
        lide_c_fifo_write(context->fifo_data_output, &code);
        context->rb --;
        if (context->rb > 0) {
            context->mode = LIDE_C_RMOD_T_MODE_BPSK;
        } else {
            context->mode = LIDE_C_RMOD_T_MODE_INIT;
        }
        break;
     case LIDE_C_RMOD_T_MODE_QPSK:
        ......
        break;
     case LIDE_C_RMOD_T_MODE_QAM16:
        ......
        break;
    default:
        context->mode = LIDE_C_RMOD_T_MODE_INACTIVE;
        break;
}
```

36.5 DSPCAD Integrative Command Line Environment

In this section, we describe the DSPCAD Integrative Command Line Environment (DICE), which is a Bash-based software package for cross-platform and model-based design, implementation, and testing of signal processing systems [5]. The DICE package is developed as part of the DSPCAD Framework to facilitate exploratory research, design, implementation, and testing of digital hardware and embedded software for DSP. DICE has also been used extensively in teaching of cross-platform design and testing methods for embedded systems (e.g., see [4]). DICE has been employed to develop research prototypes of signal processing applications involving a wide variety of platforms, including desktop multi-core processors, Field-Programmable Gate Arrays (FPGAs), GPUs, hybrid CPU-GPU platforms, low-power microcontrollers, programmable digital signal processors, and multi-core smartphone platforms. An overview of DICE is given in [5], and an early case study demonstrating the application of DICE to DSP system design is presented in [26]. In the remainder of this section, we highlight some of the most complementary features of DICE in relation to LIDE and DIF.

Because DICE is based on Bash, it has various advantages that complement the advantages of platform-based or language-specific integrated development environments (IDEs). For example, DICE can be deployed easily on diverse operating systems, including Android, Linux, Mac, Solaris, and Windows (with Cygwin). The primary requirement is that the host environment should have a Bash command line environment installed. DICE is also agnostic to any particular actor implementation language or target embedded platform. This feature of DICE helps to provide a consistent development environment for designers, which is particularly useful when developers are experimenting with diverse hardware platforms and actor implementation languages.

36.5.1 Convenience Utilities

DICE includes a collection of simple utilities that facilitate efficient directory navigation through directory hierarchies. This capability is useful for working with complex, cross-platform design projects that involve many layers of design decomposition, diverse programming languages, or alternative design versions for different subsystems. These directory navigation operations help designers to move flexibly and quickly across arbitrary directories without having to traverse through multiple windows, execute sequences of multiple `cd` commands, or type long directory paths. These operations also provide a common interface for accelerating fundamental operations that is easy to learn and can help to quickly orient new members in project teams.

DICE also provides a collection of utilities, called the Moving Things Around (MTA) utilities, for easily moving or copying files and directories across different directories. Such moving and copying is common when working with design projects (e.g., to work with code or documentation templates that need to be copied and then adapted) and benefit from having a simple, streamlined set of utilities. The MTA utilities in DICE are especially useful when used in conjunction with the directory navigation utilities, described above.

Some of the key directory navigation utilities and MTA utilities in DICE are summarized briefly in Table 36.1.

The items in Table 36.1 that are enclosed in angle brackets (< . . . >) represent placeholders for command arguments. The abbreviation-based names of the first three utilities listed in Table 36.1 are derived as follows: `dlk` stands for (create) Directory LinK, `g` stands for Go, and `rlk` stands for Remove LinK. The other two utilities listed in Table 36.1 use a naming convention that applies to many core utilities in DICE where the prefix "dx" is used at the beginning of the utility name. The name `dxco` stands for (CO)py (a file or directory), and `dxparl` stands for paste and remove the last file or directory transferred.

Table 36.1 Selected navigation utilities and MTA utilities in DICE

Utility	Description
`dlk <label>`	Associate the specified label with the Current Working Directory (CWD)
`g <label>`	Change directory to the directory that is associated with the specified label
`rlk <label>`	Remove the specified label from the set of available directory navigation labels
`dxco <arg>`	Copy the specified file or directory to the DICE user clipboard
`dxparl`	Paste (copy) into the CWD the last (most recent) file or directory that has been transferred to the to the DICE user clipboard, and remove this file or directory from the clipboard

36.5.2 Testing Support

One of the most useful sets of features in DICE is provided by its lightweight and language-agnostic unit testing framework. This framework can be applied flexibly across arbitrary actor implementation languages (C, CUDA, C++, Java, Verilog, VHDL, etc.) and requires minimal learning of new syntax or specialized languages [26]. The language-agnostic orientation of DICE is useful in heterogeneous development environments, including codesign environments, so that a common framework can be used to test across all of the relevant platforms.

In a DICE-based test suite, each specific test for an HDL or software implementation unit is implemented in a separate directory, called an Individual Test Subdirectory (ITS), which is organized in a certain way according to the DICE-based conventions for test implementation. 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`). To exclude a test from test suite evaluation, one can simply change its name so that it does not begin with `test`.

36.5.2.1 Required Components of an ITS

Here, we describe the required components of an ITS. Except for the set of input files for the test, each of these components takes the form of a separate file. The set of input files may be empty (no files) or may contain any number of files with any names that do not conflict with the names of the required ITS files, as listed below:

- A file called `test-desc.txt` that provides a brief explanation of what is being tested by the ITS, that is, what is distinguishing about this test compared to the other tests in the test suite.
- An executable file (e.g., some sort of script) called `makeme` that performs all necessary compilation steps (e.g., compilation of driver programs) that are needed for the ITS. 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; project source code is assumed to be compiled separately before a test suite associated with the project is exercised.
- An executable file called `runme` that runs the test and directs all normal output to standard output, and all error output to standard error.
- Any input files that are needed for the test.
- A file called `correct-output.txt` that contains the standard output text that should result from the test. If no output is expected on standard output, then `correct-output.txt` should exist in the ITS as an empty file.
- A file called `expected-errors.txt` that contains the standard error text that should result from the test. This placeholder provides a mechanism to test the correct operation of error detection and error reporting functionality. If no

output is expected on standard error, then `expected-errors.txt` should exist in the ITS as an empty file.

The organization of an ITS is structured in this same, language-independent, form – based on the required items listed above – regardless of how many and what kinds of design languages are involved in a specific test. This provides many benefits in DSP codesign, where several different languages and back-end (platform-based) tools may be employed or experimented with in a given system design. For example, the DICE test suite organization allows a designer or testing specialist to switch between languages or project subsystems without being distracted by language-specific peculiarities of the basic structure of tests and their operation.

As one might expect from this description of required files in an ITS, a DICE-based test is evaluated by automatically comparing the standard output and standard error text that is generated by `runme` to the corresponding `correct-output.txt` and `expected-errors.txt` files.

Note that because of the configurable `runme` interface, it is not necessary for all of the output produced by the project code under test to be treated directly as test output. Instead, the `runme` script 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. This provides great flexibility in how test output is defined and managed.

36.5.2.2 Relationship to Other Testing Frameworks and Methodologies

The DICE features for unit testing are largely complementary to the wide variety of language-specific testing environments (e.g., see [18, 24, 50]). More than just syntactic customizations, such frameworks are often tied to fundamental constructs of the language. DICE can be used to structure, organize, and execute in a uniform manner unit tests that employ language-specific and other forms of specialized testing frameworks. For example, specialized testing libraries for Java in a simulation model of a design can be employed by linking the libraries as part of the `makeme` scripts in the ITSs of that simulation model. When a designer who works primarily on hardware implementation for the same project examines such a “simulation ITS,” he or she can immediately understand the overall organization of the associated unit test and execute the ITS without needing to understand the specialized, simulation-specific testing features that are employed.

DICE is also not specific to any specific methodology for creating or automatically generating unit tests. A wide variety of concepts and methods have been developed for test construction and generation (e.g., see [20]). By providing a simple and flexible environment for implementing, executing, and managing tests, the DICE unit testing framework can be used to prototype different kinds of test development methodologies and apply them in arbitrary implementation contexts.

For further details on the process of test implementation in DICE, and the relationship of DICE to other testing frameworks, we refer the reader to [4, 5, 26].

36.6 DSPCAD Framework Example: DIF-GPU

In this section, we demonstrate the DSPCAD Framework by describing its use to develop DIF-GPU, a software synthesis tool for mapping SDF graphs onto hybrid CPU-GPU platforms. Using DIF-GPU, a DSP designer can specify a signal-flow graph as an SDF graph in the DIF language; implement the individual actors of the graph in LIDE-CUDA; automatically schedule and generate interacting CPU and GPU code for the graph; and validate the generated implementation using the cross-platform testing capabilities of DICE.

We note that the case study presented in this section is not intended to emphasize details of a specific data-flow tool for GPU implementation but rather to demonstrate how the complementary resources and capabilities in the DSPCAD Framework can be applied to efficiently prototype such a tool. For a detailed presentation of the DIF-GPU tool, we refer the reader to [32].

36.6.1 DIF-GPU Overview

DIF-GPU targets heterogeneous CPU-GPU platforms in which multi-core CPUs and GPUs operate concurrently to provide high-performance signal processing capability. Modern GPUs can contain hundreds or thousands of single instruction multiple data (SIMD) multi-processor cores to process large amounts of data in parallel. Such an architecture enables GPUs to obtain significant performance gain over CPUs on data parallel tasks. Cooperation between a multi-core CPU and GPU allows various types of parallelism to be exploited for performance enhancement, including pipeline, data, and task parallelism.

DIF-GPU targets CPU-GPU platforms that are modeled as host-device architectures where the CPU is referred to as the "host" and the GPU as the "device," and where the employed CPUs, main memory, and GPUs are connected by a shared bus. CPUs control the GPUs by dispatching commands and data from main memory, while GPUs perform their assigned computations in their local memories (device memory). A GPU's device memory is private to that GPU and separated from main memory and the memories of other devices. Data transfers between the host and individual devices are referred to as Host-to-Device (H2D) or Device-to-Host (D2H) data transfers, depending on the direction. H2D and D2H data transfers can produce large overhead and significantly reduce the performance gain provided by GPUs (e.g., see [15]). To achieve efficient implementations in DIF-GPU, such overhead is taken carefully into account in the processes of task scheduling and software synthesis.

DIF-GPU is developed using the integrated toolset of the DSPCAD Framework, including DIF, LIDE, and DICE. Methods for data-flow analysis, transformation, scheduling, and code generation are developed by building on capabilities of the DIF package. Implementation of GPU-accelerated actors and run time, multi-threaded execution support are developed by applying LIDE-CUDA. Unit testing and application verification are carried out using DICE.

36.6.2 Graph Transformations and Scheduling using DIF

Figure 36.4 illustrates the overall workflow of DIF-GPU. This workflow consists of 3 major steps, vectorization, Scheduling and Data Transfer Configuration (SDTC), and code generation. Data parallelism is exploited by the vectorization step, while pipeline and task parallelism are exploited by the SDTC step.

36.6.3 Vectorization

Data-flow graph vectorization can be viewed as a graph transformation that groups together multiple firings of a given actor into a single unit of execution [45]. The number of firings involved in such a group is referred to as the Vectorization Factor (VF). Vectorization is a useful method for exploiting data parallelism in data-flow models.

Suppose that A is an actor in an SDF graph G, and G' represents the transformed graph that results from replacing A with a vectorized version A_b of A with $VF = b$. The edges in G' are the same as those in G, except that for all input edges of A_b, the consumption rates are effectively multiplied by b (relative to their corresponding rates in G), and similarly, for all output edges of A_b, the production rates are multiplied by b.

Vectorization exposes potential for exploiting parallelism across multiple firings of the same actor. For example, when executing A_b on a GPU, blocks of b firings of A can be executed together concurrently on stream processors in the GPU.

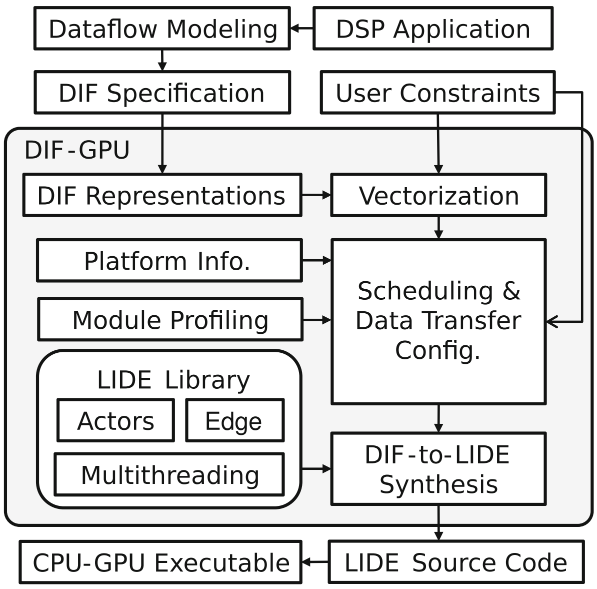


Fig. 36.4 The DIF-GPU workflow

DIF-GPU applies vectorization in a form called Graph-Level Vectorization (GLV) [52] for SDF graphs. GLV involves a positive integer parameter J that is called the *GLV degree* for the input SDF graph G. In GLV, J iterations of a minimal periodic schedule for G are scheduled together, and the GLV degree is used in conjunction with the repetitions vector q for G to derive the VF for each actor. For background on periodic schedules and repetitions vectors for SDF graphs, we refer the reader to [30].

More specifically, in GLV, the VF for each actor A is derived as $J \times q(A)$, where $q(A)$ represents the repetitions vector component that is indexed by A. After transforming each actor by its VF associated with a given GLV degree J, the resulting vectorized graph G_{vect}, is a *single-rate* SDF graph that represents the execution of J successive iterations of a minimal periodic schedule for G. Here, by a single-rate SDF graph, we mean that the repetitions vector components are uniformly equal to unity – that is, if r represents the repetitions vector for G_{vect}, then for every actor A in G_{vect}, $r(A) = 1$.

In DIF-GPU the input SDF graph is assumed to be acyclic (apart from the possibility of self-loop edges induced by actor state) so that there are no cyclic paths in the application graph that impose limitations on the GLV degree. A wide variety of practical signal processing systems can be represented in the form of acyclic SDF graphs (e.g., see [3]). The techniques employed in DIF-GPU can readily be extended to more general graph topologies, e.g., by applying them outside of the strongly connected components of the graphs. Such an extension is a useful direction for further development in DIF-GPU.

Actors in DIF-GPU are programmed using a VF parameter, which becomes part of the actor context in LIDE-CUDA. The actor developer implements vectorized code for each actor in a manner that is parameterized by the associated VF parameter and that takes into account any limitations in data parallel operation or memory management constraints imposed by actor state. For example, to implement a vectorized Finite Impulse Response (FIR) filter in DIF-GPU, a VF parameter is included in the associated LIDE-C actor context such that the actor consumes and produces VF tokens in each firing. Along with this VF parameter, the actor context contains pointers to (1) an array of filter coefficients and (2) an array of past samples for the filter. The past samples array, which contains $(N - 1)$ elements, stores the most recently consumed $(N - 1)$ tokens by the actor. Here, N is the order of the filter. Firing the vectorized GLV filter involves consuming b input tokens, generating b output tokens, and updating the actor state that is maintained in the past samples array, where b is the value of the VF parameter. Using careful buffer management within the LIDE-CUDA actor implementation, the b output samples for the actor are computed in parallel on the target GPU assuming that there are sufficient resources available in the GPU in relation to N and b.

The GLV approach employed in DIF-GPU is useful because it provides a single parameter (the GLV degree) that can be employed to control system-level trade-offs associated with vectorization and thereby facilitates design space exploration across a wide range of these trade-offs. For example, vectorization involves trade-offs involving the potential for improved throughput and exploitation of data parallelism at the expense of increased buffer memory requirements [45, 52].

36.6.4 Graph Scheduling and Mapping

After the GLV transformation is applied to the intermediate SDF graph representation in DIF, DIF-GPU generates a schedule for the vectorized, single-rate SDF graph G_{vect}. The schedule can either be generated from a user-specified mapping configuration (assignment of actors to specific GPU and CPU resources) or computed using a selected scheduling algorithm that is implemented in the DIF package. When the user specifies the mapping configurations, DIF-GPU generates a schedule by firing the actors on each processor according to their topological order in G_{vect}.

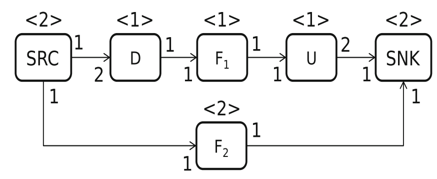
When the user does not specify the mapping configuration, the user can select a scheduling algorithm to automatically generate the mapping and schedule. DIF-GPU integrates multiple scheduling algorithms, including a First-Come First-Serve (FCFS) scheduler and Mixed Integer Linear Programming (MILP) [52] scheduler. Providing multiple schedulers, automated code synthesis capability, and the ability to easily extend the tool with new schedulers allows the user to experiment with trade-offs associated with different scheduling techniques and select the strategy that is most appropriate in relation to the complexity of the input graph and the given design constraints.

DIF-GPU avoids redundant data transfer between CPUs and GPUs by complementary design of alternative FIFO implementations in LIDE-CUDA and usage of specialized actors for managing data transfer. In particular, DIF-GPU incorporates special data-transfer actors that are designed for optimized, model-based interprocessor communication between actors across separate memory subsystems. These data-transfer actors are called the *H2D* and *D2H* actors (recall that these abbreviations stand for host-to-device and device-to-host). *H2D* copies data from a buffer allocated on the CPU (i.e., the host) memory to the GPU (i.e., the device) memory; and conversely, *D2H* copies data from a GPU buffer to the host memory. After the scheduling process in DIF-GPU is complete, *H2D* or *D2H* actors are automatically inserted in the DIF representation for application data-flow graph edges that involve communication between host and device memory. This insertion of data-transfer actors is performed as an automated post-processing step both for user-specified and automatically generated mappings.

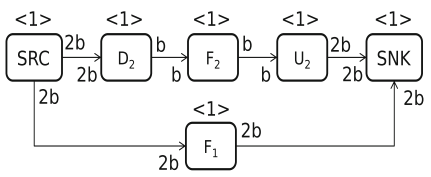
For example, in Fig. 36.5d, F_2 is mapped onto a GPU, so *H2D* is inserted between *src* and F_2, and *D2H* is inserted between F_2 and *snk*. This method employed by DIF-GPU to handle data transfer between processors aims to free the LIDE-CUDA actor designer from having to implement details of interprocessor communication and synchronization and to reduce data transfer overhead.

As a simple example to concretely demonstrate the DIF-GPU workflow, Fig. 36.5a and b show an SDF graph with execution time estimates that are proportional to the VF b. Such execution time profiles can be provided through actor-level benchmarking and then used as input to the scheduling phase in DIF-GPU. The target platform in this example is assumed to consist of a single CPU and single GPU. Brackets above the actors indicate the repetitions vector components

a



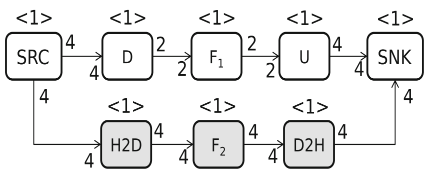
c



b

Actor	CPU-time	GPU-time	Vectorization Degree
SRC	b	N/A	4
D	2b	2b	1
F_1	2b	b	2
F_2	2b	b	4
U	2b	2b	1
H2D/D2H	b	b	4
SNK	b	N/A	4

d



☐ CPU : SRC, D, F_1, U, SNK, SRC, ...

▨ GPU : H2D, F_2, D2H, H2D, ...

Fig. 36.5 An illustration of the DIF-GPU workflow using a simple SDF graph example. (**a**) Original SDF graph. (**b**) VF-dependent execution times on CPU and GPU. (**c**) Vectorized graph with $VF = b$. (**d**) Vectorized graph for $b = 2$ with data-transfer actors inserted, and the corresponding schedule for CPU-GPU implementation

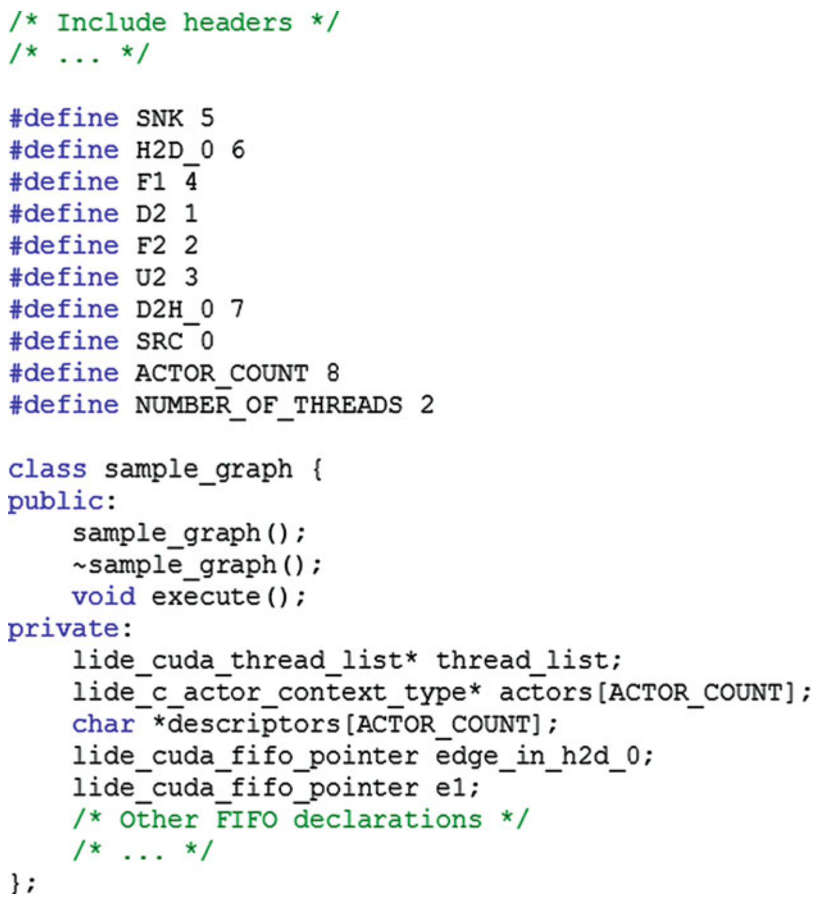
associated with the actors. Figure 36.5c shows the vectorized graph G_{vect} when $VF = b$. Figure 36.5d shows the DIF representation that results from further transformation through the insertion of *H2D* and *D2H* actors when $b = 2$ and when F_2 is mapped onto the GPU and other actors are mapped onto the CPU.

36.6.5 Code Generation

DIF-GPU generates well-structured, human-readable CUDA source code that can be linked with LIDE-CUDA libraries and compiled with standard CUDA development tools for implementation on CPU-GPU platforms.

Figures 36.6 and 36.7 show the generated LIDE-CUDA header and implementation file code for the sample graph in Fig. 36.5d. The generated code consists mainly of the *constructor*, *execute function*, and *destructor* for the synthesized SDF graph implementation. The constructor instantiates all of the actors and edges in the data-flow graph and connects the actors and edges according to the graph topology. The edges are assigned capacities, token sizes, and memory spaces automatically based on information in the DIF language graph specification, and on graph analysis techniques in the DIF package. The actors are assigned to processors based on the user-specified or auto-generated mapping information.

The generated code also initializes data structures for the LIDE-CUDA multi-thread scheduler. The execute function for the synthesized SDF graph



```
/* Include headers */
/* ... */

#define SNK 5
#define H2D_0 6
#define F1 4
#define D2 1
#define F2 2
#define U2 3
#define D2H_0 7
#define SRC 0
#define ACTOR_COUNT 8
#define NUMBER_OF_THREADS 2

class sample_graph {
public:
    sample_graph();
    ~sample_graph();
    void execute();
private:
    lide_cuda_thread_list* thread_list;
    lide_c_actor_context_type* actors[ACTOR_COUNT];
    char *descriptors[ACTOR_COUNT];
    lide_cuda_fifo_pointer edge_in_h2d_0;
    lide_cuda_fifo_pointer e1;
    /* Other FIFO declarations */
    /* ... */
};
```

Fig. 36.6 Generated header file for the example SDF graph of Fig. 36.5d

implementation starts the multi-thread scheduler and creates the threads. The threads then proceed to execute actor firings based on the mapping decisions embodied in the generated code. The destructor terminates the threads and actor structures and releases allocated memory.

36.6.6 Testing in DIF-GPU Using DICE

DIF-GPU employs DICE for unit testing in all parts of the workflow. The DIF-GPU framework is developed using a combination of Java, C, and CUDA; therefore, the multi-language support in DICE is useful for testing of the all components within the DIF-GPU framework. Components in DIF-GPU that require unit testing include (1) relevant data-flow transformation and scheduling techniques that apply

```
#include "sample_graph.h"

/* Full constructor */
sample_graph::sample_graph(){
    /* Create edges */
    edge_in_h2d_0 = lide_cuda_fifo_new(4, sizeof(float), CPU);
    e1 = lide_cuda_fifo_new(4, sizeof(float), CPU);
    e3 = lide_cuda_fifo_new(2, sizeof(float), CPU);
    edge_out_d2h_0 = lide_cuda_fifo_new(4, sizeof(float), CPU);
    edge_out_h2d_0 = lide_cuda_fifo_new(4, sizeof(float), GPU);
    edge_in_d2h_0 = lide_cuda_fifo_new(4, sizeof(float), GPU);
    e2 = lide_cuda_fifo_new(2, sizeof(float), CPU);
    e4 = lide_cuda_fifo_new(4, sizeof(float), CPU);
    /* Create actors */
    actors[SRC] = (lide_c_actor_context_type*)
            lide_cuda_src2_new(e1,edge_in_h2d_0,4,4, CPU);
    actors[SNK] = (lide_c_actor_context_type*)
            lide_cuda_snk2_new(e4,edge_out_d2h_0,4,4, CPU);
    actors[H2D_0] = (lide_c_actor_context_type*)
            lide_cuda_memcpy_new(edge_in_h2d_0,edge_out_h2d_0,4,4,
            sizeof(float), GPU);
    actors[F1] = (lide_c_actor_context_type*)
            lide_cuda_f_new(edge_out_h2d_0,edge_in_d2h_0,4,4, GPU);
    actors[D2] = (lide_c_actor_context_type*)
            lide_cuda_d_new(e1,e2,4,2, CPU);
    actors[F2] = (lide_c_actor_context_type*)
            lide_cuda_f_new(e2,e3,2,2, CPU);
    actors[U2] = (lide_c_actor_context_type*)
            lide_cuda_u_new(e3,e4,2,4, CPU);
    actors[D2H_0] = (lide_c_actor_context_type*)
            lide_cuda_memcpy_new(edge_in_d2h_0,edge_out_d2h_0,4,4,
            sizeof(float), GPU);
    /* Create thread list */
    thread_list = lide_cuda_thread_list_init(NUMBER_OF_THREADS,
            actors, ACTOR_COUNT);
}
sample_graph::~sample_graph(){
    lide_cuda_thread_list_terminate(thread_list);
    lide_cuda_fifo_free(edge_in_h2d_0);
    /* free other fifos */
    /* ... */
    lide_cuda_src2_terminate((lide_cuda_src2_context_type*)actors[SRC]);
    /* terminate other actors */
    /* ... */
}
void sample_graph::execute(){
    /* Start thread list */
    lide_cuda_thread_list_scheduler(thread_list);
}
```

Fig. 36.7 Generated source code file for the example SDF graph of Fig. 36.5d

Table 36.2 Summary of standard files employed in ITSs for DICE-based testing in DIF-GPU

File name	DIF	LIDE-CUDA
`dlcconfig`	N/A	Specifies header and library paths
`dljconfig`	Specifies class paths	N/A
`makeme`	Invoke `javac` with settings specified in `dljconfig`	Invoke `nvcc` compiler with settings specified in `dlcconfig`
`runme`	Run test on Java VM	Run compiled test executable
`correct-output.txt`	Standard output if test executes as expected	
`expected-errors.txt`	Standard error output if test executes as expected	

the (Java-based) DIF package; (2) FIFO and actor implementations for application graph components; and (3) synthesized software for the targeted CPU-GPU implementation.

By applying the language-agnostic testing features of DICE described in Sect. 36.5, DIF-GPU provides a unified approach to implementing and managing tests for different components in DIF-GPU, as well as DSP applications and subsystems that are developed using DIF-GPU. A summary of standard files that are employed in the implementation of DICE-based tests in DIF-GPU is listed in Table 36.2.

To automatically test components in the DIF-GPU framework, we use the DICE `dxtest` utility. This utility recursively traverses all ITSs (individual test subdirectories) in the given test suite. For each ITS, `dxtest` first executes `makeme` to perform any compilation needed for the test, followed by `runme` to exercise the test. The `dlcconfig` and `dljconfig` scripts listed in Table 36.2 specify compiler configurations that are employed by the corresponding `makeme` scripts. For each ITS, `dxtest` compares the standard output generated by `runme` with `correct-output.txt` and the actual standard error output with `expected-errors.txt`. Finally, `dxtest` produces a summary of successful and failed tests, including the specific directory paths of any failed tests. In this way, the test-execution process is largely automated and simplified while operating within an integrated environment across the different Java, C, and CUDA components that need to be tested.

36.7 Summary

This chapter has covered the DSPCAD Framework, which provides an integrated set of tools for model-based design, implementation, and testing of signal processing systems. The DSPCAD Framework addresses challenges in Hardware/Software Codesign (HSCD) for signal processing involving the increasing diversity in relevant data-flow modeling techniques, actor implementation languages, and target platforms. Our discussion of the DSPCAD Framework has focused on its three main component tools – the Data-flow Interchange Format (DIF), Lightweight Data-flow Environment (LIDE), and DSPCAD Integrative Command Line Environment (DICE) – which support flexible design experimentation and orthogonalization across abstract data-flow models, actor implementation languages, and integration with platform-specific design tools, respectively. Active areas of ongoing development in the DSPCAD Framework include data-flow techniques and libraries for networked mobile platforms, multi-core processors, and graphics processing units, as well as efficient integration with multimodal sensing platforms.

Acknowledgments Research on the DSPCAD Framework has been supported in part by the US National Science Foundation, Laboratory for Telecommunication Sciences, and Tekes – The Finnish Funding Agency For Innovation.

References

1. Beck K et al (2015) Manifesto for agile software development (2015). http://www.agilemanifesto.org/. Visited on 26 Dec 2015
2. Bhattacharya B, Bhattacharyya SS (2000) Parameterized dataflow modeling of DSP systems. In: Proceedings of the international conference on acoustics, speech, and signal processing, Istanbul, pp 1948–1951
3. Bhattacharyya SS, Deprettere E, Leupers R, Takala J (eds) (2013) Handbook of signal processing systems, 2nd edn. Springer, New York. ISBN:978-1-4614-6858-5 (Print); 978-1-4614-6859-2 (Online)
4. Bhattacharyya SS, Plishker W, Shen C, Gupta A (2011) Teaching cross-platform design and testing methods for embedded systems using DICE. In: Proceedings of the workshop on embedded systems education, Taipei, pp 38–45
5. Bhattacharyya SS, Plishker W, Shen C, Sane N, Zaki G (2011) The DSPCAD integrative command line environment: introduction to DICE version 1.1. Technical report UMIACS-TR-2011-10, Institute for Advanced Computer Studies, University of Maryland at College Park. http://drum.lib.umd.edu/handle/1903/11422
6. Bilsen G, Engels M, Lauwereins R, Peperstraete JA (1996) Cyclo-static dataflow. IEEE Trans Signal Process 44(2):397–408
7. Buck JT, Lee EA (1993) Scheduling dynamic dataflow graphs using the token flow model. In: Proceedings of the international conference on acoustics, speech, and signal processing, Minneapolis
8. Choi J, Oh H, Kim S, Ha S (2012) Executing synchronous dataflow graphs on a SPM-based multicore architecture. In: Proceedings of the design automation conference, San Francisco, pp 664–671
9. Desnos K, Pelcat M, Nezan J, Bhattacharyya SS, Aridhi S (2013) PiMM: parameterized and interfaced dataflow meta-model for MPSoCs runtime reconfiguration. In: Proceedings of the international conference on embedded computer systems: architectures, modeling, and simulation, Samos, pp 41–48
10. Eker J, Janneck JW (2003) CAL language report, language version 1.0 – document edition 1. Technical report UCB/ERL M03/48, Electronics Research Laboratory, University of California at Berkeley
11. Eker J, Janneck JW (2012) Dataflow programming in CAL – balancing expressiveness, analyzability, and implementability. In: Proceedings of the IEEE Asilomar conference on signals, systems, and computers, Pacific Grove, pp 1120–1124

12. Falk J, Keinert J, Haubelt C, Teich J, Bhattacharyya SS (2008) A generalized static data flow clustering algorithm for MPSoC scheduling of multimedia applications. In: Proceedings of the international conference on embedded software, Atlanta, pp 189–198
13. Gagnon E (1998) SableCC, an object-oriented compiler framework. Master's thesis, School of Computer Science, McGill University, Montreal
14. Ghamarian AH, Stuijk S, Basten T, Geilen MCW, Theelen BD (2007) Latency minimization for synchronous data flow graphs. In: Proceedings of the Euromicro conference on digital system design architectures, methods and tools, pp 189–196
15. Gregg C, Hazelwood K (2011) Where is the data? why you cannot debate CPU vs. GPU performance without the answer. In: Proceedings of the IEEE international symposium on performance analysis of systems and software, Austin, pp 134–144
16. Gu R, Janneck J, Raulet M, Bhattacharyya SS (2009) Exploiting statically schedulable regions in dataflow programs. In: Proceedings of the international conference on acoustics, speech, and signal processing, Taipei, pp 565–568
17. Gu R, Piat J, Raulet M, Janneck JW, Bhattacharyya SS (2010) Automated generation of an efficient MPEG-4 reconfigurable video coding decoder implementation. In: Proceedings of the conference on design and architectures for signal and image processing, Edinburgh
18. Hamill P (2004) Unit test frameworks. O'Reilly & Associates, Inc., Sebastopol
19. Haubelt C, Falk J, Keinert J, Schlichter T, Streubühr M, Deyhle A, Hadert A, Teich J (2007) A SystemC-based design methodology for digital signal processing systems. EURASIP J Embed Syst 2007: 22. Article ID 47580
20. Hierons RM et al (2009) Using formal specifications to support testing. ACM Comput Surv 41(2):1–22
21. Hsu C, Corretjer I, Ko M, Plishker W, Bhattacharyya SS (2007) Dataflow interchange format: language reference for DIF language version 1.0, user's guide for DIF package version 1.0. Technical report UMIACS-TR-2007-32, Institute for Advanced Computer Studies, University of Maryland at College Park. Also Computer Science Technical Report CS-TR-4871
22. Hsu C, Keceli F, Ko M, Shahparnia S, Bhattacharyya SS (2004) DIF: an interchange format for dataflow-based design tools. In: Proceedings of the international workshop on systems, architectures, modeling, and simulation, Samos, pp 423–432
23. Hsu C, Ko M, Bhattacharyya SS (2005) Software synthesis from the dataflow interchange format. In: Proceedings of the international workshop on software and compilers for embedded systems, Dallas, pp 37–49
24. Hunt A, Thomas D (2003) Pragmatic unit testing in Java with JUnit. The Pragmatic Programmers
25. Janneck JW, Mattavelli M, Raulet M, Wipliez M (2010) Reconfigurable video coding: a stream programming approach to the specification of new video coding standards. In: Proceedings of the ACM SIGMM conference on multimedia systems, New York, pp 223–234
26. Kedilaya S, Plishker W, Purkovic A, Johnson B, Bhattacharyya SS (2011) Model-based precision analysis and optimization for digital signal processors. In: Proceedings of the European signal processing conference, Barcelona, pp 506–510
27. Keinert J, Haubelt C, Teich J (2006) Modeling and analysis of windowed synchronous algorithms. In: Proceedings of the international conference on acoustics, speech, and signal processing, Toulous
28. Keutzer K, Malik S, Newton R, Rabaey J, Sangiovanni-Vincentelli A (2000) System-level design: orthogonalization of concerns and platform-based design. IEEE Trans Comput Aided Des Integr Circuits Syst 19:1523-1543
29. Kwok Y (1997) High-performance algorithms for compile-time scheduling of parallel processors. Ph.D. thesis, The Hong Kong University of Science and Technology
30. Lee EA, Messerschmitt DG (1987) Synchronous dataflow. Proc IEEE 75(9):1235–1245
31. Lee EA, Parks TM (1995) Dataflow process networks. Proc IEEE 83:773–799
32. Lin S, Liu Y, Plishker W, Bhattacharyya SS (2016) A design framework for mapping vectorized synchronous dataflow graphs onto CPU–GPU platforms. In: Proceedings of the international workshop on software and compilers for embedded systems, Sankt Goar, pp 20–29

33. Lin S, Wang LH, Vosoughi A, Cavallaro JR, Juntti M, Boutellier J, Silvén O, Valkama M, Bhattacharyya SS (2015) Parameterized sets of dataflow modes and their application to implementation of cognitive radio systems. J Signal Process Syst 80(1):3–18
34. Murthy PK, Lee EA (2002) Multidimensional synchronous dataflow. IEEE Trans Signal Process 50(8):2064–2079
35. Neuendorffer S, Lee E (2004) Hierarchical reconfiguration of dataflow models. In: Proceedings of the international conference on formal methods and models for codesign, San Diego
36. Oh H, Dutt N, Ha S (2006) Memory optimal single appearance schedule with dynamic loop count for synchronous dataflow graphs. In: Proceedings of the Asia South Pacific design automation conference, Yokohama, pp 497–502
37. Pelcat M, Aridhi S, Piat J, Nezan JF (2013) Physical layer multi-core prototyping. Springer, London
38. Pelcat M, Desnos K, Heulot J, Nezan JF, Aridhi S (2014) Dataflow-based rapid prototyping for multicore DSP systems. Technical report PREESM/2014-05TR01, Institut National des Sciences Appliquées de Rennes
39. Pelcat M, Desnos K, Maggiani L, Liu Y, Heulot J, Nezan JF, Bhattacharyya SS (2015) Models of architecture. Technical report PREESM/2015-12TR01, IETR/INSA Rennes. HAL Id: hal-01244470
40. Pelcat M, Menuet P, Aridhi S, Nezan JF (2009) Scalable compile-time scheduler for multi-core architectures. In: Proceedings of the design, automation and test in Europe conference and exhibition, Nice, pp 1552–1555
41. Pelcat M, Piat J, Wipliez M, Aridhi S, Nezan JF (2009) An open framework for rapid prototyping of signal processing applications. EURASIP J Embed Syst 2009:Article No. 11
42. Plishker W, Sane N, Bhattacharyya SS (2009) A generalized scheduling approach for dynamic dataflow applications. In: Proceedings of the design, automation and test in Europe conference and exhibition, Nice, pp 111–116
43. Plishker W, Sane N, Kiemb M, Anand K, Bhattacharyya SS (2008) Functional DIF for rapid prototyping. In: Proceedings of the international symposium on rapid system prototyping, Monterey, pp 17–23
44. Plishker W, Sane N, Kiemb M, Bhattacharyya SS (2008) Heterogeneous design in functional DIF. In: Proceedings of the international workshop on systems, architectures, modeling, and simulation, Samos, pp 157–166
45. Ritz S, Pankert M, Meyr H (1993) Optimum vectorization of scalable synchronous dataflow graphs. In: Proceedings of the international conference on application specific array processors, Venice
46. Sane N, Kee H, Seetharaman G, Bhattacharyya SS (2011) Topological patterns for scalable representation and analysis of dataflow graphs. J Signal Process Syst 65(2):229–244
47. Shen C, Plishker W, Wu H, Bhattacharyya SS (2010) A lightweight dataflow approach for design and implementation of SDR systems. In: Proceedings of the wireless innovation conference and product exposition, Washington, DC, pp 640–645
48. Shen C, Wang L, Cho I, Kim S, Won S, Plishker W, Bhattacharyya SS (2011) The DSP-CAD lightweight dataflow environment: introduction to LIDE version 0.1. Technical report UMIACS-TR-2011-17, Institute for Advanced Computer Studies, University of Maryland at College Park. http://hdl.handle.net/1903/12147
49. Sriram S, Bhattacharyya SS (2009) Embedded multiprocessors: scheduling and synchronization, 2nd edn. CRC Press, Boca Rato. ISBN:1420048015
50. T Dohmke HG (2007) HG test-driven development of a PID controller. IEEE Soft 24(3):44–50
51. Theelen BD, Geilen MCW, Basten T, Voeten JPM, Gheorghita SV, Stuijk S (2006) A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In: Proceedings of the international conference on formal methods and models for codesign, Napa
52. Zaki G, Plishker W, Bhattacharyya SS, Clancy C, Kuykendall J (2013) Integration of dataflow-based heterogeneous multiprocessor scheduling techniques in GNU radio. J Signal Process Syst 70(2):177–191. doi:10.1007/s11265-012-0696-0