In Proceedings of the International Symposium on Rapid System Prototyping, Fairfax, Virginia, June 2010.

Model-based DSP Implementation on FPGAs

William Plishker, Chung-Ching Shen, Shuvra S. Bhattacharyya
George Zaki, Soujanya Kedilaya, Nimish Sane, Kishan Sudusinghe
Dept. of Electrical and Computer Engineering
and Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland
plishker@umd.edu

Tony Gregerson, Jie Liu, Michael Schulte
Dept. of Electrical and Computer Engineering
University of Wisconsin
Madison, Wisconsin

***Abstract*—Dataflow modeling provides valuable analysis and optimization tools to digital signal processing (DSP) application designers. In recent years, FPGAs have become the target of a variety of tools that utilize dataflow to achieve high-performance, while still retaining the flexibility to describe a complex, often dynamic application. However, this diversity of dataflow models, each with its own optimization techniques and level of expressibility, is an aspect of the design space that is often under-explored. To address this gap, we present an approach for supporting a set of dataflow models transparently in the same framework. We use analysis techniques and best practices to identify and verify which model each actor in an application belongs to. By taking more generally described actors and systematically identifying them as instances of specific dataflow models, we enable the automatic application of model-specific scheduling and optimization techniques, which are often much more powerful than general purpose techniques.**

I. Introduction

As field programmable gate arrays (FPGAs) become the target of more languages and design flows, often within a single system, design teams must be able to switch between these approaches during development. This can be a challenging task as different approaches have their own programming models and development environments, and are often developed with their own design teams. The original application description also often has its own programming environment to facilitate fast development of the platform-independent algorithm, which can range from general imperative languages like C, to object oriented ones like C++ or Java, to domain specific approaches like Matlab. Furthermore, FPGAs are increasingly being used in systems with other devices, which have their own design flows. This multitude of languages and programming environments make the design flow for modern FPGA systems time consuming and error prone as developers are often manually transcoding between them.

Many best practices are utilized in industrial and academic environments to help this process, such as automatically generating documentation, auto-configuration, adherence to interface specifications, and unit testing. In particular, unit testing facilitates productive design by integrating testing early into the design flow to catch erroneous or unexpected behavior in a module earlier in the design cycle. Such techniques have proven effective for many languages and platforms individually, but for systems that employ more than one of these into a single design process, these tools still leave many manual, error prone steps possible. This leads to longer design times with lower quality implementations.

In this work, we propose to enhance existing design flows with model-based design that extracts dataflow behavior and verifies cross-platform correctness of individual actors. By extracting the dataflow behavior, we free the application developer from identifying and committing to a specific model. Instead, the actors may be written using existing design practices, and the developer is then notified of the model. Applicable analysis can then be done automatically. Furthermore, with model-based development, automatic test-bench creation is possible, improving the ease with which designers can create cross-platform tests.

The *DSPCAD Integrative Command Line Environment* (DICE) [1] is a realization of managing these enhancements to the design flow. It is a framework for facilitating efficient management of design and test of cross-platform software projects. DICE defines platform and language independent conventions for describing and organizing tests. DICE runs and analyzes these tests using shell scripts and programs written in high-level languages. DICE is an open source resource which can be downloaded [2]. We use the *dataflow interchange format* (DIF) [3] as our dataflow analysis engine which can leverage the extracted dataflow models.

We demonstrate this novel test framework on modules of a triggering system for the Large Hadron Collider (LHC) [4], which includes algorithm development modules in C++ and implementation modules developed in Verilog. Using this model-based approach and the integration of DICE, we are able to make model-based design easier on the application designer and still be rigorous, agile, and evolvable.

II. Background

To give context to our model-based design approach, this section covers the dataflow models that we base our

17072/10 $25.00 (C) 2010 IEEE DOI 10.1109/RSP_2010.SS4

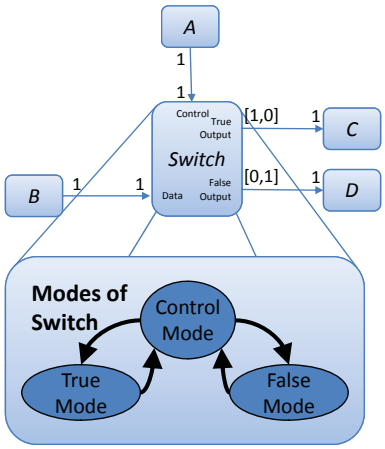

Figure 1. Boolean dataflow *switch* described in CFDF.

technique on, as well as the dataflow modeling tool we utilize for application description.

A. Dataflow Modeling

Modeling DSP applications through coarse-grain dataflow graphs is widespread in the DSP design community, and a variety of dataflow models have been developed for dataflow-based design. A growing set of DSP design tools support such dataflow semantics [5]. Designers are expected to be able to find a match between their application and one of the well-studied models, including cyclo-static dataflow (CSDF), synchronous dataflow (SDF) [6], single-rate dataflow, homogeneous synchronous dataflow (HSDF), or a more complicated model such as boolean dataflow (BDF) [7].

Common to each of these modeling paradigms is the representation of computational behavior as a dataflow graph. A dataflow graph G is an ordered pair (V, E), where V is a set of vertices (or nodes), and E is a set of directed edges. A directed edge $e = (v_1, v_2) \in E$ is an ordered pair of a source vertex $v_1 \in V$ and a sink vertex $v_2 \in V$. Nodes or *actors* represent computations while edges represent a FIFO communication links between them. In this work, we would like to not only model the application description but also have functional simulation for which we utilize functional DIF. The semantic foundation of functional DIF is *core functional dataflow* (CFDF) [8], which is capable of expressing deterministic, dynamic dataflow applications. In this formalism, each actor $a \in V$ has a set of *modes*, M_a, in which it can execute. Each mode, when executed, consumes and produces a fixed number of tokens.

For example, consider the *Switch* actor and the four SDF actors A, B, C, and D in Figure 1. SDF actors can be described with only one mode, but the *Switch* is described with 3 modes: *Control* in which one control token is read and *True* and *False* in which the data token is routed to the true and false output, respectively. While each mode has fixed production and consumption behavior, the dynamic nature of *Switch* is captured by transitions between the modes. We use this structured representation of functionality to derive the appropriate dataflow testbench for each actor.

B. Dataflow Interchange Format

To describe dataflow applications for the wide range of dataflow modeling techniques that are relevant to DSP system design, application developers can use the *dataflow interchange format* (DIF) [3], a standard language founded in dataflow semantics and tailored for DSP system design. It provides an integrated set of syntactic and semantic features that can fully capture essential modeling information of DSP applications without over-specification. From a dataflow point of view, DIF is designed to describe mixed-grain graph topologies and hierarchies as well as to specify dataflow-related and actor-specific information.

To utilize the DIF language, the DIF package has been built. Along with the ability to transform DIF descriptions into a manipulable internal representation, the DIF package contains graph utilities, optimization engines, algorithms that can prove useful properties of applications, and a C-code-synthesis framework [9]. These facilities make the DIF package an effective environment for modeling dataflow applications, providing interoperability with other design environments, and developing new tools.

C. Related Work

Typically the tools designers employ for design and verification are language specific [10], [11]. More than just a syntactic customization, such frameworks are often tied to fundamental constructs of the language. For example, in CppUnit, a unit test inherits from a base class defined by CppUnit. A test writer then overloads various methods of the base class to put the specific unit test in this framework. Tests requiring the specific features that leverage the constructs of a language (e.g. in an object oriented language, checking that method exhibits the proper form of polymorphism) are well served by these approaches. Furthermore, these language-specific approaches work well when designers are using only a single language or a single platform for their final implementation. But when designers must move between languages with different constructs (like C++ to Verilog), the existing tests must be rewritten. This creates extra design effort and creates a new verification challenge to ensure unit tests between these two languages are in fact performing the same test.

Probably the most related framework to DICE is the Test Anything Protocol (TAP) [12]. Like DICE, TAP achieves language independence by defining the protocol that manages the communication between unit tests and a test harness. Individual tests (TAP producers) communicate test results to the testing harness (TAP consumers). TAP enables multi-platform and multi-language design, but only at the

communication boundary. Unit tests need only adhere to the communication design, leaving test writers with no specific language independent mechanism for writing the tests themselves. Indeed, many language specific unit tests have TAP compatible outputs so they may be hooked into a larger multilanguage testing environment.

In contrast with these other efforts, we provide a cross-platform approach to utilizing model-based utilities and unit tests writing by inferring dataflow models and leveraging primitive datatypes with DICE. Some unit test frameworks have data generators, but DICE encourages designers to think of module interface in terms of streaming data primitives. DICE captures these input/output sequences in files and then ensures the output files match with a structured build and run framework. These assumptions allow test writers in DICE to build more complete solutions than a test communication protocol alone.

III. Cross-Platform Model-Based Design Approach

To bring model-based design to cross-platform design flows, we attempt to augment key features of the design process. Figure 2 illustrates the traditional design flow of first performing application exploration in a high-level development environment to achieve correct functionality and do preliminary planning for implementation. Once the design is finalized, the application is either synthesized or transcoded to the programming environment of the target platform. This implementation of the application often leverages some existing library to best leverage the device. Our proposed approach augments this design flow by detecting the models of the actors used in the application and using this information to provide more analysis to the application exploration phase and improve testing. We also present here a cross platform based design tool which is able to verify the functionality of the final implementation and the original high level description of it.

A. High-level Application Specification

System development often involves an initial application description in a design environment, which is then manually transcoded and tuned to target the final design platform. Often separated by languages, tools, and even different teams, going from an initial application description to a final implementation tends to be a manual, error-prone, and time-consuming problem. To improve the quality and performance while reducing development time, a cross platform design environment is needed that accommodates both early design exploration and final implementation tuning. One could make the effective use of the initial higher level application specification for testing purposes. Such a description allows testing the functionality of the application specification for a valid set of inputs. This functionally correct implementation could then be used as a benchmark

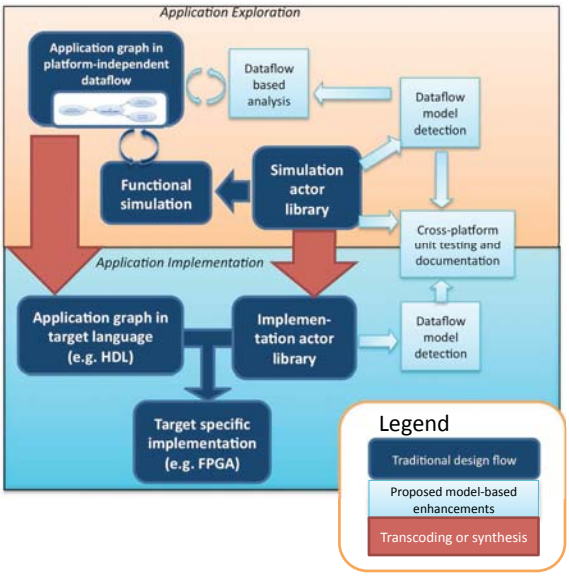

Figure 2. Traditional design flow augmented with proposed model-based features

as the development of underlying subsystem(s) in the application proceeds on various platforms using different tools. However, with the DICE framework, one does not need to redesign the test suites designed for the initial application specification.

DIF provides one such tool for model-based design and implementation of signal processing systems using dataflow graphs. A designer starts with a platform-independent description of the application in DIF. This structured, formal application description is an ideal starting point for capturing concurrency and optimizing and analyzing the application. After settling on the DIF description, a designer can refine this description to a high-performance implementation by employing platform specific tools including compilers, debuggers, and simulators. Any transcoding or platform specific enhancements are accommodated by DICE via its flexible but standardized build and test framework. This allows designers to utilize the same design framework at inception as they do at final implementation. Software developed jointly with DIF and DICE enjoys a single, cross platform software management framework, where verification of modules is handled consistently throughout each phase of development. If DIF is used as the reference description, transcoding effort is saved by having a formal, unambiguous application description to base the implementation on. Quality is controlled with a high degree of automation through the direct reuse of unit tests in DICE.

The DICE framework can be applied for testing each of the individual modules, subsystems, or even an entire application. In case of testing an individual module, we specify valid inputs and expected correct outputs for that module using concepts mentioned in Section III-C. We

create wrapper modules consisting of an individual module, its valid input interface, and the output interface. Such a wrapper module can then be tested independently of other modules. This functionally correct module description can then be used to develop platform or language specific implementations of that module. We could use the same test suite for testing and verifying the correctness of such modules using the features of DICE, as explained in Section III-D.

B. Model Detection

If an application is not initially described in a dataflow formalism, extracting the exact dataflow model of the application or its subsystems can be a challenging or even intractable task. In this work, we propose to leverage a more pragmatic approach to enable more powerful analysis with models with applications. When actors are described in a dataflow specific structural form such as CFDF, the identification of the model of actor is a case of pattern matching. For example, an actor with only one mode must be SDF, while an actor with a circular chain of modes must be CSDF.

When actors are not described in such a formal representation (or the mode descriptions themselves are not analyzable), we treat the actor as a black box, relying strictly on its observed behavior to make inferences about what model it might be. As with unit testing, we rely on significant coverage of the behaviors of an actor, instead of requiring a formal solution to analyze the code of the actor itself. While this requires good tests to exercise all of the behaviors that would occur during running the application in a real world environment, the tool need not understand language or build process of the target-specific actor. Designers are free to focus on the correct, efficient implementation of the actor, while a tool can provide identification and then the proper dataflow analysis or optimization of the application without intervention from the designer.

As described in Section II-A, there are a number of possible models that an actors behavior could belong to. Generally speaking the simpler the model and the more static the behavior, the more optimizable and analyzable that model tends to be. For the model-based testing, our aim identify a model that lends itself to the strongest analysis possible. Assuming we have a testing sequence which is a set of sequences to each of the inputs of an application graph which are not connected to a source. In the case of unit testing, the inputs to the graph are exactly the inputs to the actor. We instantiate a number of tests to identify:

- *Determinism:* Repeatability of a test is often assumed for most unit testing approaches. We recognize this widely held concept in the world of dataflow as *determinism*: a deterministic application is one in which an input sequence of data always produces the same output sequence. Various models of dataflow guarantee this, but each model relies on the actors each being individually deterministic. By repeating tests and confirming identical sequences of outputs are the result, we are verifying the underlying assumption of determinism.
- *Dataflow Model Identification:* Given a deterministic actor, we can monitor the production and consumption of each firing for each input sequence given. By making a record of this, we can make a limited inference on the apparent dataflow model represented by this behavior. Actors can be tagged with the most restrictive model that can capture all of the firings observed of that actor. For example, if the only firing behavior seen always consumed one token on each input port and produced one token on each output port, then the actor is considered to be HSDF. If the firing behavior was fixed, but consumed or produced more than one token, then the actor is identified as SDF. Even BDF actors may be identified by consistently associating the value of one input token with a fixed firing behavior.
- *Statefulness:* If the actor adheres to a repeatable firing pattern as revealed by the last step, then it may also be checked for the presence of state. A dataflow graph with stateless actors is able to replicate actors and their connections to expose more parallelism and potentially improve the performance of the final implementation. To test for such actors, we leverage the record of firing behavior from the model identification step to move a block of tokens that corresponds to a single firing in the original sequence between two other blocks of tokens that correspond to different firings. If the actor is stateless, the values of the original output will appear as a block shifted by the same number of firings.

C. Interface Specification for Tests

Most of the tools available for unit testing require the test inputs and outputs to be specified in a way that is platform or application language dependent. Such dependence makes it difficult to use tests designed for a particular platform or application language across other platforms or languages. Our framework provides a solution to this problem by allowing test inputs and outputs to be specified in a manner that is platform and application language independent. While using DICE framework, test inputs and outputs are of primitive data types. The valid sets of such inputs and the corresponding correct outputs are solely dependent on the functionality of the module being tested. A given set of valid inputs for a module should produce a corresponding expected correct output depending upon the functionality of that module and is independent of the application language or implementation platform.

DICE framework has platform independent test features and utilities that help running tests with test suites so that tests may be uniformly created and aggregated. It has scripts to build the source code using plugins that call the language-

specific compiler. This built code is tested for a valid set of inputs to generate the corresponding output. The resultant output is then compared with the expected correct output as determined by the functionality of the module being tested.

D. Model-based Testing

In order to accommodate cross platform operation, the DICE engine consists of a collection of utilities implemented as Bash scripts, C programs, and python scripts. By writing DICE based on these open-source command line interfaces and languages, DICE is able to operate on different platforms such as Windows (equipped with Cygwin), OS-X, Solaris, and Linux. This gives DICE a wide base from which to integrate specific design flows. From this base, we have architected a testing approach that is applicable across design flows.

To implement a unit testing suite, the developer provides a test reference for the functional behavior of the *module under test* (MUT). This reference consists of a set of inputs and the set of outputs that are produced as a result of correct processing of those inputs. Unit testing is performed by executing the module and comparing the actual output with the correct expected behavior. In the event of a module throwing an expected error (e.g. a designer is trying to see that when an input condition is violated, the application returns the proper error), this can be added to the test reference as well.

The basic component of the DICE unit testing framework is a directory referred to as an *individual test subdirectory* (ITS). The test suite consists of several ITSs that test the different behaviors of the MUT. Every ITS name must start with the prefix "test" (e.g., test01, test02, test-square-matrix-1, test-square-matrix-2, etc.). By doing so, changing the ITS prefix to any word other than "test" will exclude it from the test suite. An ITS consists of the following required files:

- A *readme.txt* file that contains an explanation of what part of the MUT functionality this ITS tests.
- A *makeme* script that contains all compilation steps required before running the test. It is important to note that *makeme* does not compile the source code of the MUT, but it compiles any additional code required for the test (e.g., a driver program that supplies the MUT with inputs and prints its outputs).
- A *runme* script that runs the test. The contents of *runme* may vary depending on the type of the MUT. For example, when testing a C program, one may need to just run an object file, but for a Verilog module, a hardware simulator such as ModelSim may need to be run. Also the *runme* file may contain a call to other executables that perform different post processing on the MUT output before doing the comparison with correct-output and expected-error files.
- A *correct-output.txt* file that contains the correct standard output that has to be produced by the test (i.e, after running the *runme* file).
- An *expected-errors.txt* file that contains the error messages that the test is expected to produce on standard error. This file is useful when the ITS checks for the errors that the MUT should be catching.

The basic DICE utility that makes use of the required files and exercises the test suite is called *dxtest*. By running *dxtest* from a certain directory, it recursively traverses all subdirectories that begin with the prefix "test". A subdirectory that contains a *runme* file is considered as an ITS. When *dxtest* traverses an ITS, it first executes the *makeme*, followed by the *runme*. It then compares the actual output generated after running *runme* with the *correct-output.txt* and the actual standard error output with the *expected-error.txt*. After traversing all the subdirectories, a summary of successful and failed tests is produced.

In the example in Figure 3, there are three implementations under test for the same module: DIF, C++, and Verilog. The input and output patterns are common to each of these tests and reside in the *util* directory. While each language has customized build and simulation scripts in *makeme* and *runme*, and *correct-output.txt* and *expected-error.txt* tailored to their simulation environments, the fundamental inputs and outputs are directly shared between these platforms. By using such a framework that automates the process of test verification, any change to the basic MUT can be verified not only for the new functional correctness, but also to ensure that it does not ruin a previous correct behavior. This also enables an incremental code development, and reduces the development and verification time.

In addition to the DICE unit testing framework, many other utilities are provided in the DICE package to facilitate the command line interface. DICE provides directory navigation utilities that allow one to assign a user-defined identifier to a directory, which can later be used to navigate to that particular directory without specifying the complete path. Another set of utilities are implemented to easily move files and folders between different directories.

IV. Demonstration - Trigger System of the Compact Muon Solenoid

DICE is actively being used as a test framework for the Trigger system of the Compact Muon Solenoid (CMS) Detector of the Large Hadron Collider (LHC) at CERN [13]. Systems for high energy physics (HEP) applications like the CMS trigger increasingly depend on FPGAs for data processing and communication. Increased use of and need for complex FPGA-based designs in scientific fields like HEP necessitates application development techniques more accessible to scientists and engineers specialized in the application area, but not in high performance hardware design. This naturally leads to the need for having multiple implementations for algorithm development and hardware designs for an application such as the CMS trigger.

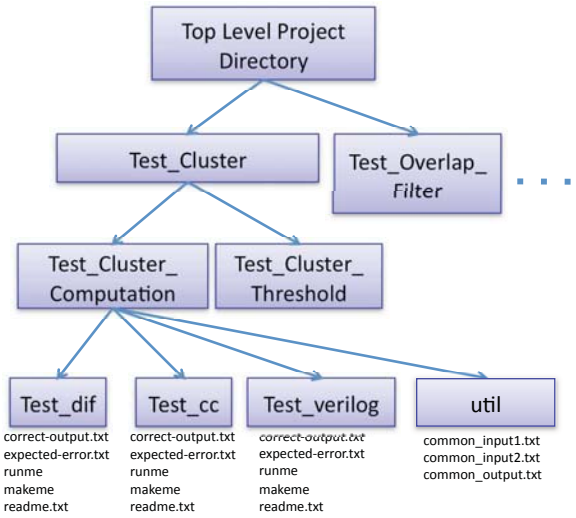

Figure 3. DICE file and directory structure of a cross-platform project using our testing approach

Table I
DETECTED MODELS OF CMS ACTORS

Actor	Inputs	Outputs	Determ-inistic	Model Detected	State
Cluster Thresh	12	12	Yes	HSDF	No
Cluster Compute	12	6	Yes	HSDF	No
Overlap Filter	8	4	Yes	SDF	No
Jet Reconstruction	1	2	Yes	SDF	No

The design of the CMS trigger, being a complex FPGA-based system, is collaborative between multiple geographically distributed research groups, each of which create their designs independently, often using different styles and techniques. This complicates system testing and integration, and hinders the use of good practices important to decrease the development time. For example, with the current CMS trigger system, dozens of teams from different institutions have contributed to the design of hundreds of boards. Individual teams use different design tools, hardware description languages, and FPGA platforms or ASICs in their designs. This non-uniform method of design has made the digital systems in the current CMS trigger difficult to maintain, test, and enhance. DICE as a platform independent framework supports projects that involve such heterogeneous programming languages and offers a unified framework for testing and integration.

By having access to a common test suite through DICE, designers can verify any changes they make to the design incrementally and independently. Furthermore because of the emphasis placed on adding tests in the test suite as an integral part of the development process, designers can have high confidence that their changes to the code are being tested comprehensively with respect to the rest of the code base.

Building on our integration of testing considerations into this project, we have explored novel synergies between dataflow-based system design and unit testing methodologies. We use the evolving test suites in the project to help ensure consistency between multiple implementations of the same actor in a design. For example, a given functional block in the CMS trigger has a Java-based version for the DIF simulation, a C++ based version for software emulation, and a Verilog version for FPGA implementation. The common DIF representation to which these actors interface ensures that the actors are designed with a standard, precise, and efficient method of communication to the enclosing application by standardizing its input and output interfaces. Unit testing provides a complementary form of integration support by validating equivalent input/output functionality across multiple versions of the same actor in an extensive and highly automated way. All three actor versions are tested with the same tests thereby ensuring that tests need not be rewritten for different languages and that the multiple implementations are consistent with each other.

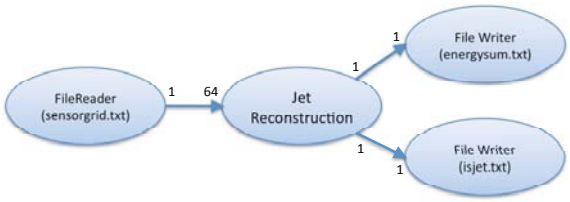

Figure 4. Testbench automatically generated from the model description of the Jet Reconstruction actor, which reads a grid of sensors to determine if a jet is present and its energy

Through the integration offered by DICE, the application development has been systematic and development time has been reduced by identifying and fixing implementation and programming bugs early in the development cycle. Inconsistencies in versioning, data representation, and deviations from design specifications and requirements are some of the errors that were identified by utilizing the unified testing framework for the CMS trigger.

The CMS trigger system is developed with a model-based design approach. Components of this model-based approach have multiple versions (C++, Verilog and DIF) that are individually tested and their models identified. Table I indicates the properties of the actor detected with thousands of inputs as the stimulus. Due to the automatic generation of testbench and streaming input data, the actors can be seamlessly hooked into the framework making the testing process largely automated and hassle-free. With more efficient dataflow scheduling techniques, the simulation time is expected to improve further. From the formal description of the functionality of a model, a DIF graph of the resulting testbench is generated (Figure 4) that hooks into input text file readers and writes to the appropriate output text files, which were supplied by the designer.

V. Conclusion

The development of electronic systems in the CMS module represents a typical problem of modern, high-performance system designs that rely on FPGA implementations. Many possible target platforms, a separate algorithmic specification in another language, and an emphasis on performance make for a long, error-prone design cycle. In this work, we have proposed to alleviate some of this difficulty by combining model-based design with cross-platform unit testing. We utilized DIF and DICE for these two aspects of the development process. For future work, we will continue to leverage our model-based application description for improved simulation times through better scheduling and provide more automation to other platforms through synthesis and analysis.

References

[1] S. S. Bhattacharyya, S. Kedilaya, W. Plishker, N. Sane, C. Shen, and G. Zaki, "Using the DSPCAD integrative command-line environment: User's guide for DICE version 1.0," Maryland DSPCAD Research Group, Department of Electrical and Computer Engineering, University of Maryland at College Park, Tech. Rep. DSPCAD-TR-2009-01, 2009.

[2] "http://www.ece.umd.edu/DSPCAD/home/software.htm."

[3] C. Hsu, I. Corretjer, M. Ko., W. Plishker, and S. S. Bhattacharyya, "Dataflow interchange format: Language reference for DIF language version 1.0, users guide for DIF package version 1.0," Institute for Advanced Computer Studies, University of Maryland at College Park, Tech. Rep. UMIACS-TR-2007-32, June 2007, also Computer Science Technical Report CS-TR-4871.

[4] C. Lefevra, "LHC: The guide," CERN, Geneva, Tech. Rep., 2008.

[5] *Using Simulink*, Version 3 ed., The MathWorks Inc., Jan 1999.

[6] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous dataflow programs for digital signal processing," *IEEE Transactions on Computers*, February 1987.

[7] J. T. Buck and E. A. Lee, "Scheduling dynamic dataflow graphs using the token flow model," in *In Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, April 1993.

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

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

[10] T. Dohmke and H. Gollee, "Test-driven development of a pid controller," *IEEE Softw.*, vol. 24, no. 3, pp. 44–50, 2007.

[11] P. Hamill, *Unit test frameworks*. O'Reilly, 2004.

[12] S. Cozens, *Advanced perl programming, 2nd edition*. O'Reilly, 2005, ch. 8.

[13] M. Collaboration, "CMS TriDAS Project: Technical design report; the trigger systems," CERN, Geneva, Tech. Rep., 2000.