Instrumentation-driven Framework for Validation of Dataflow Applications

Ilya Chukhman and Shuvra S. Bhattacharyya
Department of Electrical & Computer Engineering, and
Institute for Advanced Computer Studies
University of Maryland, College Park, MD, USA
{ilya, ssb}@umd.edu

Abstract—Dataflow modeling offers a myriad of tools in designing and optimizing signal processing systems. A designer is able to take advantage of dataflow properties to effectively tune the system in connection with functionality and different performance metrics. However, a disparity in the specification of dataflow properties and the final implementation can lead to incorrect behavior that is difficult to detect. This motivates the problem of ensuring consistency between dataflow properties that are declared or otherwise assumed as part of dataflow-based application models and the dataflow behavior that is exhibited by implementations that are derived from the models. In this paper, we address this problem by introducing a novel dataflow validation framework (DVF) that is able to identify disparities between an application's formal dataflow representation and its implementation. We demonstrate the utility of our DVF through design and implementation case studies involving an automatic speech recognition application, and a JPEG encoder.

Keywords—Dataflow graphs, models of computation, signal processing systems, validation.

I. INTRODUCTION

Dataflow modeling is an important tool often used by the designers of communication and signal processing systems to facilitate system level analysis and optimizations by exposing high level application structure. Such dataflow based analysis and optimization has the potential to increase design quality in various dimensions — e.g., by taking advantage of parallelism [1], minimizing resource utilization [2], or enhancing the use of vectorization [3], [4]. Developers then apply these optimizations to the implementation of the hardware and software components in the final system.

Formal methods and testing are two important approaches to system verification. Such methods attempt to validate that the system complies with its original specifications. Formal methods use mathematical means to prove that the system meets a specification. Although such formal methods provide correctness guarantees, their rigor often makes them impractical to prove the correctness of large, complex systems [5].

Testing methods can be used to improve the quality of a product by detecting and removing as many defects as possible, and increasing developer confidence in the proper functioning of the system. Testing methods cannot guarantee the absence of defects, but are easier to use when compared to formal techniques. As a result, testing methods are widely used to test various system properties.

During the implementation process, carefully performed testing and functional validation can mitigate functional errors. Unit tests in particular help to validate the correct behavior of each core function by comparing the generated outputs to the expected outputs [6]. However, conventional approaches to testing do not systematically uncover implementation mistakes that cause the violation of dataflow properties specified in the original design. Such violations are difficult to detect if they are not explicitly targeted in the testing process — for example, because they may not result in incorrect output values of a given functional component (actor). However, in signal processing systems, violations of the assumed dataflow properties often lead to sporadically incorrect behavior of the application as a whole, since key optimizations and design decisions, such as the employed scheduling strategy, may depend strongly on specific dataflow properties.

In this paper, we present a system level validation technique that complements functional validation that is traditionally done through unit tests. It ensures that system level concerns (e.g., scheduling and buffer management) that make use of declared dataflow properties will not fail due to bugs in the property declarations or due to bugs in the implementations that lead to deviation from the specifications. Our dataflow validation framework (DVF) can be used to detect inconsistencies in individual dataflow ports, entire actors, or dataflow subgraphs. The technique builds on a systematic approach to dataflow graph instrumentation introduced in [7]. The general DVF validation approach works by (1) having the designer specify expected dataflow behavior for components that the designer wants to test; (2) executing the application, which results in parts of the dataflow graph being instrumented to validate that the observed dataflow behavior matches the expected behavior; and (3) correcting any reported errors that result from the application execution in Step (2). In addition to supporting this dataflow-property-oriented validation approach, DVF facilitates the diagnosing and fixing of detected errors by saving the execution state of the application at the time of the violation, thereby providing the developer with insights into the cause of the detected malfunction.

We demonstrate DVF in the context of an audio and image processing applications, and show that typical dataflow properties can be validated with minimal amounts of extra processing (i.e., with low run-time overhead). The correctness of the supported validation approach relies on having a thorough set of unit-tests that exercises as much of the dataflow graph as possible. Thus, DVF provides systematic integration.
of dataflow property considerations into the general framework of unit testing.

II. RELATED WORK

By helping to alleviate the problem of system-level validation, the contributions in this paper addresses one of the major bottlenecks in design processes for signal processing systems. Our work combines aspects of dataflow modeling, application profiling, and system verification. In this section, we briefly discuss relevant background in these areas and compare our work to the current state of the art.

Many different specification models have been introduced over the years that vary in the strictness of their semantics and their utility for different application domains. Using such specifications facilitates the verification and testing processes [8]. For example, the authors in [9] describe the formal specification and verification of a multicore digital signal processor that employs 64 processors. They abstract implementation details by specifying the expected behavior from the programmer’s perspective, and then use an assume-guarantee method with a model checker called MOCHA to perform compositional verification.

Dataflow-based specification is used to describe MPEG reconfigurable video coding (RVC) to abstract away implementation specific complexities [10]. In [11], the authors propose using CAL [12] to specify the system, and then perform functional validation using the OpenDF environment [13]. The approach that we present in this paper complements the functional validation methods provided by these related works in its emphasis on validation of dataflow properties.

Various tools have been introduced to facilitate profiling and complexity analysis for signal processing systems. For example, Ravasi and Mattavelli describe the Software Instrumentation Tool (SIT), which acts as a C virtual machine and allows extraction of information related to fine-grained computation and memory utilization [14]. Like SIT, DVF also instruments the application. A distinguishing aspect of DVF is that it operates at the dataflow level, and focuses on validation of dataflow properties, thus complementing SIT’s fine-grained functional analysis.

Causation traces are introduced in [15]. Such traces are obtained from specific executions of the system under development, and represent dataflow actions that have token dependencies. These traces can then be used in offline analysis to improve scheduling and parallelization decisions.

Instrumentation-based validation (IBV) uses monitors to instrument Simulink applications and find violations of requirements [16]. The validation occurs by checking if the observed run-time behavior captured by the monitors matches that of the requirements, which are encoded as assertions. In [17], the authors demonstrate the utility of IBV by applying it to the validation of automotive controllers. While both IBV and DVF use runtime instrumentation, IBV tests the functional correctness of an application, while DVF uses instrumentation to validate dataflow properties. Although dataflow properties may influence the functional correctness of an implementation, our specialized focus on dataflow properties allows more precise identification of design defects that have such influence.

III. VALIDATION FRAMEWORK

Figure 1 illustrates our approach to validating application behavior using DVF. DVF consists of 3 general phases: the offline behavior specification phase, application setup phase, and application execution phase. During the first phase, which occurs offline, before running the application, the designer creates a specification of the expected behavior. The application setup phase consists of processing that specification and instrumenting the dataflow application. During the application execution phase, the instrumentation inserted during the second phase collects run-time information from the application context, and then validates that observed behavior with the expected behavior. The run-time information extracted during the third phase is collected by monitoring and storing information associated with how actors access their input and output edges (FIFOs) during dataflow graph execution.

In the remainder of this section, we provide further details on the behavior specification phase and application setup phase. Then in Section IV, we discuss the application execution phase.

A. Behavior Specification

In the behavior specification phase, the designer specifies the expected dataflow behavior of the actors in the design through a text file, called the behavior specification file. This specification file contains a high-level description of the expected behavior for the application components that need to be validated. The description is provided in an XML format, thereby decoupling it from the language-specific implementation of the final design. As a result, the same specification file can be used to validate the high-level language-agnostic dataflow design as well as the language-specific implementation of the final solution. By using the same specification at the early design stage and later implementation stages, the designer can guarantee consistent, correct behavior and track any errors that may have been introduced during the implementation.

An example specification file for an actor containing three ports is shown in Figure 2. Two of the ports are input ports.
conforming to the synchronous dataflow (SDF) model. Each of these ports is specified to have a consumption rate of 2 tokens per firing. The actor also has one output port, which conforms to the homogeneous synchronous dataflow (HSDF) model, meaning that it has a constant product rate of 1 token per actor firing. By design, the XML-based behavior specification file does not contain any information about the functional behavior of the associated component. This is because DVF is oriented toward orthogonalizing dataflow properties during the validation process so they can be focused on during testing, and addressed in a method that is independent of the implementation platform or language.

In dataflow graphs containing multiple instances of the same actor, the `actor id` disambiguates which instance needs to be instrumented. Similarly, `port numbers` are used to identify specific ports within a DVF behavior specification file.

```xml
<actor>
  <name="adder"/>
  <id=1/>
  <port>
    <number=1/>
    <direction=input/>
    <type=SDF/>
    <behavior=2/>
  </port>
  <port>
    <number=2/>
    <direction=input/>
    <type=SDF/>
    <behavior=2/>
  </port>
  <port>
    <number=3/>
    <direction=output/>
    <type=HSDF/>
  </port>
</actor>
```

Fig. 2. The contents of a DVF behavior specification file for an adder actor.

### B. Application Setup

The behavior specification file is applied as input to the application setup phase, and consists of two stages — (1) specification processing and (2) instrumentation — as illustrated in Figure 1. The specification processing stage parses the behavior specification file to construct and initialize interfaces that will be employed in the instrumentation stage. These interfaces will specify how instrumentation code interacts with the associated dataflow graph actors and edges.

For the example of Figure 2, the specification processing parser extracts the expected dataflow behavior for each port of the `adder` actor. This XML parser captures any dependencies that may exist between the ports, and allows the instrumentation and validation stages to utilize such dependency information. For example, a Boolean dataflow (BDF) [18] actor containing a control input port would require the other input ports to specify behaviors for the `TRUE` and `FALSE` cases. This dependency would be captured in the specification processing stage and encoded in the expected and observed behavior structures, which will be applied during application execution to provide appropriate instrumentation and validation.

In DVF, the interfaces that are initialized in the specification processing stage dictate the instrumentation required to validate the dataflow behavior of each component. Instrumentation code is generated automatically from the behavior specification file. This includes code to validate the connection of the correct ports in the implementation with the corresponding actors, and results in the insertion of lightweight segments of monitoring code that execute before and after each instrumented actor. We refer to these code segments as **DVF monitoring code segments (DMCSs)**.

To provide this kind of monitoring code, we have generalized the instrumentation technique introduced in [7] such that the `pre-invoke instrumentation` and `post-invoke instrumentation` operations can be applied in a manner that is independent of the type of scheduling strategy used.

The DMCSs for each instrumented port execute an associated `statistics collection function` before and after the corresponding actor executes. The type of statistics collection function applied is determined by the port behavior, as specified in the behavior specification file. Both the specification processing and instrumentation stages are parts of the application setup phase, as illustrated in Figure 1, and are carried out before the application starts executing.

In the example of Figure 2, DMCSs would be added for each port, resulting in a dataflow graph model of the form depicted in Figure 3. As the application executes, the DMCSs for each instrumented port invoke the `statistics collection` process before and after **Actor 1** fires.

![Fig. 3. The expected behavior for each port, shown in purple, is extracted from the behavior specification file during the specification processing stage. DMCSs, represented in blue, are added to each instrumented port such that the associated statistics collection function is called before and after the actor fires.](image)

### IV. APPLICATION EXECUTION PHASE

The execution phase of DVF contains two stages, as illustrated in Figure 1. In this section, we present details on the operation of these two stages.

#### A. Statistics Collection

The separation of specification processing, instrumentation, and statistics collection activities into separate stages results in a flexible framework capable of validating a variety of dataflow behaviors, while simultaneously allowing reuse of common validation components. The `statistics collection` stage collects selected statistics of each instrumented component when the associated actor fires. The type of data collected varies for each port and depends on the assigned statistics collection function, which is selected from a library of available functions during the instrumentation stage. The flexibility offered by DVF enables the designer to add new statistics collection functions to the library as desired.
The validation of the behavior for static dataflow models, such as HSDF and SDF, requires collecting the consumption and production rates for each instrumented port when the associated actor fires. The resulting statistics collection function collects the FIFO populations before and after the instrumented actor fires, thus enabling computation of the consumption and production rates.

Collection of data required to validate dynamic dataflow behavior generally requires more complex statistics collection functions that collect more kinds of data. For example, the statistics collection function for a BDF actor, whose behavior varies depending on the input to the control port, collects the value of the token at the control port as well as the consumption and production rates for the data ports.

Figure 4 shows that the collected information from an instrumented component is stored in an entity called the observed behavior structure. The observed behavior structure in DVF can be viewed as a placeholder for an arbitrary data structure that is used to store and organize collected statistics during execution. The complexity of the observed behavior structure depends on the behavior being validated. For example, for an SDF model, the observed behavior structure can be implemented as a vector, where each element corresponds to the consumption or production rate of a specific actor port, and for cyclo-static dataflow (CSDF) [19], one can use an array of vectors with each port represented by a separate vector of non-negative integers.

**B. Behavior Validation**

The behavior validation stage involves comparing the observed behavior structure populated during the statistics collection stage with the expected behavior structure that is defined in the specification processing stage. To ensure that the validation process does not report false errors from the beginning of application execution, the observed behavior structure is initialized to the expected behavior. Such an initialization translates to the premise that the application behavior is assumed to be valid until proven otherwise.

Because the DVF validation process is carried out concurrently with application execution (rather than as a kind of trace-based post-processing), an invocation of the validation process does not need to compare the observed and expected structures in their entirety. Only the observed behavior structure elements set during the most recent actor firing need to be compared to the corresponding elements in the expected behavior structure, thus resulting in a significant reduction in the required processing. For example, with the observed behavior vector in Figure 4, only the most recent elements populated by the statistics collection function would be compared to the corresponding elements in the expected behavior vector.

Matching of the compared elements in the observed and expected behavior structures results in the pass of the validation check, in which case the application continues to execute and the validation process continues with the statistics collection stage, as shown in Figure 1. However, if the validation fails, execution of the application terminates with an error report, and the execution state of the application is saved to a file to facilitate the diagnosis and repair of the detected “dataflow behavior malfunction”. If DVF is operating within an actual application deployment (e.g., rather than during testing), a controlled shutdown or warning indication can be initiated upon detection of a dataflow behavior malfunction. Such proactive fault handling is often preferable to silently allowing the system to enter an invalid execution state that is related to mismatches between design time assumptions and actual implementation characteristics.

**V. DVF Application Examples**

To further concretize DVF, we have used the framework to validate dataflow properties for an automatic speech recognition (ASR) application and a JPEG encoder. In this section, we report on our experiments with these applications.

**A. Experimental Setup**

The dataflow properties for each component being validated were encapsulated in a separate behavior specification file (as described in Section III). A corpus consisting of 20 audio files was used as input to the ASR application while validating the dataflow properties for each actor. A corpus of 12 bitmap (BMP) files was used as input to the JPEG encoder application. Each experiment was repeated 50 times per input file to better characterize the effects of DVF in the presence of background applications. These experiments were conducted on an Intel(R) Core(TM) i7-2675QM 2.20 GHz machine with 6 GB of RAM.

**B. Automatic Speech Recognition Example**

We implemented the embedded automatic speech recognition (ASR) algorithm developed by Phadke et al. [20] using the lightweight dataflow environment (LIDE) [21]. In particular, we used LIDE-C, which provides libraries and application programming interfaces to construct dataflow actors and graphs using the C language. The ASR system functionality was logically separated into 7 actors, as shown in Figure 5. These actors are described as follows.

The reader actor detects the voiced signal and partitions the 8 kHz-sampled signal into frames that span 256 samples (31.25 ms) each. The pre-emphasis actor processes each frame by applying a first order high-pass filter to compensate for variations in the low and high frequency components of
speech. The framing actor applies a Hamming window to the overlapping frames. The FFT actor computes the spectrum of the windowed signal. The feature extraction actor derives the Mel Frequency Cepstral Coefficients (MFCCs) of the signal by (a) applying the Mel filter bank to the frequency spectrum to compute the Mel spectrum, and then (b) applying an inverse discrete-cosine transform (IDCT) to the logarithm of the Mel spectrum. The MFCCs are managed as a vector of 15 coefficients. This vector is computed for each speech frame. The matching actor uses a dynamic time warping (DTW) algorithm to find the best match between the input signal and the pre-collected database of speech signals. The matching actor outputs the database index and the distance between the input signal and the corresponding best match in the database. The writer actor saves the results of the matching actor to a text file.

Dataflow properties for each ASR actor are summarized in Table I. These properties include the dataflow model, the number of input and output ports, and the consumption and production rates. Here, by the “dataflow model” of the actor, we mean the model that the designer declares the actor to conform to or the most specialized model that has been determined for the actor through prior application of relevant analysis tools, such as model detection [7]. The reader produces one extra token corresponding to an SNR value every 20 frames, making it a CSDF actor (the notation in Table I shows the actor producing 257 tokens during the first firing and then producing 256 tokens during each of the next 19 firings). The sizes of the FIFOs connecting ASR actors were adjusted several times during the development process. During one of the design iterations, an error was introduced that caused the ASR application to reach a deadlock because of a misconfigured FIFO between the pre-emphasis and framing actors. DVF was able to identify both actors responsible for the deadlock and provided information about the behavior specification violations that helped in repairing the misconfiguration.

The setup time needed to process the behavior specification and instrument the dataflow graph is shown in the right column of Table I. From this, we see that the setup time is 220 μs on average. The average time for the entire ASR application is 10 ms without DVF. Thus, the DVF setup adds about 2% overhead to the runtime of the ASR system.

Recall that in addition to the application setup stage, which occurs once at the start of the application, DVF includes statistics collection and behavior validation stages, which occur every time the instrumented component executes. The runtime overhead caused by DVF for each instrumented component of ASR is shown in Figure 6. For each actor, the figure shows two box plots, where the left plot shows the baseline execution time without DVF and the right plot shows the execution time with the actor being validated using DVF. From these results, we see that the runtime validation overhead was on average 7% of the execution time for the actors tested.

C. ASR Application: Results

Applying DVF to the ASR application in this case study enabled detecting and correcting several critical errors caused by violations of various dataflow assumptions.

Incorrect Buffer Size
A change to the behavior of the reader actor was not propagated to the FIFO connecting it to the rest of the dataflow graph. DVF was able to detect the reader attempting to output more data than what was specified in its behavior specification.

SDF Behavior Violation
To optimize for varying signal-to-noise (SNR) in the input speech signal, the reader was augmented to calculate SNR at the start of every speech sample. During initialization for a given speech sample, the optimized reader actor produces 256 tokens (signal samples), followed by a single token that encapsulates the SNR value. Ensuing invocations of reader result in the output of 256 tokens, as described in Table I, for the remaining processing of the current speech signal. DVF detected the discrepancy between the actual dataflow behavior of the optimized reader, and the specified (obsolete) behavior, which was based on the original reader. This allowed quick detection of the error and corresponding repair of the overall dataflow schedule.

Deadlock Detection
The sizes of the FIFOs connecting ASR actors were adjusted several times during the development process. During one of the design iterations, an error was introduced that caused the ASR application to reach a deadlock because of a misconfigured FIFO between the pre-emphasis and framing actors. DVF was able to identify both actors responsible for the deadlock and provided information about the behavior specification violations that helped in repairing the misconfiguration.

The results of our validation experiments are summarized in Table III. Here, EIDF stands for enable-invoke data-
TABLE II. DATAFLOW PROPERTIES OF ACTORS IN THE JPEG ENCODER.

<table>
<thead>
<tr>
<th>Actor</th>
<th>Dataflow Model</th>
<th>#Inputs/Outputs</th>
<th>Cons./Prod. Rates</th>
<th>Ver. Setup Time μs</th>
</tr>
</thead>
<tbody>
<tr>
<td>BMP Reader</td>
<td>HSDF</td>
<td>1/1</td>
<td>1/1</td>
<td>232</td>
</tr>
<tr>
<td>DCT</td>
<td>HSDF</td>
<td>1/1</td>
<td>1/1</td>
<td>204</td>
</tr>
<tr>
<td>Huffman Encoder</td>
<td>HSDF</td>
<td>3/0*</td>
<td>1/0</td>
<td>213</td>
</tr>
<tr>
<td>JPEG Prep</td>
<td>EIDF</td>
<td>2/2</td>
<td>1/6</td>
<td>196</td>
</tr>
<tr>
<td>Quantizer</td>
<td>HSDF</td>
<td>1/2</td>
<td>1/1</td>
<td>233</td>
</tr>
</tbody>
</table>

*writes the data to a file

flow, which is a very general (expressive) dynamic dataflow model [22]. DVF found that our initially presumed dataflow model for the JPEG prep actor was incorrect. Thus, optimizations that were made on the assumption that all the actors in the graph exhibit static dataflow behavior were also incorrect and had to be modified.

The setup time needed to process the behavior specification and instrument the dataflow graph is shown in the right column of Table II. This setup time is seen to be 216 μs on average. The average time to run the entire JPEG encoder application is 15 ms without DVF. Thus, the DVF setup adds less than 2% overhead to the runtime of the JPEG encoder.

As discussed above, DVF includes statistics collection and behavior validation stages, which occur every time the instrumented component executes. The two right columns in Table III can be used to examine the DVF overhead for the JPEG encoder actors. On average, the baseline configuration (i.e., execution without DVF) runs 8% faster than the configuration with DVF. Thus, the overall overhead caused by DVF is less than 10% when compared to the baseline.

VI. CONCLUSION

In this work, we have presented a framework, called dataflow verification framework (DFV), for validating dataflow properties during design and implementation of signal processing systems. We have demonstrated the utility of DVF using case studies involving automatic speech recognition and JPEG encoding. Using DVF, we were able to identify and quickly diagnose and repair several violations of dataflow properties. Our results show that DVF adds about 7% execution time overhead to the existing application, making it a suitable approach both for design-time testing and for run-time fault detection. Useful directions for future work include extending the proposed methods in the context of run-time fault detection to help drive efficient and reliable system reconfiguration for fault recovery.

TABLE III. DVF RESULTS FOR THE ACTORS IN THE JPEG ENCODER.

<table>
<thead>
<tr>
<th>Actor</th>
<th>Presumed DF Model</th>
<th>Actual DF Model</th>
<th>Baseline Time μs</th>
<th>With DVF Time μs</th>
</tr>
</thead>
<tbody>
<tr>
<td>BMP Reader</td>
<td>HSDF</td>
<td>HSDF</td>
<td>2245</td>
<td>2285</td>
</tr>
<tr>
<td>DCT</td>
<td>HSDF</td>
<td>HSDF</td>
<td>1.02</td>
<td>1.03</td>
</tr>
<tr>
<td>Huffman Encoder</td>
<td>HSDF</td>
<td>HSDF</td>
<td>1.24</td>
<td>1.47</td>
</tr>
<tr>
<td>JPEG Prep</td>
<td>HSDF</td>
<td>EIDF</td>
<td>6.86</td>
<td>7.39</td>
</tr>
<tr>
<td>Quantizer</td>
<td>HSDF</td>
<td>HSDF</td>
<td>1.05</td>
<td>1.16</td>
</tr>
</tbody>
</table>

REFERENCES