Testing Properties of Dataflow Program Operators

Zhihong Xu
University of Nebraska
Lincoln, NE
zxu@cse.unl.edu

Martin Hirzel
IBM Watson Research
Yorktown Heights, NY
hirzel@us.ibm.com

Gregg Rothermel
University of Nebraska
Lincoln, NE
grother@cse.unl.edu

Kun-Lung Wu
IBM Watson Research
Yorktown Heights, NY
klwu@us.ibm.com

***Abstract*—Dataflow programming languages, which represent programs as graphs of data streams and operators, are becoming increasingly popular and being used to create a wide array of commercial software applications. The dependability of programs written in these languages, as well as the systems used to compile and run these programs, hinges on the correctness of the semantic properties associated with operators. Unfortunately, these properties are often poorly defined, and frequently are not checked, and this can lead to a wide range of problems in the programs that use the operators. In this paper we present an approach for improving the dependability of dataflow programs by checking operators for necessary properties. Our approach is dynamic, and involves generating tests whose results are checked to determine whether specific properties hold or not. We present empirical data that shows that our approach is both effective and efficient at assessing the status of properties.**

I. Introduction

Dataflow programming languages represent programs as graphs of data streams and operators. This representation facilitates parallelization, since each operator can run independently, subject only to availability of data on its input streams. Furthermore, these languages feel natural to programmers who wish to process large amounts of data. Dataflow languages have been the subject of much research, as chronicled by several survey papers [13], [14], [23]. Moreover, thanks to the advantages of parallelism and a data-centric paradigm, dataflow languages have received a lot of commercial attention for their ability to analyze "big data".

There are many examples of dataflow programming languages and systems for executing dataflow programs. Pig Latin [17], FlumeJava [2], and other languages compile to MapReduce [5], a system that parallelizes programs over hundreds of machines and has been used in building Google's search index. While MapReduce provides a "batch" approach for executing dataflow programs, another execution approach is stream processing, which continuously analyzes data streams online as they are produced. Languages that support stream processing include StreamIt, which emphasizes optimizations for media streaming kernels [24], and the Streams Processing Language (SPL), which has use-cases in telecommunications, healthcare, financial trading, and several other domains [9]. Systems for executing stream processing dataflow programs include Borealis [1] and IBM's InfoSphere Streams [11], both of which run on clusters.

The semantics of dataflow programs depend on the semantic properties of the individual operators they employ. These properties include determinism, selectivity, blocking, statefulness, commutativity, and partition-isolation (we define these in Section III). Unfortunately, in current research and practice, operator properties frequently remain unchecked. This can have serious consequences. For example:

- A *confluent* dataflow graph is a graph in which two paths converge on a single operator. This operator must either be commutative, or enforce data ordering using blocking or state [22]. If none of these properties hold, the entire program is non-deterministic. Furthermore, blocking operators may cause deadlocks [15], [21].
- The *MapReduce programming model* assumes that the Map operator, which is used to filter data and partition it into subsets by key, is stateless, and the Reduce operator, which is used to aggregate data, is partition-isolated [5]. Languages meant to run on MapReduce do not check these properties for user-defined code [2], [17]. If the properties do not hold, MapReduce may yield unpredictable results.
- *Compiler optimizations* for dataflow programs rely on operator properties. For instance, synchronous dataflow languages use operator selectivity for scheduling and buffer allocation [14]. Some parallelizers require stateless operators [24], while others accommodate stateful operators that satisfy partition-isolation [20]. If properties are unknown, the program cannot be optimized; if properties are unreliable, the program may be optimized incorrectly.

These examples illustrate that operator properties offer clear benefits. Further, operator properties are not overly difficult to specify, because an operator developer need only indicate whether a property holds for an operator or not. Even so, once specified, the correctness of operator properties should be verified, and in the absence of specifications, it would still be useful to have a means for determining whether particular properties hold for given operators.

One approach that might be used to check operator properties for dataflow programs is static analysis. This is challenging for several reasons. Operator code often uses pointers and multi-threading [1], leading to a large analysis state. This causes static analyses to be expensive and to produce overly conservative results. Another issue is that real-world dataflow programs tend to be multi-lingual. For example, SPL programs often use operators written in SPL, C++, or Java [9]. Static analyzers need to handle each language involved, as well as cross-language interactions. A third impediment to static analysis is code-generation. For example, operators for

XML processing [16] or composite event detection [8] are actually mini-compilers, and it is difficult to statically analyze a compiler to determine properties of the code it generates.

This paper presents *a dynamic-analysis approach for verifying properties of dataflow operators*. Given a property and an operator, our technique generates tests and checks their results to observe whether the property is violated. The analysis results are easy to understand; they either indicate that no counter-evidence for the property was found, or, if the property is violated, they include a concise concrete input as evidence for why it does not hold.

Our approach is analogous to the "unit testing" of individual components that make up a "traditional" software system. In dataflow programs these components are operators, supplied with dataflow processing systems or created by engineers to enact particular operations. By applying our approach to these operators, engineers can help ensure that the building blocks on which their programs are founded behave appropriately, prior to fielding entire programs in which failures can be more difficult to detect and faults can be more difficult to localize.

In this paper we make three significant contributions:

1) We are the first to *formally define* core properties of operators in dataflow languages, including the properties of determinism, selectivity, blocking, statefulness, commutativity, and partition-isolation.
2) We describe a *testing methodology* for checking whether these operator properties hold. Our methodology requires only the operator-under-test. No prior test inputs are needed, and the operator may be written in any language, and may even use code-generation. Our methodology gains efficiency by sequencing the consideration of properties intelligently, in a manner that allows us to check fewer properties overall.
3) We present *empirical results* on a suite of 56 SPL operators [9] that are widely used by engineers to create dataflow programs. Our results compare several test generation techniques across our target properties, and show that our approach is both effective and efficient.

II. Background and Related Work

A. Dataflow Program Terminology

Dataflow programs involve streams and operators. We distinguish between operator *instances* and operator *definitions*. An operator instance (at the program level) is a vertex in a dataflow graph. An operator definition is a configurable blueprint for operator instances.

Figure 1 shows a simple financial streaming application. Operator instance sepTQ separates a market feed into trades and quotes. The instance avgPrice of the Aggregate operator computes, for each stock symbol, the average price over the last 30 seconds. The instance match of the Join operator finds deals (higher-than-average quotes). The instance sumPrice of the Average operator adds the profit from all deals, not partitioned by stock symbol.

Different instances of the same operator can have different properties. For example, in Figure 1, the avgPrice instance of Aggregate is partitioned, whereas the sumPrice instance is not.

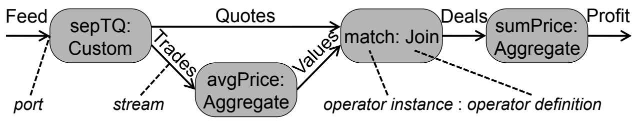

Fig. 1. Sample application

Therefore, the objects of analysis in this paper are operator instances, not operator definitions. That is to say, operator instances constitute the basic "unit" of dataflow programs that our approach tests individually.

The point at which a stream connects to an operator is called a *port*, as shown in Figure 1. We are concerned only with operators that have at least one input and one output port.

A *stream* is an ordered sequence of data items and punctuations. A *data item* is a stream element that carries payload information of interest to the application. In this paper, we are not concerned with the format or representation of data items; they may be as simple as integers or as complicated as records with deeply nested data structures. A *punctuation* is a control signal that carries no further information; it is used to separate batches of data items, for example, to indicate *window* boundaries. (Because streams are conceptually infinite, reliable, first-in first-out communication channels, many operators use the window concept to limit the number of data items in the stream. For example, avgPrice in Figure 1 uses data items from the most recent 30 seconds.)

We assume a model in which the arrival of a data item or a punctuation at an input port of an operator triggers an operator *firing* [22]. When an operator fires, it performs some computation, which may cause data items and punctuations to be submitted to some or all of its output ports.

B. Related Work

In our review of prior work, we could find no attempts to formally define the properties of dataflow operators. Further, while we found some static analyses capable of checking some properties, we found no dynamic analyses or testing techniques for determining semantic properties of operators.

Olston et al. [17] present a technique for generating example data for dataflow programs. They use a dynamic test case generation approach, as we do. While they create example data relevant to the entire behavior of a whole dataflow graph, we focus on specific properties of single operator instances. One drawback of their technique is that it requires sample data, as well as elaborate specifications for each operator ("equivalence classes" and "inverting computation"). In contrast, our technique requires only the subject operator instance.

Most of the remaining related work on checking properties of dataflow operators uses static analysis. Static analysis can produce false positives, while dynamic analysis can produce false negatives. It is meaningless to say that one or the other is better in general; for our task, either could be useful. We begin with dynamic analysis because we need to analyze multiple languages including generated code, which would be difficult with static analysis. Another advantage of dynamic analysis in our setting is that it can easily generate concrete evidence for a true positive. This is more difficult with static analysis.

Rinard and Diniz [19] present a static analysis for commutativity in a C++ subset, which they use for auto-parallelization. Raman et al. [18] describe an analysis that breaks the body of a C loop into dataflow operators, and also discovers operator statefulness, which they use for auto-parallelization. Jayaram and Eugster [12] analyze EventJava to discover inter-operator properties that help reduce communication overhead (in contrast, we focus on properties of individual operators.) Hueske et al. [10] statically analyze Java code for selectivity, read-sets, and write-sets. Schneider et al. [20] statically analyze SPL code to discover selectivity, statefulness, and partition isolation, for use in auto-parallelization.

An attractive alternative to static and dynamic analysis is to establish dataflow operator properties by construction in the language design. In StreamIt, selectivity and statefulness are immediately obvious from the code, and the compiler exploits that for scheduling, buffering, and auto-parallelization [24]. DryadLINQ can use annotations to establish a property they call AssociativeDecomposable, used for the partial aggregation hoisting optimization [26]. In general, however, dataflow applications often call out to traditional languages like C++ or Java for some of their functionality, at which point the design of the dataflow language alone cannot establish semantic properties.

III. Properties

For this work, we selected six operator properties that are particularly important for reliability and optimization of dataflow programs. We left some less important properties (such as idempotence, associativity, and attribute forwarding) to future work. This section introduces the six properties. We consider only properties for which our approach can potentially find certain evidence via testing, so we define each property with existential quantifiers ($\exists$).

A. Notation and Wellformedness

Before introducing the properties, we define the notation used in their formal definitions. Square brackets $[\cdots]$ denote lists, curly braces $\{\cdots\}$ denote sets, and angle brackets $\langle\cdots\rangle$ denote tuples. The underscore $\_$ is a don't-care binding (a wild-card). The remaining notation comes from first-order predicate logic. The definitions use the following domains:

- Data domain $D = \mathit{DataItems} \cup \{\bullet, \uparrow\}$. Each $d \in D$ is either a data item, a punctuation (denoted $\bullet$), or a divergence (denoted $\uparrow$, meaning that no more input can be sent to a given port of an operator due to blocking).
- Input domain $I = D \times \mathrm{N}$. Each input $i \in I$ to an operator firing is a pair consisting of a stream element and an input port number. For example, $\langle 1, 0\rangle$ indicates stream element 1 on input port 0.
- Output domain $O = \mathit{list}(D \times \mathrm{N})$. Each output $o \in O$ from an operator firing is a list of $\langle$stream element, output port number$\rangle$ pairs. For example, $[\langle 2, 0\rangle, \langle 2, 1\rangle]$ indicates stream element 2 on output port 0 and then output port 1.
- Execution trace domain $\mathit{trc}(op) = \mathit{list}(I \times O)$. Each trace $t \in \mathit{trc}(op)$ for an operator op is a list of $\langle$input, output$\rangle$ pairs. For example, trace $\big[\langle\langle 1, 0\rangle, [\,]\rangle, \langle\langle 1, 0\rangle, [\,]\rangle\big]$ has two operator firings, each consisting of an input stream element 1 on port 0 and an empty output $[\,]$.

The formal definition of a property P is specified as a predicate $P(op) \triangleq \cdots$ of an operator instance op. Variables must be bound as formal parameters, or via quantifiers ($\exists$ or $\forall$), or by appearing on the left side of a pattern match whose right side is fully bound (for example, if t and j are bound, then $\langle i, \_\rangle = t[j]$ binds i).

B. Non-determinism

A deterministic operator always generates the same output sequences for a given input sequence. If an operator generates two different output sequences for the same inputs, it is non-deterministic. For example, an Aggregate operator instance with a window based on physical machine time may generate different outputs if it receives the same inputs with different inter-arrival times.

Definition.
$\mathit{isNonDeterministicOp}(op) \triangleq$
$\quad \exists t, t' \in \mathit{trc}(op) : \mathit{sameInput}(t, t') \wedge \mathit{differentOutput}(t, t')$
$\mathit{sameInput}(t, t') \triangleq$
$\quad \forall j \in \mathit{dom}(t) : \big(\langle i, \_\rangle = t[j] \wedge \langle i', \_\rangle = t'[j]\big) \Rightarrow \big(i = i'\big)$
$\mathit{differentOutput}(t, t') \triangleq$
$\quad \exists j \in \mathit{dom}(t) : \langle \_, o\rangle = t[j] \wedge \langle \_, o'\rangle = t'[j] \wedge o \neq o'$

For example, traces $[\langle\langle 1, 0\rangle, [\langle 1, 0\rangle]\rangle]$ and $[\langle\langle 1, 0\rangle, [\langle 2, 0\rangle]\rangle]$ for an operator have different outputs for the same input stream; these provide evidence that the operator is non-deterministic.

Motivation. Whether or not determinism is required depends on the application. Knowing about non-determinism is important, especially when users require an application's outputs to be repeatable. If any operator in an application is non-deterministic, the entire application may be non-deterministic. If the outputs from the application differ for the same input data, simply "diffing" outputs is insufficient for correctness checking. Thus, engineers using our approach can detect potential faults in operators and debug them more effectively.

C. Selectivity

Selectivity constrains the number of data items produced by an operator per data items consumed. If an operator produces > 1 data items for at least one firing, it is *prolific*; for example, this is the case for a Split operator configured to duplicate its input data items on multiple output ports. If an operator produces ≤ 1 data items for all firings and none for at least one firing it is *selective*; for example, this is the case for a Filter operator configured to output data items meeting some criterion. If an operator produces exactly 1 data item for each firing it is *one-to-one*; for example, this is the case for a Custom operator configured to count the number of input data items received and output the current count each time it fires.

Definition.
$\mathit{isProlificOp}(op) \triangleq \exists t \in \mathit{trc}(op) : \mathit{isProlific}(t)$
$\mathit{isProlific}(t) \triangleq \exists j \in \mathit{dom}(t) : \langle \_, o\rangle = t[j] \wedge \mathit{multipleTuples}(o)$
$\mathit{multipleTuples}(o) \triangleq$
$\quad \exists k, k' \in \mathit{dom}(o) : k \neq k' \wedge \langle d, \_\rangle = o[k] \wedge \langle d', \_\rangle = o[k']$
$\qquad \wedge\, d \in \mathit{Tuples} \wedge d' \in \mathit{Tuples}$

$$
\begin{aligned}
&\mathit{isSelectiveOp}(op) \triangleq \\
&\quad \neg\mathit{isProlificOp}(op) \wedge \big(\exists t \in \mathit{trc}(op) : \mathit{isSelective}(t)\big) \\
&\mathit{isSelective}(t) \triangleq \exists j \in \mathit{dom}(t) : \langle \_, o\rangle = t[j] \wedge \mathit{noTuples}(o) \\
&\mathit{noTuples}(o) \triangleq \forall k \in \mathit{dom}(o) : \langle d, \_\rangle = o[k] \wedge d \notin \mathit{Tuples} \\
&\mathit{isOneToOneOp}(op) \triangleq \neg\mathit{isProlific}(op) \wedge \neg\mathit{isSelective}(op)
\end{aligned}
$$

For example, since trace $[\langle\langle 1,0\rangle, [\langle 1,0\rangle, \langle 1,1\rangle]\rangle]$ has one input $\langle 1,0\rangle$ with two outputs $\langle 1,0\rangle$, $\langle 1,1\rangle$, it is evidence for an operator being definitely prolific. On the other hand, trace $[\langle\langle 1,0\rangle, [\langle 1,0\rangle]\rangle, \langle\langle 2,0\rangle, [\,]\rangle]$ has no output for input $\langle 2,0\rangle$, which is evidence for being potentially selective.

Motivation. Selectivity can be used for scheduling and allocating buffers to improve performance [14], [24]. It can also help in parallelization, since it can simplify sequence numbers for ordering [15], [20]. Providing correct selectivity specifications thus enables safe optimization. Operator developers can use our technique to increase their confidence that they have specified selectivity correctly.

D. Blocking

Unlike the previous two properties, blocking is specific to an input port. Input port p is blocking if a firing can get stuck partway, so no further firings are possible at p until it is unblocked. For example, the Gate operator in SPL blocks one input port until it receives an acknowledgment from another input port.

Definition.

$$
\begin{aligned}
&\mathit{isBlockingOp}(op, p) \triangleq \exists t \in \mathit{trc}(op) : \mathit{isBlocking}(t, p) \\
&\mathit{isBlocking}(t, p) \triangleq \\
&\quad \exists j \in \mathit{dom}(t) : \\
&\qquad \langle\langle \_, p\rangle, o\rangle = t[j] \wedge \big(\exists k \in \mathit{dom}(o) : \langle \uparrow, \_\rangle = o[k]\big)
\end{aligned}
$$

For example, trace $[\langle\langle 1,0\rangle, [\,]\rangle, \langle\langle 1,0\rangle, [\langle \uparrow, 0\rangle]\rangle]$ provides evidence that the operator is blocking on input port $p = 0$.

Motivation. Blocking can help ensure that stream elements are placed in proper order. A stream graph with a confluence, but without a blocking or stateful operator, may behave unpredictably. However, blocking is also the main source of deadlock in dataflow applications [15]. A developer trying to locate the root cause of a deadlock can use knowledge about blocking operators as a starting point.

E. Statefulness

An operator is stateful if its current output is affected by input stream elements earlier in the same trace. An example in SPL is a Custom operator configured to count input stream elements and output the current count each time it fires.

Definition.

$$
\begin{aligned}
&\mathit{isStatefulOp}(op) \triangleq \exists t \in \mathit{trc}(op) : \mathit{isStateful}(t) \\
&\mathit{isStateful}(t) \triangleq \\
&\quad \exists j, j' \in \mathit{dom}(t) : \\
&\qquad j \neq j' \wedge \langle i, o\rangle = t[j] \wedge \langle i, o'\rangle = t[j'] \wedge o \neq o'
\end{aligned}
$$

For example, trace $[\langle\langle 1,0\rangle, [\langle 1,0\rangle]\rangle, \langle\langle 1,0\rangle, [\langle 2,0\rangle]\rangle]$ shows that for the same input stream element, there are two different output stream elements; this provides evidence that the operator is stateful.

Motivation. A stateless operator is easy to parallelize for better performance, and easy to restart or migrate for better fault-tolerance. On the other hand, a stateful operator can buffer out-of-order stream elements in a confluent graph. An application developer can use knowledge of which operators are stateful to help establish whether the entire program has predictable behavior. Furthermore, many applications inherently need stateful operators, for instance, for aggregation.

F. Non-commutativity

Non-commutativity, like blocking, is specific to an input port. An input port p is commutative if a change in the order of input data items sent to p within some range (such as within a window) does not change the outputs. For example, an Aggregate operator configured to find the maximum of five input data items is commutative, since the order of the five data items does not affect the maximum. In contrast, an Aggregate operator configured to find the last of five input data items is non-commutative.

Definition.

$$
\begin{aligned}
&\mathit{isNonCommutativeOp}(op, p) \triangleq \\
&\quad \exists t \in \mathit{trc}(op) : \mathit{isNonCommutative}(t, p) \\
&\mathit{isNonCommutative}(t, p) \triangleq \\
&\quad \exists j, k, n \in \mathit{dom}(t) : j < k \\
&\qquad \wedge \mathit{endsWindow}(t, j-1) \wedge \mathit{endsWindow}(t, j+n-1) \\
&\qquad \wedge \mathit{endsWindow}(t, k-1) \wedge \mathit{endsWindow}(t, k+n-1) \\
&\qquad \wedge \mathit{onlyUsesPort}(t, j, k-j+n, p) \\
&\qquad \wedge \mathit{sameSetOfInputs}(t, j, k, n) \wedge \mathit{differentOutput}(t, j, k, n) \\
&\mathit{endsWindow}(t, j) \triangleq \\
&\quad j = -1 \vee \langle \_, o\rangle = t[j] \wedge \big(\exists k \in \mathit{dom}(o) : \langle \bullet, \_\rangle = o[k]\big) \\
&\mathit{onlyUsesPort}(t, l, n, p) \triangleq \\
&\quad \forall j \in \mathit{dom}(t) : \\
&\qquad \big(l \leq j \wedge j < l+n \wedge \langle\langle \_, p'\rangle, \_\rangle = t[j]\big) \Rightarrow \big(p = p'\big) \\
&\mathit{sameSetOfInputs}(t, j, k, n) \triangleq \\
&\quad \{i : l \in \mathit{dom}(t) \wedge j \leq l \wedge l < j+n \wedge \langle i, \_\rangle = t[l]\} = \\
&\quad \{i : l \in \mathit{dom}(t) \wedge k \leq l \wedge l < k+n \wedge \langle i, \_\rangle = t[l]\} \\
&\mathit{differentOutput}(t, j, k, n) \triangleq \\
&\quad \langle \_, o\rangle = t[j+n-1] \wedge \langle \_, o'\rangle = t[k+n-1] \wedge o \neq o'
\end{aligned}
$$

For example, $[\langle\langle 1,0\rangle, [\,]\rangle, \langle\langle 2,0\rangle, [\langle 2,0\rangle, \langle \bullet, 0\rangle]\rangle]$ followed by $[\langle\langle 2,0\rangle, [\,]\rangle, \langle\langle 1,0\rangle, [\langle 1,0\rangle, \langle \bullet, 0\rangle]\rangle]$ shows that after changing the order of two input data items, the output data items are different before the window punctuation; this trace provides evidence that the operator is non-commutative.

Motivation. When two paths in a dataflow graph converge on a single operator, differences in the speed and scheduling of upstream operators can cause data items to be out of order. In general, this can lead to application-level non-determinism even if all individual operators are deterministic. An application developer can use commutativity to find out whether an operator tolerates disorder.

G. Partition-interference

A partitioning *key* is a part of each input data item used for processing subsets of the stream separately. Keys are often simply record attributes, but to keep the definition general, we write $\mathit{read}(d, k)$ for reading key k from data item d. In SPL, such keys are specified by configuring an operator with

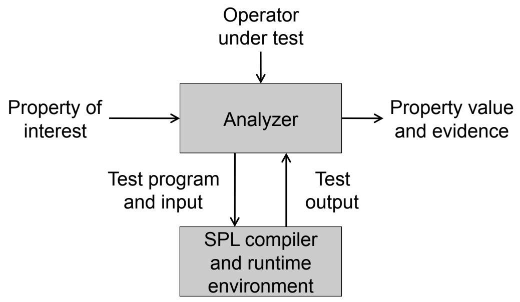

Fig. 2. Testing framework

a partitionBy parameter. In a partition-isolated operator, output data items for one key are not affected by input data items with different keys. If data items with different key values can affect each other, the operator is partition-interfering. For example, an Aggregate operator configured to compute the average price of a stream of stock trades separately for each stock symbol is partition-isolated. On the other hand, an Aggregate configured to add the prices of a stream of deals irrespective of stock symbol is partition-interfering.

Definition.
$\mathit{isPartitionInterferingOp}(op, k) \triangleq$
$\quad \exists t, t' \in \mathit{trc}(op) : \exists c \in D :$
$\quad\quad \mathit{sameInputs}(t, t', k, c) \wedge \mathit{differentOutputs}(t, t', k, c)$
$\mathit{sameInputs}(t, t', k, c) \triangleq$
$\quad [i : \langle i, \_\rangle \in t \wedge \langle d, \_\rangle = i \wedge \mathit{read}(d, k) = c] =$
$\quad [i : \langle i, \_\rangle \in t' \wedge \langle d, \_\rangle = i \wedge \mathit{read}(d, k) = c]$
$\mathit{differentOutputs}(t, t', k, c) \triangleq$
$\quad [o : \langle i, o\rangle \in t \wedge \langle d, \_\rangle = i \wedge \mathit{read}(d, k) = c] \neq$
$\quad [o : \langle i, o\rangle \in t' \wedge \langle d, \_\rangle = i \wedge \mathit{read}(d, k) = c]$

For example, in the traces $[\langle\langle\langle 1, c\rangle, 0\rangle, [\,]\rangle, \langle\langle 2, c\rangle, 0\rangle, [\langle 2, 0\rangle]\rangle]$ and $[\langle\langle\langle 1, c\rangle, 0\rangle, [\,]\rangle, \langle\langle\langle 1, c'\rangle, 0\rangle, []\rangle, \langle\langle 2, c\rangle, 0\rangle, [\langle 3, 0\rangle]\rangle]$, an input data item whose key c' differs from the key c of the other two input data items affected the output. This pair of traces provides evidence that the operator is partition-interfering.

Motivation. Partition-isolation is useful for parallelization [5], [20]. An optimizer can parallelize partition-isolated operators by using a hash-split, thus giving each operator replica a disjoint partition of the key domain. Operator developers can use our approach to test whether they mistakenly created operators that are partition-interfering.

IV. Testing Methodology

The desired value of an operator property in the context of a dataflow program depends on the intent of the program's developer. For example, a developer could use a stateful operator on purpose to compute an average, or use a stateless operator on purpose to simplify parallelization. In either case, a mismatch between the developer's intent and the actual fact is a defect. This section introduces a testing framework for determining operator properties.

A. Testing Framework

Figure 2 shows our testing framework. The main component is the Analyzer, which takes as input the property that users want to test and an operator-under-test (OUT). The Analyzer generates a test program wrapping the OUT and sends it to the SPL compiler to generate an executable program. The Analyzer then generates inputs appropriate for the property and executes the program using the SPL runtime environment. Execution results are returned to the Analyzer, which checks whether evidence is found to show that the property holds. If evidence is found, the Analyzer presents users with that evidence. If evidence is not found, the Analyzer generates additional tests until it reaches a technique time limit, and on reaching that limit, reports that the property "potentially" does not hold. The Analyzer can also report statistical information that can help users assess the extent of the evidence provided.

For example, suppose that a developer creates an operator *op* that is required, due to expected compiler optimizations (e.g., as per [24]), to be stateless. Suppose the Analyzer generates test cases that serve as evidence that *op* is stateful. Presented with this evidence, the developer can correct their code or take other necessary steps to ensure that incorrect optimization does not occur. If, on the other hand, the Analyzer finds no test cases that indicate that *op* is stateful, the developer can have some confidence (even if not certainty) that *op* is stateless.

We now describe how the Analyzer wraps an OUT to form a test program that produces output as part of test oracles. Figure 3 shows the dataflow graph of a test program that contains four operator instances: Source, Controller, the OUT, and Sink. Source reads in a test from a file. Controller sends input stream elements from Source to the appropriate input port of the OUT. (Most OUTs have exactly one input port and one output port, but an OUT can have multiple input and output ports.) The OUT receives the stream(s) from Controller and executes to generate output stream(s). Sink reads the stream(s) from OUT and prints them out. The printed information from Controller and Sink is used by the Analyzer to make decisions.

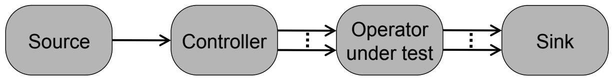

Fig. 3. Dataflow graph of test program

B. Testing Order

The order in which operator properties are tested is important: the status of certain properties can preclude the need to check others. Thus, we have analyzed the relationships between properties in order to identify a testing order that maximizes testing efficiency. Our testing order is the partial order shown in Figure 4. (The numbers shown in the figure are discussed later.) As the figure shows, the first and most important property to test for is non-determinism. If an OUT is non-deterministic, it is less important to test for other properties in practice, because they are harder to exploit. From the definition of selectivity, we can see that it focuses on the OUT's behavior after each single input stream element, while other properties focus on two or more stream elements, so selectivity is independent of other properties. Thus, we can test for selectivity before, after, or in parallel with other properties.

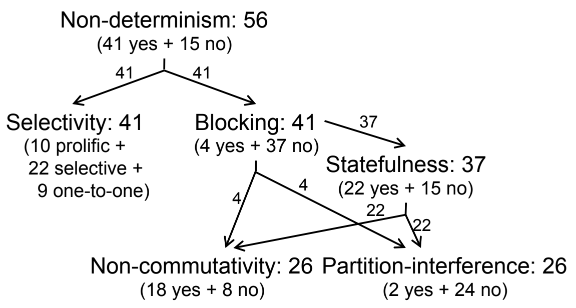

Fig. 4. Testing order

Next, we check whether the OUT has two or more input ports. If it does, we check blocking first, because blocking situations in SPL occur only when one input port is stalled while the operator instance waits for data on another input port. If the OUT is blocking, it is stateful, so we can skip the check for statefulness and directly check for non-commutativity and partition-interference. If the OUT is non-blocking, or if it has only one input port, we check whether it is stateful first. If it is not stateful, it is commutative and partition-isolated, and requires no further checking. If it is stateful, we check non-commutativity and partition-interference.

C. Testing Each Property

The formal definitions of the properties given in Section III provide insights into ways to design testing strategies. Algorithm 1 describes our general testing strategy, encapsulating property-specific details in calls to four functions (*initialize*, *generateTest*, *checkProperty*, and *evidenceValue*). Table I shows the functions for each of the six properties in detail. Below we provide additional comments on the specific design choices used to perform property testing for SPL.

Algorithm 1 Algorithm for checking properties

```
evidence = null
initialize(property)
while evidence==null ∧¬(techniqueTimeLimit reached) do
    test = generateTest(property)
    evidence = checkProperty(property, test)
end while
return evidenceValue(property, evidence)
```

1) Non-determinism: The main cause of operator non-determinism in SPL is the inter-arrival time of stream elements, so we need to include this factor in tests. An input for non-determinism thus includes stream elements and port numbers, as well as delays between those stream elements.

2) Selectivity: The functions for testing selectivity in Table I follow directly from the formal definition.

3) Blocking: In SPL, blocking occurs in operators with more than one input port. Such operators can block one port for synchronization until an expected stream element arrives at another port. Therefore, our tests contain stream elements for only the input port under test. For blocking, we compile the test program into a single process, not distributed, so that stream communication is a simple function call. We check whether the submit function in the upstream operator instance (the Controller in Figure 3) returns within *blockingTimeLimit*.

4) Statefulness: The functions for testing selectivity in Table I follow directly from the formal definition, but for simplicity, we input the same data item to each firing.

5) Non-commutativity: Commutativity is specific to a particular input port p. We begin by checking whether submitting data items to this input port eventually causes punctuations to appear on an output port. Let w be the number of input data items seen before an output punctuation appears. We generate permutations of w data items and check whether any two permutations generate different outputs.

6) Partition-interference: In SPL, programmers declare partitioning by configuring operators with a partitionBy parameter, so we need only check those operators for the specified key. We check two traces, one in which all input data items have the same key, and the other for the same input data items, but interspersed with data items whose keys are different.

D. Test Generation

For this work, we created test generation techniques that are particularly appropriate given the characteristics of dataflow operators and our testing methodology, along with techniques that facilitate the empirical comparison of techniques.

To create test generation techniques we must first generate tests, and there are many approaches that could be utilized to do this. In this work, we begin with a *random* test generation [6] approach, in which, for each input to an operator, values are randomly chosen within a range appropriate for its type. We chose this approach because it is relatively simple to implement and provides a baseline for comparison.

Random test generation involves a huge search space, and thus, we sought test generation approaches that reduce that search space. The second approach that we consider leverages the fact that in SPL code, there are many comparisons between attributes and constants. For example, some operators will submit data items if one attribute of an input data item equals a number. If tests cover both sides of those comparisons, true and false, properties may be revealed. To create such tests we created a scanner to extract constants such as strings, integers, and floating-point numbers from the code, and place this information in pools. We then randomly select values from these pools. We call this approach *pool-based* generation.

The third approach that we consider leverages the fact that in SPL code, there are many operators that use combinations of predicates, e.g., `x.y==5 && x.name=="Smith"`. In this case, the selection space for the pool selection approach can still be inordinately large, so we apply an approach widely used in combinatorial testing to further reduce the space: *pairwise* testing [4]. Pairwise-generated tests cover all combinations of two. This makes their numbers much smaller than the number of possible random or pool-generated tests. We call this approach *pair-based* generation.

To employ the foregoing approaches in the context of SPL, we considered two orthogonal features of stream processing

TABLE I
FUNCTION IMPLEMENTATIONS FOR PROPERTIES

Property	*initialize*	*generateTest*	*checkProperty*	*evidenceValue*
Non-determinism		Return two lists l_1 and l_2 of triples $\langle delay, d, p\rangle$ with the same values for d and p, but different *delay* values.	Run $l_{1,2}$ with d, p values as inputs, timed by the *delay* values. If the outputs differ, return $\langle l_1, l_2\rangle$, else return **null**.	If *evidence* $\neq$ **null**, return "definitely non-deterministic", else return "potentially deterministic".
Selectivity	Set *selectiveEvidence* = **null**.	Return list l of pairs $\langle d, p\rangle$.	Run l. If any firing has > 1 output data items, return l. If any firing has 0 output data items, set *selectiveEvidence*=l. Return **null**.	If *evidence* $\neq$ **null**, return "definitely prolific", else if *selectiveEvidence* $\neq$ **null**, return "potentially selective", else return "potentially one-to-one".
Blocking		Return list l of pairs $\langle d, p\rangle$, where all p values are the port-under-test.	Run l. If any firing stalls for *blockingTimeLimit*, return l, else return **null**.	If *evidence* $\neq$ **null**, return "potentially blocking", else return "potentially non-blocking".
Statefulness		Return list l of pairs $\langle d, p\rangle$, where all d values are the same for simplicity.	Run l. If any two firings produce different outputs, return l, else return **null**.	If *evidence* $\neq$ **null**, return "definitely stateful", else return "potentially stateless".
Non-commutativity	Generate and run tests l of increasing length until a window punctuation is generated. Let w be the length of such a list.	Generate list l of w pairs $\langle d, p\rangle$; return permutations of l.	Run each permutation. If any two permutations produce different outputs, return them, else return **null**.	If *evidence* $\neq$ **null**, return "definitely non-commutative", else return "potentially commutative".
Partition-interference		Generate list l_1 of pairs $\langle d, p\rangle$ with the same $read(d, k)$ for all d. Create l_2 containing elements from l_1 interspersed with data items with different keys.	Run l_1 and l_2. If any of the outputs for corresponding data items differ, return $\langle l_1, l_2\rangle$, else return **null**.	If *evidence* $\neq$ **null**, return "definitely partition-interfering", else return "potentially partition-isolated".

languages: dataflow operators and our testing methodology. First, we often need to check output data items from the OUT; thus, we require tests for which the output is non-empty before we can check for properties. For example, when checking for commutativity, we first need a test that generates output data items, and then we can try permutations of the input data items to see whether the output changes. This suggests that the application of *mutation* [3] may be helful in our context. Specifically, we apply mutation to existing tests that do not generate output data items. In our approach, we add additional input data items by mutating a single character of string attributes, adding one and minus one for numeric attributes, and acting similarly for enumeration constants and timestamps. In this way, we expect to be able to trigger comparisons that generate output data items.

The second additional feature we consider involves *test reuse*. When we follow the order of property testing displayed in Figure 4, we generate tests while checking properties earlier in the order. Since these tests are ready to use we may save time by reusing them, thus improving testing efficiency. Note that when using this approach and when checking properties earlier in the order we cannot save every generated test; thus, we save only those that actually generate output data items.

The three approaches for generating test inputs that we have described can be combined with these two orthogonal factors to create 12 different test generation techniques. These techniques combine the three possible test input generation approaches (random, pool-based, and pair-based) with mutation usage (with and without), and reuse of tests (with and without). We believe that these techniques may have different strengths across different properties and operators, and thus, in our empirical studies, we investigate all of them.

Finally, when resources permit, test engineers may simultaneously apply multiple testing techniques, and in doing so take advantage of the relative strengths and weaknesses of individual techniques. To investigate this approach, in addition to studying the 12 techniques individually, we created a hybrid technique that applies all 12 techniques simultaneously in parallel for a given OUT. As soon as any technique finds evidence to show that a tested property definitely holds, it stops and returns the answer and evidence. If no single technique finds evidence, all 12 techniques reach the technique time limit, and the property potentially does not hold.

V. Empirical Study

To evaluate our approach to verifying properties there are several dimensions that we need to assess. The first dimension involves how effective the approach is at determining whether properties hold or not. Here, we are interested in the *precision* of the approach (how often we report no evidence when indeed no evidence exists, on the "potentially" side of a property) and the *recall* of the approach (how often it finds evidence when such evidence does in fact exist, on the "definitely" side of a property). The second dimension of interest involves the *efficiency* of the approach; that is, how quickly it returns results. The third dimension of interest involves the tradeoffs between the different test generation techniques that we have proposed. To investigate these dimensions of our approach we pose the following research questions:

RQ1: How do the test generation techniques fare, in terms of precision and recall, in testing the six properties considered.

RQ2: How do the test generation techniques fare, in terms of efficiency, in testing the six properties considered.

Note that by addressing these questions, we also obtain data that allows us to discuss differences that occur in assessing the six different properties we consider.

A. *Objects of Analysis*

We selected 56 operator instances from the standard toolkit of IBM's InfoSphere Streams [11], and from examples found in online tutorials for that product. We selected these operators in part because InfoSphere Streams is a widely-used commercial product, most of the operators are in commercial use, and the properties we selected for study are the most helpful for optimizations and correctness arguments in that programming environment. Furthermore, the operator instances we selected cover all operator definitions in the standard toolkit that have at least one input and one output port, including 6 relational and 11 utility operator definitions. They also include 4 primitive operators written in Java and 6 primitive operators written in C++. The lines-of-code per operator instance are between 54 and 252, and the average is 145.7.

When applying our approach we utilized the testing order described in Section IV-B; Figure 4 indicates the numbers of operator instances that needed to be considered at each step. If we did not use the testing order, we would have needed to check all 56 instances for each property (336 checks altogether); the use of the ordering reduced the number of checks to 227.

B. *Variables and Measures*

Independent Variable. Our experiment manipulated one independent variable: testing technique. We applied each of the 13 test generation techniques on all properties except non-determinism. For non-determinism, given the order in which we considered properties, there were no existing tests for reuse, so only seven techniques were applicable.

Ideally, we would like to compare our techniques to some baseline. Unfortunately, there are currently no automated techniques that could serve as such baselines. A second way to compare techniques is to use a theoretical optimal approach to judge whether properties hold. Such an approach yields correct answers, allowing the precision and recall of techniques to be assessed relative to those correct answers. This approach can be approximated by using human judgment and code inspection to determine whether properties hold; however, this process is also expensive (a primary motivation for developing our automated methodology). Thus, to determine whether properties held, we began by running our techniques on all properties. If we found evidence for a property, we were certain about that case. We then investigated remaining cases in which the techniques found no evidence, one by one. In these cases, the first author inspected each of the operators relative to each remaining property, and determined whether the property held. The second author then did the same, and then the two authors met and came to a consensus opinion. In this latter step, the authors found a single case in which their judgments disagreed. (In our experiments, our techniques also ultimately revealed the correct answer for this case).

Dependent Variables. We measured the precision, recall, and efficiency of each technique as follows. Let P be a property, let Ops_P be the set of operator instances on which P is tested, let t be a technique, and let h be the human expert baseline.

DV1: Precision. Precision measures the extent to which a technique does not find false evidence when indeed no evidence exists. We calculated the precision of t as:

$$\frac{\left|\left\{op \in \mathit{Ops}_P : \neg \mathit{evidence}(op, P, t) \wedge \neg \mathit{evidence}(op, P, h)\right\}\right|}{\left|\left\{op \in \mathit{Ops}_P : \neg \mathit{evidence}(op, P, h)\right\}\right|}$$

DV2: Recall. Recall measures the extent to which a technique finds evidence when indeed such evidence exists. We calculated the recall for t as:

$$\frac{\left|\left\{op \in \mathit{Ops}_P : \mathit{evidence}(op, P, t) \wedge \mathit{evidence}(op, P, h)\right\}\right|}{\left|\left\{op \in \mathit{Ops}_P : \mathit{evidence}(op, P, h)\right\}\right|}$$

DV3: Efficiency. For any application of a technique to a property we hope to obtain evidence as to whether that property holds as quickly as possible. Thus, to measure the efficiency of a technique t, we measured the time used by t to find evidence for each property P in the set of positive operator instances, $\{op \in \mathit{Ops}_P : \mathit{evidence}(op, P, h)\}$.

C. *Experiment Setup and Operation*

In theory, we can let techniques run forever, since evidence that a property holds can be found at any time. In practice, of course, time limits are required. To determine what technique runtime limits might be reasonable we conducted preliminary trials using our techniques. We began with a technique runtime limit of one minute and increased it by multiplying it by three. When we failed to gain more than a 1% improvement on effectiveness for a technique runtime limit on all operators and properties, we stopped and chose the previous limit. Since we reached the point of diminishing returns at nine minutes, we chose three minutes as the *techniqueTimeLimit* in Algorithm 1.

We chose two seconds as the *blockingTimeLimit* in Table I, because preliminary trials showed that a one second threshold did not work well whereas two seconds allowed us to draw correct conclusions.

To check each property, we began with a small number of input data items, since it is easier for people to understand shorter evidence and tests. We chose to start with two input stream elements in l, since for checking statefulness, we need at least two input stream elements.

Finally, because each technique that we considered can act differently in individual runs due to non-determinism in the test generation approach, we applied each technique 10 times to each operator.

Our experiments were run on a RedHat Linux box with an Intel Core2duo at 2.4GHz. We used InfoSphere Streams 2.0.0.1.

D. *Threats to Validity*

Where external validity is concerned, the operators we studied were all drawn from one pool of operators associated with the InfoSphere Streams product, and different operators might yield different results. Further, we studied only six properties. However, the operators we used are in wide commercial use, and thus, represent operators of particular interest.

TABLE II
RECALL

Property	Without Reuse						With Reuse						Hybrid
	Without Mutation			With Mutation			Without Mutation			With Mutation			
	Rand	Pool	Pair	Rand	Pool	Pair	Rand	Pool	Pair	Rand	Pool	Pair	
Non-determinism	48.7	53.3	52.0	24.0	40.7	36.7	-	-	-	-	-	-	***74.7***
Selectivity	75.3	93.4	91.6	91.9	***100.0***	***100.0***	84.7	93.8	93.1	93.8	***100.0***	***100.0***	***100.0***
Blocking	***83.3***	***83.3***	***83.3***	***83.3***	***83.3***	***83.3***	***83.3***	***83.3***	***83.3***	***83.3***	***83.3***	***83.3***	***83.3***
Statefulness	68.2	72.7	72.7	72.7	81.8	81.8	77.3	81.8	81.8	81.8	***86.4***	***86.4***	***86.4***
Non-commutativity	35.0	71.3	73.8	35.0	72.5	77.5	32.5	71.3	75.0	38.8	70.0	72.5	***90.0***
Partition-interference	80.0	93.3	88.8	79.6	90.8	91.7	83.8	90.4	88.8	83.8	88.8	89.6	***94.6***

TABLE III
EFFICIENCY

Property	Without Reuse						With Reuse						Hybrid
	Without Mutation			With Mutation			Without Mutation			With Mutation			
	Rand	Pool	Pair	Rand	Pool	Pair	Rand	Pool	Pair	Rand	Pool	Pair	
Non-determinism	114.9	109.8	118.7	146.0	127.8	136.3	-	-	-	-	-	-	***78.8***
Selectivity	149.2	134.9	134.8	136.5	126.2	126.8	129.8	126.3	126.7	130.0	125.1	125.4	***124.3***
Blocking	32.4	32.3	32.4	32.1	32.1	32.1	32.0	***31.9***	***31.9***	32.0	32.0	32.0	***31.9***
Statefulness	62.2	53.4	47.5	55.5	41.6	36.2	45.2	37.4	32.3	48.8	40.4	37.7	***25.8***
Non-commutativity	126.0	63.6	55.3	125.8	56.6	48.4	127.7	61.5	56.1	120.8	59.2	58.3	***23.5***
Partition-interference	58.6	39.8	43.4	53.0	38.0	38.0	55.3	47.8	51.4	56.7	53.0	56.5	***18.3***

Our greatest concern for internal validity involves our implementations of properties and scripts used to gather data. To reduce these threats we carefully evaluated our properties and scripts on a wide range of examples. Furthermore, while our testing order obviates the need to run all techniques on all properties, as an additional check on our implementations, we did also apply all techniques in those cases in which they were not required, and verified that they returned correct results in these cases as well. Another potential threat involves the time limits chosen for use in testing properties. As detailed above, however, we based our time limit choices on preliminary trials.

Where construct validity is concerned, as a baseline for comparing our results we used property determinations performed by humans, which could have been incorrect. Also, additional measures, such as the usefulness of the property determinations, could be valuable.

E. Results

1) RQ1: Technique effectiveness: Precision. As expected, in our study, no cases occurred in which a technique found false evidence. Therefore, all techniques achieved 100% precision.

Recall. Table II presents recall data. The first column indicates the property name. Columns 2-13 are divided into two sections: the first six present data on techniques without test reuse and the next six present data on techniques with test reuse. Within each of these sections, the first three columns present data on techniques without mutation and the last three present data on techniques with mutation. Within each set of three columns, the columns represent random (Rand), pool-based (Pool), and pair-based (Pair) test generation techniques, respectively. The rightmost column shows the data on the hybrid technique. Each cell shows the average recall rate percentage over all ten runs. Bold font indicates which technique(s) were most effective for a given property.

From the data, we can see that the most effective technique was (or included) the hybrid technique; it was the most effective on all six properties and it achieved a recall rate ranging from 74.7% to 100.0%. For selectivity, using pool and pairwise techniques with mutation achieved the same effectiveness as the hybrid technique, 100%. For blocking, all techniques achieved the same recall rate, 83.3%.

For all properties, pair-based or pool-based test generation approaches were at least as effective as (and on five properties more effective than) random test generation. Also, as noted, techniques that use mutation were at least as effective as those that do not on three of six properties. Finally, techniques that use test reuse were at least as effective as those that do not on three of six properties.

2) RQ2: Technique efficiency: Table III presents efficiency data. The table has the same structure as Table II, but here each cell represents the average time (in seconds) required by each technique to find evidence. To be consistent with Table II, we report data only for operators on which there exists evidence. As the data shows, average times for each technique were all relatively small, ranging from 18.3 to 149.2 seconds. The hybrid technique was the most efficient for all properties. Finally, random test generation required at least the same amount of time as the other two test generation approaches given the same mutation and reuse approaches.

VI. DISCUSSION AND IMPLICATIONS

We now provide additional insights into the results of our study, and comment on implications.

Failed cases: Our techniques failed to determine correct answers for properties on only seven operators; in other cases, at least one technique in one run returned a correct answer.

When checking for non-determinism, there were two cases in which none of our techniques found evidence that the property held. One is a DeDuplicate operator using 120 seconds as its timeout. If we control the number of stream elements arriving at the operator in 120 seconds we can find evidence to show that it is non-deterministic. To do this, we could use an inter-element delay period of 120 seconds. In our implementation, however, we limited inter-element delay periods to five seconds, and we limited the number of delay periods on an input stream to three, so that we could try more tests in a given time. In general, any operator having a window or timeout based on a large physical time interval will present such challenges. To address this problem, we could ask users to provide delay period times.

The second case in which checking for non-determinism failed involved a Join operator that generates output only when a concatenation of three strings equals a fourth string. In general, this is a difficult constraint to satisfy. Asking users for sample input data could help in this case.

When checking for blocking our techniques missed one case: a Gate operator with buffer size 1,000. As implemented, none of our techniques generates test inputs with 1,000 stream elements in three minutes. To find evidence for this operator, the techniques need to run longer, or begin with a larger number of input stream elements.

When checking for statefulness our techniques missed three cases: two Aggregate instances and a Sort instance. These instances are configured with a delta window in which the difference for an attribute between the first and the last stream element in the window is specified. Our techniques are not likely to find stream elements satisfying such a condition. Asking users for sample input data could help.

When checking for partition-interference our techniques missed one case; a C++ primitive operator. Except on very specific inputs this operator is typically blocking, in which case its outputs cannot be checked for partition-interference. Generating appropriate inputs in this case is difficult. Again, asking users for sample data could help.

Test generation approaches: Table II shows that random test generation was less effective than the other test generation techniques. Because random generation performs an unguided search, its search space is too large to let it select useful values. Comparing the pair-based and pool-based techniques, there were only six cases in which pair-based was more effective, and only four in which pool-based was more effective. As mentioned in Section IV, pair-based generation benefits when multiple conditions need to be satisfied to obtain an output. On inspection, we found that only two of the operators we tested contained this type of code.

Mutation strategy: For selectivity and statefulness, techniques that used mutation performed better than those that did not. However, when checking for non-determinism, mutation missed operators with time-based windows greater than two seconds. On such operators, with mutation, the time required is too large to allow techniques to generate sufficiently long input streams. Therefore, in these cases none of the techniques generates an output, and they report that the operator is potentially deterministic. For non-commutativity and partition-interference, mutation did not seem to be helpful. Mutation is used to generate a test that is able to produce some output. If existing tests already generate outputs, mutation will waste time and then fail to find evidence in a technique time limit.

Test reuse: For selectivity and statefulness, techniques with reuse were more effective and efficient than those without, because they can use existing tests directly. In particular, when checking for statefulness, the testing strategy described in Section IV uses the same input stream elements. This strategy can miss cases in which we require different input stream elements to reveal the property, while test reuse can help with this. For non-commutativity and partition-interference, however, techniques with reuse displayed no advantage over those without. These properties have more strict requirements for tests, so existing tests may not satisfy them. For blocking we deleted the input stream elements for the input port that was not under test; thus, test reuse did not provide efficiency advantages in this case.

Technique comparisons: Given our data and the foregoing discussion, it is clear that among the 12 non-hybrid techniques, there is no consistent winner. Mutation and reuse were not always helpful. Thus, it is difficult to recommend any single technique as appropriate in all cases. This emphasizes the importance of the hybrid technique. The hybrid technique is able to produce more varied test cases, which help it succeed on operators on which single techniques could not. Essentially, the approach achieves greater "search diversity", in a manner similar to that discussed in [7]. Of course, in some sense, the hybrid technique achieves its effectiveness because it allots additional time to testing than the other techniques through its use of additional resources (e.g., multiple processors). Nonetheless, if a user wishes to obtain a more accurate answer more quickly and has access to such resources, the hybrid technique is the best choice.

VII. Conclusions and Future Work

Dataflow programming is a widely-used paradigm, and with the rise of big data it has gained significant commercial importance. In this work, we have presented an approach for testing dataflow program operators, and provided empirical data that shows that this approach can be effective and efficient at providing evidence that properties of operators do or do not hold. By assisting with the verification of operator properties, our approach can help software engineers create more dependable operators and dataflow programs.

We have focused on testing individual SPL operator instances. In follow-up work, we adapted our approach to MapReduce, and applied it to characterize semantic properties of MapReduce workloads [25]. As future work, we intend to consider the next stages of dataflow program validation, in which testing approaches analogous to integration and system testing are needed. We also plan to expand on the scope of our empirical study, and to examine the potential usefulness of the methodology in the hands of engineers.

Acknowledgements

This work was supported in part by the AFOSR through award FA9550-09-1-0129 to the University of Nebraska - Lincoln. Portions of this work were performed by the first author during the course of an internship at IBM Research. We thank Gabriela Jacques-Silva, Andy Frenkiel, and Scott Schneider for inspiring discussions of dataflow operator properties.

References

[1] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Uğur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The design of the Borealis stream processing engine. In *Conference on Innovative Data Systems Research (CIDR)*, pages 277–289, 2005.

[2] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In *Conference on Programming Languages Design and Implementation (PLDI)*, pages 363–375, 2010.

[3] Kumar Chellapilla. Combining mutation operators in evolutionary programming. *IEEE Transactions on Evolutionary Computation (TEC)*, 2(3):91–96, September 1998.

[4] David M. Cohen, Siddhartha R. Dalal, Jesse Parelius, and Gardner C. Patton. The combinatorial design approach to automatic test generation. *IEEE Software*, 13(5):83–88, September 1996.

[5] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 137–150, 2004.

[6] Joe W. Duran and Simeon Ntafos. A report on random testing. In *International Conference on Software Engineering (ICSE)*, pages 179–183, 1981.

[7] Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. Swarm testing. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 78–88, 2012.

[8] Martin Hirzel. Partition and compose: Parallel complex event processing. In *Conference on Distributed Event-Based Systems (DEBS)*, pages 191–200, 2012.

[9] Martin Hirzel, Henrique Andrade, Buğra Gedik, Gabriela Jacques-Silva, Rohit Khandekar, Vibhore Kumar, Mark Mendell, Howard Nasgaard, Scott Schneider, Robert Soulé, and Kun-Lung Wu. IBM Streams Processing Language: Analyzing big data in motion. *IBM Journal of Research and Development (IBMRD)*, 57(3/4):7:1–7:11, 2013.

[10] Fabian Hueske, Mathias Peters, Matthias J. Sax, Astrid Rheinländer, Rico Bergmann, Aljoscha Krettek, and Kostas Tzoumas. Opening the black boxes in data flow optimization. In *Conference on Very Large Databases (VLDB)*, pages 1256–1267, 2012.

[11] IBM Infosphere Streams. http://www.ibm.com/software/data/infosphere/streams/.

[12] K. R. Jayaram and Patrick Eugster. Program analysis for event-based distributed systems. In *Conference on Distributed Event-Based Systems (DEBS)*, pages 113–124, 2011.

[13] Westley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1):1–34, 2004.

[14] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[15] Peng Li, Kunal Agrawal, Jeremy Buhler, and Roger D. Chamberlain. Deadlock avoidance for streaming computations with filtering. In *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 243–252, 2010.

[16] Mark Mendell, Howard Nasgaard, Eric Bouillet, Martin Hirzel, and Buğra Gedik. Extending a general-purpose streaming system for XML. In *Conference on Extending Database Technology (EDBT)*, pages 534–539, 2012.

[17] Christopher Olston, Shubham Chopra, and Utkarsh Srivastava. Generating example data for dataflow programs. In *International Conference on Management of Data (SIGMOD)*, pages 245–256, 2009.

[18] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. Parallel-stage decoupled software pipelining. In *Symposium on Code Generation and Optimization (CGO)*, pages 114–123, 2008.

[19] Martin C. Rinard and Pedro C. Diniz. Commutativity analysis: A new analysis framework for parallelizing compilers. In *Conference on Programming Languages Design and Implementation (PLDI)*, pages 54–67, 1996.

[20] Scott Schneider, Martin Hirzel, Buğra Gedik, and Kun-Lung Wu. Auto-parallelizing stateful distributed streaming applications. In *Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[21] Robert Soulé, Martin Hirzel, Buğra Gedik, and Robert Grimm. From a calculus to an execution environment for stream processing. In *Conference on Distributed Event-Based Systems (DEBS)*, pages 20–31, 2012.

[22] Robert Soulé, Martin Hirzel, Robert Grimm, Buğra Gedik, Henrique Andrade, Vibhore Kumar, and Kun-Lung Wu. A universal calculus for stream processing languages. In *European Symposium on Programming (ESOP)*, pages 507–528, 2010.

[23] Robert Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997.

[24] William Thies and Saman Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 365–376, 2010.

[25] Zhihong Xu, Martin Hirzel, and Gregg Rothermel. Semantic characterization of mapreduce workloads. In *Proceedings of the International Symposium on Workload Characterization (IISWC)*, 2013.

[26] Yuan Yu, Pradeep Kumar Gunda, and Michael Isard. Distributed aggregation for data-parallel computing: Interfaces and implementations. In *Symposium on Operating Systems Principles (SOSP)*, pages 247–260, 2009.