BloomUnit: Declarative Testing for Distributed Programs

Peter Alvaro
UC Berkeley
palvaro@cs.berkeley.edu

Andrew Hutchinson
UC Berkeley
ahutchinson@berkeley.edu

Neil Conway
UC Berkeley
nrc@cs.berkeley.edu

William R. Marczak
UC Berkeley
wrm@cs.berkeley.edu

Joseph M. Hellerstein
UC Berkeley
hellerstein@cs.berkeley.edu

ABSTRACT

We present BloomUnit, a testing framework for distributed programs written in the Bloom language. BloomUnit allows developers to write *declarative test specifications* that describe the input/output behavior of a software module. Test specifications are expressed as Bloom queries over (distributed) execution traces of the program under test. To allow execution traces to be produced automatically, BloomUnit synthesizes program inputs that satisfy user-provided constraints. For a given input, BloomUnit systematically explores the space of possible network message reorderings. BloomUnit searches this space efficiently by exploiting program semantics to ignore "uninteresting" message schedules.

We illustrate the utility of BloomUnit by demonstrating an incremental process by which a programmer might provide and refine a set of queries and constraints until they define a rich set of correctness tests for a distributed system.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Distributed debugging, testing tools*

Keywords

BloomUnit, Bloom, disorderly programming, distributed systems, input generation, unit testing

General Terms

Languages, Design, Reliability, Verification

1. INTRODUCTION

Although distributed systems are increasingly ubiquitous, writing correct distributed programs remains difficult and error-prone. Recently, several researchers have proposed using declarative languages drawn from database research to implement distributed systems (e.g., [1, 4, 14, 16]).

Nevertheless, writing correct distributed programs remains challenging because developers must reason about asynchrony, concurrency, and partial failure—achieving confidence in program correctness requires considering the many possible combinations of message reorderings and failures. To address this problem, there are two broad approaches: *formal methods* and *testing*.

Formal methods: In this approach, the developer writes a formal specification of the program's behavior in a modeling language (such as Promela [10]) and then proves that the specification satisfies the developer's desired correctness properties. Formal methods are powerful because they allow the specification's complete state space to be explored, which can find hard-to-reproduce errors. Unfortunately, formal methods typically require considerable user expertise in formal modeling languages and verification tools. Hence, these techniques have not seen widespread developer adoption to date.

Testing: In contrast, many developers write test cases to validate that, for a particular concrete input, the program produces the expected result. Testing has seen widespread adoption because it requires little upfront investment in tools or mathematical training—developers can quickly write test cases for simple behaviors and see an immediate benefit. However, testing distributed programs is relatively ineffective. First, testing cannot easily handle the wide array of network configurations allowed by many distributed programs. Each test case only covers a single system configuration and input set; to test a wide range of configurations, network topologies and program inputs requires writing many test cases and building considerable infrastructure. Second, most test frameworks cannot control network nondeterminism; hence, it is difficult to write a test case to verify that a program is correct in the face of a particular network behavior.

In this paper, we describe our initial work on *BloomUnit*, a testing framework that addresses these concerns. BloomUnit is designed to test programs written in Bloom, a declarative language for distributed programming recently developed by our group [2, 6]. BloomUnit employs three key techniques:

1. BloomUnit users write *declarative test specifications* that describe the intended input-output behavior of the program. Test specifications are written as Bloom queries over the (distributed) execution trace of the program; using Bloom avoids the need for users to learn another language. BloomUnit allows "pay as you go" testing: a simple test specification is equivalent to a single test case, while a more complex test specification can encapsulate many different test cases.

2. Rather than supplying concrete inputs for test specifications, users instead specify the constraints that the input must satisfy; BloomUnit then automatically generates inputs that are consistent with those constraints.

3. Finally, BloomUnit can systematically explore the space of

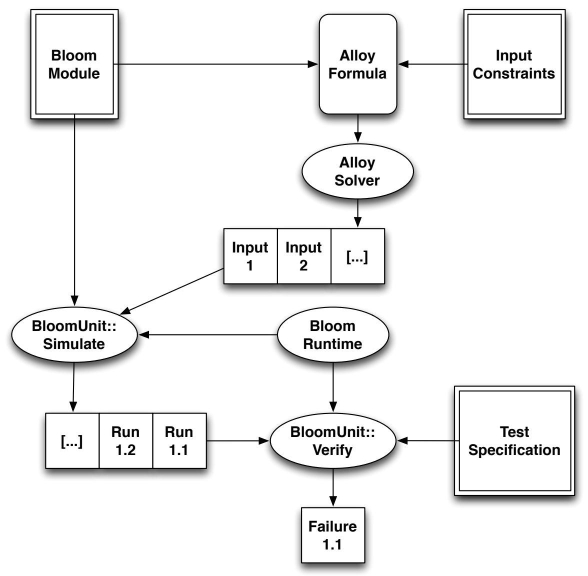

Figure 1: The BloomUnit system.

Op	Valid lhs types	Meaning
<=	**table**, **scratch**	lhs includes the content of the rhs in the current timestep.
<+	**table**, **scratch**	lhs will include the content of the rhs in the next timestep.
<-	**table**	tuples in the rhs will be absent from the lhs at the start of the next timestep.
<~	**channel**	tuples in the rhs will appear in the (remote) lhs at some nondeterministic future time.

Figure 2: Bloom collection types and operators.

possible network behaviors. To reduce the size of this space, we leverage recent results on the connection between logical monotonicity and distributed consistency [2, 3, 9]. BloomUnit can recognize that certain code fragments are insensitive to message delivery order, which reduces the space of message orders that must be explored.

The BloomUnit architecture is sketched in Figure 1. Double rectangles represent inputs provided by the user. A series of inputs are automatically generated from the provided constraints, and for each input, a series of executions is explored by the simulator. An execution may produce a witness to an assertion failure, at which time the simulation halts.

2. BACKGROUND: BLOOM

BloomUnit is a testing framework for programs written in Bloom, a declarative language for programming distributed systems. Due to space constraints, we describe Bloom only briefly here; the interested reader is referred to the Bloom website [6] for details.

Bloom programs are bundles of declarative *statements* about collections of *facts* (tuples). A statement can only reference data that is local to a node. A Bloom program proceeds through a series of atomic “timesteps.” In each timestep, certain “ground facts” exist in collections due to persistence or the arrival of messages from outside agents (e.g., network messages). The statements in a Bloom program specify the derivation of additional facts, which can be declared to exist either in the current timestep, at the very next timestep, or at some nondeterministic time at a remote node. Bloom collection and operator types are listed in Figure 2.

A statement takes the form:

<collection-identifier> <op> <collection-expression>

The left-hand side (lhs) is the name of the output collection and the right-hand side (rhs) is an expression that produces a collection.[1] The operator defines the temporal behavior of the statement—i.e., whether derived lhs facts will appear in the current timestep (<=), be added or removed in the next timestep (<+ or <-, respectively), or will appear at a remote node at a nondeterministic future time (<~).

Bloom provides several collection types to represent different kinds of program state. The contents of a table persist across consecutive timesteps (unless that persistence is interrupted via a Bloom statement containing the <- operator described above). Scratch collections are useful for transient data like intermediate results and “macro” definitions that enable code reuse. Network messages are captured via channel collections; a channel is a scratch collection whose content “appears” at nondeterministic timesteps. When a fact is derived into a channel (via the <~ operator), a network message will be sent to the address contained in the channel’s *location specifier* column (prefixed by @ in the channel’s state declaration).

To enable encapsulation, Bloom programs are organized into *modules* with explicit inputs and outputs declared as interface collections. A module’s interface declarations constitute its *signature*: Figure 3 shows the signature of an abstract delivery protocol. Facts inserted into `pipe_in` are sent over the network to the address appearing in the `dst` field. They subsequently appear in `pipe_out` at the receiver and in `pipe_sent` at the sender (to indicate successful delivery). Different protocol implementations may provide different semantics for this abstract protocol. For example, a reliable delivery protocol would not produce `pipe_sent` facts until an acknowledgment has been received from the recipient, whereas an ordered delivery protocol would constrain the order in which facts appear in `pipe_out`.

```
module DeliveryProtocol
  state do
    interface input, :pipe_in,
      [:dst, :src, :ident] => [:payload]
    interface output, :pipe_sent,
      [:dst, :src, :ident] => [:payload]
    interface output, :pipe_out,
      [:dst, :src, :ident] => [:payload]
  end
end
```

Figure 3: Abstract delivery protocol in Bloom.

The Bloom interpreter provides static analysis capabilities to (conservatively) detect when distributed programs may produce nondeterministic results [2]. We will demonstrate how BloomUnit complements this static analysis, by producing traces of executions that witness predicted nondeterminism, and by building confidence in deterministic programs that may fail the analysis.

The current Bloom prototype is implemented as a domain-specific language (DSL) on top of Ruby. Hence, Bloom programs take the form of annotated Ruby classes; statements and collections are declared in `bloom` and `state` blocks, respectively. A small amount of Ruby code is needed to instantiate the Bloom program and begin executing it; more details are available on the Bloom website [6].

3. TEST SPECIFICATIONS

In this section, we demonstrate how to write declarative test specifications using BloomUnit. As an example, we consider how to write test specifications for a simple FIFO delivery protocol. FIFO delivery

[1]The rhs can include any of the typical relational operators, including selection, projection, join, grouping, and aggregation.

```
module FIFOSpec
  bloom do
    fail <= (pipe_out_log * pipe_out_log).pairs do |p1, p2|
      if p1.src == p2.src and p1.dst == p2.dst and
         p1.ident < p2.ident and p1.time >= p2.time
        ["out-of-order delivery: #{p1.inspect} < #{p2.inspect}"]
      end
    end
  end
end
```

Figure 4: A test specification for FIFODelivery.

```
// Exclusion Constraints
all p1 : pipe_in | p1.src = p1.location
all p1, p2 : pipe_in
  | (p1.src = p2.src and p1.ident = p2.ident)
    ⇒ p1.payload = p2.payload

// Inclusion Constraints
some p1, p2 : pipe_in
  | p1 ≠ p2 ⇒ (p1.src = p2.src and p1.dst = p2.dst)
```

Figure 5: Input constraints for the FIFO delivery protocol, specified using Alloy.

implements the abstract `DeliveryProtocol` interface (Figure 3), with the additional property that if node A sends message m_1 to B and then sends m_2 to B in a subsequent timestep, m_1 should appear in `pipe_out` at B before m_2 does. The FIFO ordering between messages is encoded as a sequence number in the `ident` column.

In traditional testing, a programmer associates a set of concrete inputs with one or more *assertions* that must hold over program output. For example, to test the FIFO delivery protocol we could set up a scenario in which two messages x and y are sent from n_1 to n_2. We would then write an assertion to check that when the messages are delivered at n_2, the order of delivery matches the order in which the messages were sent (i.e., x is delivered before y iff $x.ident < y.ident$). Note that each test describes a *single* concrete scenario; to test that the program operates correctly in another scenario (e.g., involving different numbers of messages or nodes), another test would be required.

A *test specification* is a Bloom program that does not use any temporal operators (<~, <+ and <-) and has a single output interface `fail`. By convention, deriving a tuple into `fail` indicates a violation of the specification. Most specifications will define constraints that must hold over the input and output interfaces of the module under test; these constraints might also refer to temporal details like the ordering of events. To support this pattern, BloomUnit automatically creates a "log table" for every collection in the tested program. Each log table has the suffix "`_log`" and contains all the tuples ever inserted into the collection, along with a `time` column that records the (node-local) timestep when the corresponding fact was derived.

Figure 4 shows an example test specification for the FIFO delivery program. The specification considers all pairs of delivered messages; the specification is violated if two messages were sent from n_1 to n_2 but the order of delivery (given by `time`) is inconsistent with the sender's FIFO order (given by `ident`). Note that because Figure 4 is a Bloom program, we can check that the specification holds over a particular execution trace by simply executing the specification using an unmodified version of the Bloom runtime. Because specifications cannot contain temporal operators, they are essentially "one-shot" queries that can be computed in a single timestep.

4. INPUT GENERATION

When a programmer writes a unit test, they apply a mental model of the space of relevant executions and test a point or set of points from that space. By increasing the number of distinct test points, a programmer increases her confidence that the behavior of the program is consistent with her model of its input/output contract. While declarative unit tests can succinctly "cover" a large space of executions, we still must exploit the programmer's intuitions about the space of possible program inputs to ensure that we cover "relevant" executions. Inspired by work on random testing from user-defined distributions like QuickCheck [8], and by the use of model finding tools to generate inputs from constraints [13], BloomUnit automatically generates a set of plausible inputs for a given Bloom module from user guidance in the form of constraints.

The set of inputs that a Bloom module receives over time during a test run may be viewed as a database instance. For example, an input trace for an implementation of the `DeliveryProtocol` shown in Figure 3 can be represented as an instance with a single relation, `pipe_in`, whose schema is extended with two additional key columns, `time` and `location`, representing logical time and physical location. Because most Bloom programs are distributed, their executions involve multiple agents that cannot in general view the same state at the same time.

We observe two general classes of constraints that aid a programmer in designing input data: *exclusion* and *inclusion* constraints. *Exclusion* constraints rule out inputs that are not possible in the environment in which the program will run. For example, in the delivery protocol shown in Figure 3, each client (identified by the `src` column) identifies each message `payload` with a unique integer `ident`. This is naturally encoded as a functional dependency: $src, ident \rightarrow payload$. We can rule out "spoofing" with another exclusion constraint that ensures that the value of the `location` column be identical to the value of the `src` column. Ideally, programmers should also ensure that the generated input/output instances produce a range of different, "interesting" executions. While exclusion constraints excluded certain records from admissible input instances, *inclusion* constraints ensure that input data can sufficiently cover the space of possible executions by ensuring the *inclusion* of certain records. An execution that tests a FIFO delivery module is uninteresting unless it involves at least two messages to order: we encode this as a constraint on the cardinality of `pipe_in`. Figure 5 shows how this collection of constraints can be concisely expressed in the Alloy language.

We approach the problem of input generation as a problem of model-finding in first-order relational logic. Any model of a set of constraints constitutes an input instance or test. We use the Alloy solver [11], which provides an intuitive first-order language on relations and integration with state of the art SAT solvers for finding satisfying models. Alloy enumerates satisfying models, which we convert to concrete inputs for testing executions of Bloom modules. We use a large number of generated input instances to best cover the space of possible inputs. Alloy's capability to break symmetry during model enumeration increases our confidence that generated inputs are non-isomorphic [11, 13].

BloomUnit analyzes a particular Bloom module's interface and generates default Alloy model constraints. The programmer can then specify additional *exclusion* and *inclusion* constraints. We expect that programmers will typically specify initial constraints before testing and add others during testing in a "pay as you go" fashion.

5. EXPLORATION OF EXECUTION NON-DETERMINISM

Distributed executions are nondeterministic because of asynchronous communication; agents executing the same program over the same inputs may perceive different message orderings in different runs. The BloomUnit system we have described—which ensures that a collection of user-defined assertions are respected as the module under test is presented with a series of generated inputs—is incomplete because any run we might execute on a fixed input is only one of many possible executions. Programmers who have written unit or integration tests for distributed systems are familiar with the miserable phenomenon of nondeterministic test failures and assertions that "usually pass."

To ensure that a given assertion passes on a given set of inputs, we must ensure that it passes in all possible executions over those inputs. Because Bloom relegates all nondeterminism to channel reorderings and omissions, the exploration of all possible executions reduces to the exploration of all possible message orderings and losses. In the style of software model checking [5, 12, 15], we could modify the Bloom runtime to support the exhaustive exploration of message delivery orders and omissions. Unfortunately, such a naive search of this space is intractable.

The CALM theorem [2, 3, 9]—which establishes that logical monotonicity implies eventual consistency—can be used to prune the space of possible message delivery orders. Assuming no message omissions, monotonic code will produce the same output for any given input, regardless of network nondeterminism. Therefore exploring a single delivery order is sufficient to test a monotonic program fragment. Further, some delivery orders are uninteresting even in the execution of a nonmonotonic program. A distributed Bloom program can only produce different outputs for the same inputs when a nonmonotonic operation *follows* an asynchronous operation in the program's dataflow [2]. Hence, to explore all executions that could produce different outputs, we need only explore all delivery orderings for messages that could concurrently be in-flight and destined for an agent at which a nonmonotonic operation will process the message.

To illustrate the effect of nonmonotonicity on program outputs, consider a Bloom program involving agents A, B, C and D. A sends a set of messages (call it M_1) to B, which stores them in a log. C sends a set of messages (M_2) to B, each causing B to count the messages in the log and send the count (M_3) to D, which records the count in a table. Exploring different message delivery order permutations for $M_1 \cup M_2$ will indeed reveal that M_3 has different contents in different executions, reflecting for each $m \in M_3$ how many of the messages in M_1 had arrived when each $m_2 \in M_2$ arrived. The nonmonotonic *count* aggregate continually "revises" its outputs as its inputs change, and because we ship and then store these "estimates," we effectively record evidence of races between the messages in M_1.

However, varying the delivery order of the messages in M_3 has no effect on the final state of the program visible in D's log table. D stores the messages in a set, and when the messages are delivered the set will have the same contents, regardless of the delivery order.

Now consider executions in which messages are lost in addition to being reordered. The space of possible lossy executions is smaller than that of all permutations but is still intractable. If we consider losses of messages in the delivery of $M_1 \cup M_2$, we also must explore all subsets and then all permutations of elements in those subsets.

Exhaustive search of this reduced space of "interesting" message orderings is impractical for even modest inputs: the best that we can do is sample from this space of message orders and omissions. We have modified the Bloom runtime to provide a stochastic model of lossy communication and message reordering. Each node in the distributed system is simulated at a single site, so that the simulator has complete knowledge of the global state. We redefine channel collections so that they buffer tuples before interacting with the network. Each time the system quiesces,[2] the runtime delivers a randomly chosen subset of the buffered messages on all channels. At the end of the execution, a specified number of messages are guaranteed to have been dropped.

```
module CartClientProtocol
  state do
    interface input, :client_checkout,
        [:client, :server, :session] => [:reqid]
    interface input, :client_action,
        [:client, :server, :session, :reqid] => [:item, :action]
    interface output, :client_response,
        [:client, :server, :session] => [:items]
  end
end
```

Figure 6: Shopping cart client protocol.

```
module DisorderlyCart
  include CartProtocol

  state do
    table :action_log, [:session, :reqid] => [:item, :action]
    scratch :item_sum, [:session, :item] => [:num]
    scratch :session_final, [:session] => [:items, :counts]
  end

  bloom :on_action do
    action_log <= action_msg do |c|
      [c.session, c.reqid, c.item, c.action]
    end
  end

  bloom :on_checkout do
    temp :checkout_log <=
        (checkout_msg * action_log).rights(:session => :session)
    item_sum <= checkout_log.group([:session, :item],
                                   sum(:action)) do |s|
      s if s.last > 0
    end
    session_final <= item_sum.group([:session],
                                    accum(:item), accum(:num))
    response_msg <~ (session_final * checkout_msg).pairs
        (:session => :session) do |c,m|
      [m.client, m.server, m.session, c.items.zip(c.counts).sort]
    end
  end
end
```

Figure 7: Implementation of a shopping cart server.

6. CASE STUDY: SHOPPING CARTS

In this section, we illustrate how BloomUnit combines declarative assertions, constraint-guided input generation and exploration of execution nondeterminism to substantially improve the ease and accuracy of testing Bloom programs. As a running example, we will reintroduce the e-commerce scenario presented in Alvaro et al. [2] and describe the process of using BloomUnit to create a rich set of application-specific tests.

Figure 6 presents an abstract interface for a shopping cart service. Clients interact with the service by adding and removing items from the cart (by inserting tuples into `client_action`) and performing a checkout (by inserting a tuple into `client_checkout`). Note that in `client_action`, adding k copies of an item to a cart is represented as a fact with k in the `action` field; removing k copies of an item is represented as a $-k$ action. The `session` attribute of both interfaces is an identifier for a particular cart: we assume that only one client makes updates within a particular session.

[2]This is easy to detect, as we are locally simulating a distributed system. The system is quiescent when all nodes have reached a fixpoint and have no more messages to send.

The shopping cart client implementation (shown in Figure 10 in the appendix) is trivial: it satisfies the `CartClientProtocol` interface simply by sending its inputs to the server and returning the server's response as the client's output. Figure 7 contains an implementation of the shopping cart server code, similar to the "disorderly" design presented in Alvaro et al. The server logs the client actions it receives (lines 11–13). When the server receives a checkout message, the log of actions for the appropriate session is summarized, and the resulting summary is returned to the client (lines 17–28).

In practice, programs are often implemented before being carefully specified. We recount our experience starting with an implementation of the shopping cart service and then using BloomUnit to assist us in "bootstrapping" a set of correctness tests.

6.1 Input constraints

We began by building a set of Alloy constraints to describe legal inputs to the shopping cart service. Given the code in Figures 10 and 7, BloomUnit generated a "default" Alloy specification using the information in the Bloom collection declarations. That is, the default constraints ensured that input tuples contained the appropriate number of columns and respected the interface key constraints, but each column value could be an arbitrary string.

We then ran the system using Alloy-generated input and observed an immediate runtime error. The error occurred because line 20 of Figure 7 evaluates the `sum` aggregate over the `action` column of `client_action`, but Alloy generated arbitrary string values that cannot be summed. We added an exclusion constraint to force the `action` column to take on only integer values. To simplify our model, further, we only consider actions in the set $\{-1, 1\}$.

When we ran the system with this additional constraint, we observed that the `(session, reqid)` key of the `action_log` collection was violated by the statement on lines 11–13. This happened because the cart service assumes that there will not be two different client actions that have the same request ID and session ID. Hence, we added another Alloy constraint to capture key uniqueness:

```
all a1,a2: client_action
  | a1 ≠ a2 ⇒ a1.session ≠ a2.session or a1.reqid ≠ a2.reqid
```

After adding this constraint we observed no more explicit errors, but we also noticed that the system did not produce any facts in the program's output interface. Using Bloom's dataflow debugging tools [6], we observed that the synthesized input data passed through the dataflow only as far as the `checkout_log` collection, which is defined on lines 17–18 as a join between `checkout_msg` and `action_log` on `session`. That is, the cart service did not return any results upon receiving a checkout for an "empty" client session. Depending on application requirements, this might be considered a bug, which could be fixed by replacing the inner join on lines 17–18 with an outer join. Instead, we chose to add another input constraint to ensure that every set of client updates is associated with at least one checkout operation:

```
all a: client_action | some c: client_checkout
  | a.session = c.session
```

After adding this constraint, all test runs produced output and did not yield any runtime exceptions.

Alloy attempts to find "small" models that satisfy the input constraints, so we added a few more inclusion constraints to rule out models that were "too small" for our purposes. We wanted to ensure that all interactions with the shopping cart involve some minimum number of actions. Moreover, an "interesting" execution of a shopping cart service involves multiple updates within the same session—and within a session, to the same item. We added another inclusion constraint to achieve this:

```
some a1,a2: client_action
  | a1.session = a2.session and a1.item = a2.item
```

6.2 Test specification

At this point, BloomUnit was capable of producing reasonable input values for the cart system, but we did not yet know whether the outputs produced by our cart implementation were correct. To establish this, we developed a BloomUnit test specification—that is, a Bloom program whose input was an execution trace of the system, and whose output, if any, constituted correctness violations observed in the trace. Figure 8 shows such a specification. Lines 9–12 separate the input stream `action_msg` into a stream of additions and deletions of individual cart items, summing the total number of each. Lines 13–17 calculate the totals for each item by taking the difference of the sum of additions and the sum of deletions (and ignoring items with a negative count, a business rule also enforced by our cart service). Lines 19–23 flag a violation if the totals computed by the specification differ from those computed by the implementation under test.

When we reran BloomUnit using the test specification, we encountered a correctness violation. Replaying the trace using the dataflow debugger immediately identified the cause: Alloy generated an input set in which a checkout operation was transmitted to the server, followed by additional add/remove operations for the same session. This causes the server's checkout response to not include the "late" client operations. This could be fixed by rejecting "late" operations in the client code, but we chose instead to add an Alloy constraint to require that no client actions be submitted after a checkout operation for a given session:

```
all c: client_checkout | all a: client_action
  | a.session = c.session ⇒ a.time < c.time
```

Next, we used BloomUnit's ability to explore network nondeterminism to examine how the cart service behaved in the presence of message reordering. This revealed a serious bug: because channels do not provide any ordering guarantees, client actions and checkout operations can arrive in arbitrary order at the cart service (regardless of the order in which these messages are generated by the client). Because the cart service performs a nonmonotonic operation (aggregation) on the cart state when a checkout is received, the CALM theorem suggests that the cart server *may* need additional coordination logic to ensure deterministic results. BloomUnit's ability to explore different message schedules confirms that coordination is indeed required. We adjusted the client to include a "manifest" in the checkout message describing all the cart actions that must be reflected in the checkout response, and then modified the server to wait until the manifest has been satisfied before sending back the checkout response. After making this change, our cart service satisfied the test specification.

7. DISCUSSION

When programmers write unit tests, they invest a certain amount of intuition about a program's desired behavior and expected inputs. In return, they increase their confidence that the program behaves correctly and that it will continue to do so as its functionality evolves. Our goal in developing BloomUnit was to retain the simplicity and "pay as you go" nature of testing but to provide a greater "return on investment." In Section 6, we discussed using BloomUnit in one (albeit common) scenario: given an implementation, we incrementally developed a set of input constraints and a test specification. We briefly discuss some other use cases below.

We were not surprised to learn that the first, "uncoordinated" implementation of the disorderly cart had nondeterministic outputs

```
module CartSpec
  state do
    scratch :adds, [:session, :item, :cnt]
    scratch :dels, [:session, :item, :cnt]
    scratch :itemcnt_final, [:session, :item, :cnt]
  end

  bloom do
    adds <= client_action_log {|l| l if l.action == 1}.group
        ([:session, :item], count(:reqid))
    dels <= client_action_log {|l| l if l.action == -1}.group
        ([:session, :item], count(:reqid))
    itemcnt_final <= (adds * dels).outer.pairs
        (:session => :session, :item => :item) do |l, r|
      deletes = r.cnt.nil? ? 0 : r.cnt
      [l.session, l.item, l.cnt - deletes] if l.cnt - deletes > 0
    end

    fail <= (itemcnt_final * client_response_log).pairs
        (:session => :session) do |c, r|
      cnt = r.items.find{|i| i.first == c.item}[1]
      ["#{cnt} vs #{c.cnt}"] if cnt != c.cnt
    end
  end
end
```

Figure 8: Test specification for the shopping cart system.

given message reordering; the CALM static analysis [2] had already warned us of this possibility. Instead of relying on the Bloom compiler to synthesize a heavyweight coordination protocol to rule out this nondeterminism, we chose to coordinate by hand, using domain knowledge about the cart application. When we repaired the program by manually coordinating action and checkout messages, our conservative analysis continued to flag the program as potentially inconsistent. Thus BloomUnit provides a complementary technology to our static analyses. First, it provides a concrete "witness" of the incorrect runs predicted by CALM analysis. Once the program is repaired, BloomUnit gives empirical evidence that the added logic rules out the witness execution and increases our confidence that the program is indeed deterministic.

We have already uncovered a surprising number of bugs—mostly related to message ordering—in our library code using techniques like the one sketched in the previous section. As we add new libraries, we employ a test-driven style, beginning with an input model and a loose specification, and incrementally refining the spec as we develop the program. When we encounter bugs in the field, we generalize from the concrete inputs that reproduce the issue to input constraints that, in principle, cover a larger space of problematic inputs and make the "fix" more general as well.

Designers must sometimes choose between multiple candidate implementations of a particular system (e.g., the two shopping cart implementations presented in Alvaro et al. [2]). Just as we used BloomUnit to establish a notion of "loose determinism" for manually coordinated code, we may use it to reason about "loose equivalence" of programs implementing the same specification. For example, Figure 8 is a valid test specification for both cart implementations; running a large number of tests on both implementations in which different inputs and message orderings are explored builds confidence that the "disorderly" and "destructive" carts essentially implement the same abstract program.

Often, a BloomUnit specification is effectively a local, synchronous implementation of the distributed module under test. In principle, we could synthesize specifications of Bloom programs automatically by rewriting them to remove all temporal operators. Doing so would ensure that message delivery is instantaneous and all races are avoided. We have not adopted this strategy because it fails to address the question of whether the rewritten serial program is correct; as a definition of correctness, it either tautological or incomplete. Instead, in the style of multiversion programming [7], we deliberately write a separate specification, when possible using a different style or different constructs that the original implementation. This gives us greater confidence that the specification does not actually contain and conceal bugs from the original implementation.

Acknowledgments

This work was supported by the Air Force Office of Scientific Research (grant FA95500810352), the Natural Sciences and Engineering Research Council of Canada, the Natural Science Foundation (grants IIS-0713661, CNS-0722077, and IIS-0803690), and gifts from NTT Multimedia Communications Laboratories and Microsoft Research.

8. REFERENCES

[1] P. Alvaro et al. BOOM Analytics: exploring data-centric, declarative programming for the cloud. In *EuroSys*, 2010.

[2] P. Alvaro et al. Consistency Analysis in Bloom: a CALM and Collected Approach. In *CIDR*, 2011.

[3] T. J. Ameloot, F. Neven, and J. Van den Bussche. Relational Transducers for Declarative Networking. In *PODS*, 2011.

[4] N. Belaramani et al. PADS: A Policy Architecture for building Distributed Storage Systems. In *NSDI*, 2009.

[5] D. Beyer et al. The software model checker BLAST: Applications to software engineering. *Int. J. Softw. Tools Technol. Transf.*, 9(5):505–525, Oct. 2007.

[6] Bloom programming language. `http://www.bloom-lang.org`.

[7] L. Chen and A. Avizienis. N-version programming: a fault-tolerance approach to reliability of software operation. In *International Conference on Fault Tolerant Computing*, 1978.

[8] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *ICFP*, 2000.

[9] J. M. Hellerstein. The declarative imperative: experiences and conjectures in distributed logic. *SIGMOD Rec.*, 39:5–19, September 2010.

[10] G. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[11] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[12] R. Jhala and R. Majumdar. Software model checking. *ACM Comput. Surv.*, 41, 2009.

[13] S. Khurshid and D. Marinov. TestEra: Specification-based testing of Java programs using SAT. *Automated Software Engg.*, 11(4):403–434, Oct. 2004.

[14] B. T. Loo et al. Declarative networking: language, execution and optimization. In *SIGMOD*, 2006.

[15] M. Musuvathi et al. CMC: A pragmatic approach to model checking real code. In *OSDI*, 2002.

[16] F. Yang et al. Hilda: A high-level language for data-driven web applications. In *ICDE*, 2006.

APPENDIX

A. ADDITIONAL FIGURES

For completeness, we include the Bloom source code for two of the modules discussed in Section 6: the protocol used to communicate between shopping cart clients and servers (Figure 9), and a simple shopping cart client implementation (Figure 10).

```
module CartProtocol
  state do
    channel :action_msg,
      [:@server, :client, :session, :reqid] => [:item, :action]
    channel :checkout_msg,
      [:@server, :client, :session, :reqid]
    channel :response_msg,
      [:@client, :server, :session] => [:items]
  end
end
```

Figure 9: Shopping cart network protocol.

```
module CartClient
  include CartProtocol
  include CartClientProtocol

  bloom :client do
    action_msg <~ client_action do |a|
      [a.server, a.client, a.session, a.reqid, a.item, a.action]
    end
    checkout_msg <~ client_checkout do |a|
      [a.server, a.client, a.session, a.reqid]
    end
    client_response <= response_msg
  end
end
```

Figure 10: Implementation of a shopping cart client.