McGill University
School of Computer Science
Sable Research Group

A Formalization for Specifying and Implementing Correct Pull-Stream Modules

Sable Technical Report No. sable-2018-01

Erick Lavoie, Laurie Hendren

12-Jan-2018

Contents

List of Figures

List of Tables

Abstract

Pull-stream is a JavaScript demand-driven functional design pattern based on callback functions that enables the creation and easy composition of independent modules that are used to create streaming applications. It is used in popular open source projects and the community around it has created over a hundred compatible modules. While the description of the pull-stream design pattern may seem simple, it does exhibit complicated termination cases. Despite the popularity and large uptake of the pull-stream design pattern, there was no existing formal specification that could help programmers reason about the correctness of their implementations.

Thus, the main contribution of this paper is to provide a formalization for specifying and implementing correct pull-stream modules based on the following: (1) we show the pull-stream design pattern is a form of declarative concurrent programming; (2) we present an event-based protocol language that supports our formalization, independently of JavaScript; (3) we provide the first precise and explicit definition of the expected sequences of events that happen at the interface of two modules, which we call the pull-stream protocol; (4) we specify reference modules that exhibit the full range of behaviors of the pull-stream protocol; (5) we validate our definitions against the community expectations by testing the existing core pull-stream modules against them and identify unspecified behaviors in existing modules.

Our approach helps to better understand the pull-stream protocol, to ensure interoperability of community modules, and to concisely and precisely specify new pull-stream abstractions in papers and documentation.

1 Introduction

Pull-stream [13] is a JavaScript demand-driven functional design pattern based on callback functions that enables the creation and easy composition of independent modules that are used to create streaming applications, initially proposed by Dominic Tarr [11]. It is simple, because it does not require language support other than higher-order functions, yet it is rich enough to provide flow-control and correct termination behavior in case of errors. It also simplifies factoring complex applications into simpler reusable modules. It has already shown its worth by being used in the implementation of a data dissemination protocol for new social applications (`ssb` [14]), in the JavaScript implementation of a peer-to-peer networking stack (`js-libp2p` [9]), in a JavaScript implementation of a peer-to-peer hypermedia protocol (`js-ipfs` [10]), and by being widely downloaded on the `npmjs` website[1]. Furthermore, an open-source community has grown around it and produced more than a hundred compatible pull-stream modules [2].

While the description of the pull-stream design pattern may seem simple, it does exhibit complicated termination cases. For example, we examined and tested the core pull-stream library [13] that has been under development for 5 years now and found two cases of unspecified behaviors [19]. While both are not major issues, and seem to not have created interoperability problems so far, their existence in a well-used library does show that even a seemingly simple callback protocol can have unexpected corner cases.

Despite the popularity and large uptake of the pull-stream design pattern, there was no existing formal specification that could help programmers reason about the correctness of their implementations. Thus, the main contribution of this paper is to provide a formalization for specifying and implementing correct pull-stream modules.

[1] At the time of writing, the library had been downloaded over 90,000 times in the previous month. We believe most of these downloads are from small personal or custom in-house tools.

We arrived at our current results in multiple steps. We first experimented by reimplementing some pull-stream modules in Oz [22, 34], a language with native stream support, to see if modules would be easier to implement and reason about in it. While it gave us insights about the nature of the pull-stream design pattern, the language is not well known and it was hard to explain the insights to a more general audience. We therefore decided to provide a notation with a small number of rules that could capture those insights yet would be independent of both JavaScript and Oz. We then asked questions to the pull-stream community about the expected sequences of events that happen at the interface of two pull-stream modules, which we call the *pull-stream protocol*. This way we identified all valid sequences of events and concisely captured the constraints in our notation. We then used the same notation to specify reference modules that use the full capabilities of the pull-stream protocol. They give a concise, precise, and complete reference of expected behavior, and can be used to test other module implementations. We finally validated our understanding on community contributed modules by implementing our reference modules in JavaScript. Using these modules allowed us to automatically discover unspecified behaviors in well-used modules. This formalization effort therefore helped clarify the expected behavior of pull-stream modules, should help module maintainers to ensure all community modules are inter-operable in the future, and provides a notation for concisely presenting new pull-stream abstractions in modules' documentation and future papers.

1.1 Contributions

In this paper we therefore make the following contributions:

- we show how the pull-stream protocol implements an implicit stream of single-assignment dataflow variables using callbacks and how the benefits of a declarative concurrent programming model also apply to the pull-stream design pattern (Section 3);
- we present an event-based protocol language that is used both to describe the pull-stream protocol and specify the behavior of pull-stream modules, independently of JavaScript (Section 4);
- we provide the first precise and explicit definition of the expected sequences of events that follow the pull-stream protocol at the interface of modules (Section 5);
- we specify parameterized modules that exhibit the full range of behaviors of the pull-stream protocol and can be used as references for implementations and for testing other modules (Section 6);
- we evaluate the conformity of community-contributed modules against our definitions to ensure the definitions adequately describe community expectations and can be used to find modules that do not correctly implement the protocol in all cases (Section 7).

To provide the necessary context, we first introduce the pull-stream design pattern using its JavaScript implementation (Section 2). We then present the previous contributions in the aforementioned sequence. We then compare our work to the existing literature (Section 8). We finally conclude with a brief recapitulation of our contributions and some future research directions (Section 9).

2 Background

The pull-stream design pattern is illustrated in Figure 1. It consists of both a composition mechanism to assemble individual modules in a pipeline and a callback protocol for enabling adjacent modules to communicate.

A pipeline is composed of three types of modules: a single source that produces values, a series of zero or more transformers[2] that modify those values, and a single sink that consumes the values. The composition of multiple transformers is itself a valid transformer. Likewise, the composition of a transformer with a source or a sink is itself a valid source or sink. The stream values flow from left to right, from the source to the sink.

Adjacent modules communicate with a two-parameters callback protocol. The downstream module first makes a request by invoking the output function of the upstream module with a callback. Then, the upstream module replies with an answer by invoking the callback. The first parameter of either function determines the type of operation: the type of a request is determined by the `abort` parameter and the type of an answer is determined by the `done` parameter. There are therefore multiple cases to consider.

In the normal and common case, a request *asks* for a value by invoking the output function with `abort` set to `false` and a callback for the expected answer as a second parameter. An answer then returns a value by invoking the callback with the `done` parameter set to `false` and the value provided as a second parameter.

In addition to the normal case, a request may *abort* processing normally by invoking the output function with `abort` set to `true`, or abort abnormally with an error with `abort` set to `new Error(...)` (which is also truthy[3]).

An answer may also *terminate* the stream normally by setting the first parameter `done` to `true`, or abnormally with an error by setting the first parameter `done` to `new Error(...)`.

2.1 Example Module Implementations

The following modules illustrate the key features of the pull-stream protocol. An example source of values, that counts from 1 to n, is implemented in Figure 2a. Instantiating the module returns a function named *output*. A request is performed by invoking the output function with an abort flag and a callback function x. If the source is aborted from downstream (`abort` is `true` or an error), done will be set to the abort value and x, if defined, is called with the same value. This case is used by the module downstream to abort *early*, before all values have been output. Otherwise, if there are still values to output, x is called with the current value (`done=false`). This is the normal case where a value flows from the output of an upstream module to the input of a downstream module. Finally, in the last case, no more values are available and x is called with *done* (`done=true`). This is the normal termination case, where a source is allowed to output all its values and complete. This source example does not raise errors and therefore the third answer case is not illustrated.

An example of a sink, the complement of a source, is implemented in Figure 2b. The sink takes an r

[2]The existing documentation uses the name *through* for transformers. The original designer later mentioned that he would have preferred transformer but stuck with the original name because the community adopted it. We break community conventions here to favor clarity.

[3]In JavaScript, thruthy values can be used as a true value in conditional statements or expressions.

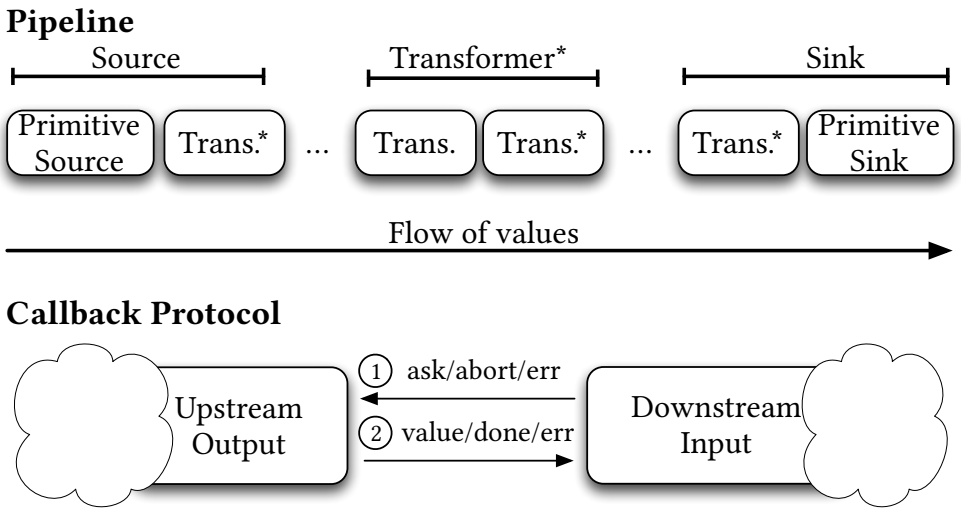


Figure 1: Pull-stream design pattern: pipeline of composable modules on top and callback protocol at the bottom.

```
function source (n) {
  var done = false
  var i = 1
  return function output (abort, x) {
    if (abort)
      return x(done=abort)
    else if (i<=n)
      return x(false, i++)
    else
      return x(done=true)
  }
}
```

(a) Source example.

```
function sink (r) {
  var i = 1
  var abort = false
  function empty() {}
  return function input (request) {
    request(i>r, function x (done, v) {
      if (done) {
        if (done === true) console.log('done')
        else console.error(done)
        return
      }
      console.log(v)
      if ((++i)>r) return request(abort=true, empty)
      else return request(abort=false, x)
    })
  }
}
```

(b) Sink example with aborting support after r requests.

Figure 2: Source and Sink: the *request* parameter of the sink function is the *output* function of the module upstream (source or transformer). The abort flag is made explicit so the inverted logic of the protocol is easier to read.

parameter to define the number of non-abort requests to perform. Instantiating the module returns an *input* function. Invoking the input function with an output function as argument *connects* both. The sink then *requests* values from the upstream module by calling its *output* function[4], hence the parameter is named *request*. Once the input function is invoked, it starts making requests immediately. If 0 requests are demanded, then the output function is *aborted*. Otherwise, a new value is *asked* (`abort=false`). In both cases, the callback x is passed to obtain an answer. The module then waits for an answer to be provided and therefore for x to be invoked. If the source has completed or has failed, `'done'` or an error is printed on the console. If a new value is returned then it is printed on the console and a new request is made if some are left. Since requests are initiated from the sink, the protocol is demand-driven and lazy: a new value is not produced until one has been explicitly *requested*.

An example of a transformer, which takes input values from upstream, applies a function f on them, and outputs the results downstream, is implemented in Figure 3a. It combines both an *input* and an *output* function. The module is instantiated with a single-parameter function f which outputs a result when given an input value. It returns an *input* function, that expects an *output* function as a parameter, similar to a sink. Once invoked, the input function returns a new *output* which may be used as a source. Passing a source to the input function of a transformer therefore returns a new source. The output function of the transformer, directly forwards its *requests* to the upstream module, including the abort cases but with a different callback x rather than xp, to process the incoming value. It then waits for an answer until x is invoked. If the upstream module is done or has failed, it forwards the answer downstream. Otherwise, it applies f on the value v and pass the result downstream by invoking xp.

```
function transformer (f) {
  return function input (request) {
    return function output (abort, xp) {
      request(abort, function x (done, v) {
        // Also handles the error case
        if (done) return xp(done)
        xp(false, f(v))
      })
    }
  }
}
```

(a) Transformer example.

```
// Example: pull(source(10),
//                transformer(function (x) {
//                  return x*2
//                }),
//                sink(Infinity))
function pull () {
  var pipeline = [].slice.call(arguments)
  var output = pipeline.shift()  // Dequeue
  while (pipeline.length > 0) {
    input = pipeline.shift()
    output = input(output)
  }
}
```

(b) Pull helper function to create the stream pipeline.

Figure 3: Transformer and Pull: in JavaScript, `arguments` contains all the call-site arguments regardless of the function definition. Moreover, it behaves like an array but does not have all its methods therefore it is converted to an array using the reflection API (`call` on the slice method of an array that produces the `pipeline` value).

Other modules, such as bi-directional network sockets may also have both an input and an output.

[4]Functions are both objects (nouns) and represent actions (verbs) that are initiated from outside the module. The existing documentation on pull-streams sometimes name the output function *source* and sometimes *read*. When we refer to the object that returns values from inside a module, we call it an *output* function. When we refer to the action of obtaining values from that object from outside the module, we write *requesting* a value and name the function a *request* function.

On one side of the communication channel they can be used as a transformer and on the other as both a source and a sink. In either case, their behavior is similar to the three previous cases shown.

An entire pipeline may be connected by passing the output function of a module upstream to the input function of the next module downstream. The process is illustrated in Figure 3b. The actual implementation[5] is a bit more complicated. It allows any possible combination of modules that is not a full pipeline to return a source, a transformer, or a sink module than can be reused later. It also allows modules to be defined in object form, in which the input and output functions are methods[6].

2.2 Pull-Stream Design Pattern Properties

The pull-stream design pattern provides a combination of many interesting properties:

- An upstream module (producer) and a downstream module (consumer) may both regulate the flow of values by respectively delaying the current answer and the next request;
- The consumer may abort the stream early even though the producer may still have more values to provide;
- Errors are handled within the protocol;
- Any module may propagate an error and has an opportunity for cleaning up after an error or the termination of the stream;
- The values are generated lazily therefore a source may produce infinitely many values;
- Modules may be composed before the construction of the complete pipeline which favors reuse of code when building libraries;
- Both the composition of modules and the construction of a pipeline is declarative: it does not require an understanding of the callback protocol by the users of modules;
- The implementation of modules may use concurrency to improve the overall throughput (ex: it may request multiple values and process them in parallel before returning its results). Outputs may or may not be in order.

3 Insights and Approach

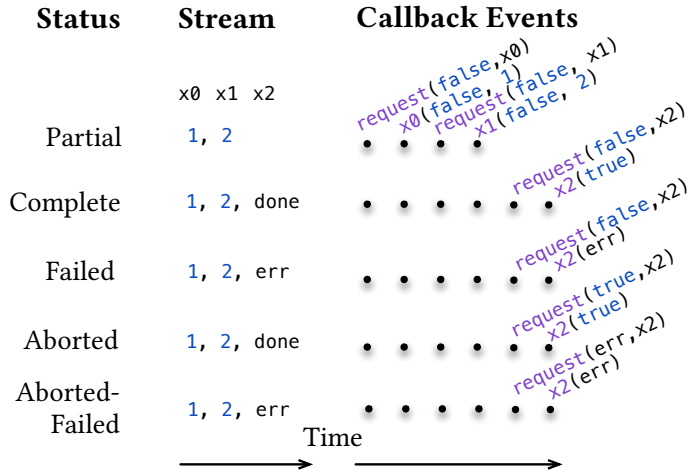
To better understand the pull-stream design pattern, we implemented some pull-stream modules in the Oz language. We explain here the insights we obtained from the experience which in turn informed our formalization approach.

Our key insight is that the sequence of callbacks at the interface of two modules creates an implicit stream. We may view this stream as a stream of *single-assignment dataflow variables*, as illustrated in Figure 4a. Invoking the output function *extends* the stream with a new variable and invoking a

[5] `https://github.com/pull-stream/pull-stream/blob/master/pull.js`

[6] Respectively named source and sink. We prefer input and output because the first letter of each is different and makes it easier to identify the *ports* later in the formalism.

callback *binds* the value of that variable. As each callback is invoked only once, the variables are assigned only once. Since the behavior of modules is triggered by callback events, the assignment of variables can be used for synchronization as in dataflow programming.



(a) Implicit callback streams.

Syntax	JavaScript Events
$ask[\bar{x}_i]$	`request(false, xi)`
$abort[\bar{x}_i]$	`request(true, xi)`
$error[err, \bar{x}_i]$	`request(err, xi)`
$x_i := v_i$	`xi(false, vi)`
$x_i := done$	`xi(true)`
$x_i := err$	`xi(err)`

(b) Formal equivalent.

Figure 4: Callback events that form implicit streams and introduction to their equivalent formal representation.

Programming with concurrent streams of single-assignment dataflow variables is a form of *declarative concurrency*[7], with Unix pipes probably being the best-known example of the programming model. This model makes reasoning about concurrent applications easier than other concurrent models because the non-determinism in the concurrent events is not observable: regardless of the concrete execution order, the result is always the same. In other words, a declarative concurrent streaming program produces the same stream output, regardless of the exact order in which the individual values have been computed. Abstractions built within that model bring the benefits of parallel processing without the complexity of reasoning about multiple orders of executions. Moreover, similar to the composition of functions which is itself a function, the composition of declarative concurrent streaming modules is itself declarative concurrent. It allows complex programs to be easily built from simpler modules.

Some of the most complex pull-stream modules involve managing multiple streams concurrently, `pull-many` [12] and `pull-lend-stream` [18] being two examples. Nonetheless, they are still easy to use because they are declarative concurrent. While their usage is simple, their implementation is not. Unless all possible execution cases are correctly handled, the implementation may break the declarative concurrency model in some cases. Moreover, because there are multiple termination cases in the pull-stream protocol, the correct termination of the implementation is non-trivial to establish for all of them. Both concurrency and termination need to be accounted for to provide correct implementations, therefore the notation we introduce later supports specifying concurrent events and behaviors.

The JavaScript execution model is single-threaded. However, the *asynchronous* execution of some

[7]Chapter 4 of *Concepts, Models, and Techniques of Computer Programming* by Van Roy and Haridi [34] provides an exhaustive discussion of declarative concurrency.

of the libraries available in the execution environment, such as the Document Object Model (DOM) functions in a browser or the input-output functions in Node.js, may happen in parallel. This means that the results, usually obtained in callbacks, may arrive in any order. This is precisely what the inter-leaving semantics[8] for concurrent programs models captures and therefore fits nicely with the actual programming model programmer use when programming JavaScript applications.

Figure 4b provides a preview of the equivalence between the events that happen in the JavaScript implementation we presented in Section 2 and the syntax we use in the rest of the paper. The next section introduce it more formally.

4 Event-Based Protocol Language

In this section we present the notation we use in the rest of the paper. The main goal of our notation was to provide a precise and concise notation to specify the pull-stream protocol and new pull-stream modules in papers that would be independent of JavaScript. Surprisingly to us, the notation we obtained allowed us to avoid the description of the internal state of modules, which makes it usable with many languages, by a more general audience than the original JavaScript community that currently use it.

4.1 Syntax

The syntax of our notation is described using the Extended Backus-Naur Form [1]. We organize the rules in groups and explain them in sequence. In addition to the usual symbols, we sometimes use mathematical syntax elements such as overline (ex: $\overline{x}$), underline (ex: $\underline{x}$), and subscripts (ex: x_2).

Table 1 shows the syntax of basic elements of the notation. *boolean*, *number*, *letter*, *alphanumeric*, *name*, and *variable* have common syntaxes. In addition, we use a particular syntax for stream related concepts. The *stream-index* represents the position of a variable or a value in a stream. It may either be a number that represents a concrete single position in a stream (ex: 1 represents the first position), or a variable that can represent *any* position in a stream. A *stream variable* represents a *single-assignment dataflow variable*, which in JavaScript is implemented with a callback function that should be called only once. A *stream value* represents a value the variable takes once it is bound. The *stream complete* symbol is a special variable value that signifies the stream has completed and has no more values. The *stream failed* symbol is another special variable value that signifies that the stream has failed with an error.

We use some conventions to make the notation easier to read, as listed in Table 2. We use i and j to represent stream indexes, quotes on variables and values (ex: $\bar{x}'_1, v'_i$) to represent respectively the variables and values in a stream that have gone through a single stage of transformation. Any other letters are used to represent function parameters.

The basic elements and syntactic conventions are used to represent events, themselves listed in Table 3. This is useful to present the semantics rules later. An *empty-event* is a special symbol used to express ordering rules without knowing what the target event is going to be already, whereas *arguments* are used in the syntax of other events and may either be a *stream value*, a *stream variable*, or a *stream failed*. A *method call* is an event that represents calls to a method of a pull-stream

[8] *Idem*, Section 4.1.1

Syntax	**Examples**
$boolean = \text{"}\top\text{"} \mid \text{"}\bot\text{"};$	$\top$ (true), $\bot$ (false)
$number = digit, \{digit\};$	0 1337
$letter = character;$	a b z
$alphanumeric = digit \mid letter;$	0 a
$name = letter, \{alphanumeric\};$	ask $abort$
$variable = letter, \{letter\};$	n r te
$stream\text{-}index = variable \mid number;$	i 1 23
$stream\text{-}variable = \overline{letter}_{stream\text{-}index};$	$\bar{x}_i$ $\bar{x}_1$
$stream\text{-}value = \text{"}v\text{"}_{stream\text{-}index};$	v_i
$stream\text{-}complete = \text{"}d\text{"},\text{"}o\text{"},\text{"}n\text{"},\text{"}e\text{"};$	$done$
$stream\text{-}failed = \text{"}e\text{"},\text{"}r\text{"},\text{"}r\text{"}_{[number]};$	err err_1

Table 1: Syntax of basic elements in EBNF (also using overline, underline, and subscript syntax).

Kind	**Convention**	**Example**
Stream index var.	$\text{"}i\text{"} \mid \text{"}j\text{"}$	$\bar{x}_i, v_j$
Transformation stage	$variable^{\{\text{"}\prime\text{"}\}}$	$\bar{x}_i \to \bar{x}'_i, v_i \to v'_i$
Function parameter	other letter(s)	n, r, te

Table 2: Conventions used for variables and values.

module that is not part of the base protocol. For example, some parameters of modules may be dynamically changed during execution and the method call represents when that happened and with which arguments. *request* and *answer* events are the basic events of the pull-stream protocol. As explained in Section 2 and Table 4b, a request corresponds to the `request` function call, and an answer corresponds to the invocation of the callback provided in the request. A *port* represents where an event is initiated, such as a downstream module in the case of a request. Ports enable the description of complex modules which interact with multiple streams at a time. An *event* is a port and one of the other event types mentioned. A *history* is a sequence of events separated by commas and represent *all* the concrete events that happened during an execution.

Syntax	**Examples**
$empty\text{-}event = \text{"}e\text{"},\text{"}m\text{"},\text{"}p\text{"},\text{"}t\text{"},\text{"}y\text{"};$	$empty$
$argument = stream\text{-}value \mid stream\text{-}variable \mid stream\text{-}failed;$	v_i $\bar{x}_i$ err
$arguments = argument, \{\text{"},\text{"} argument\};$	v_i $err, \bar{x}_i$
$method\text{-}call\text{-}event = name_{stream\text{-}index}, [\text{"}(\text{"}, arguments, \text{"})\text{"}];$	$lendStream_1$
$request\text{-}event = name, [\text{"}[\text{"}, arguments, \text{"}]\text{"}];$	$abort$ $ask[\bar{x}_i]$
$answer\text{-}value = stream\text{-}value \mid stream\text{-}failed \mid stream\text{-}complete;$	v_i err $done$
$answer\text{-}event = stream\text{-}variable, \text{" : "}, \text{" = "}, answer\text{-}value;$	$\bar{x}_i := v_i$
$port\text{-}index = [number \mid variable];$	s
$port = (uppercase\text{-}letter, \{uppercase\text{-}letter\})_{port\text{-}index};$	SSU_1 DI
$event = [port, \text{" : "}], (method\text{-}call\text{-}event \mid request\text{-}event \mid answer\text{-}event \mid empty\text{-}event);$	I: abort
$history = event, \{\text{"},\text{"}, event\};$	$ask[\bar{x}_1], \bar{x}_1 := v_1$

Table 3: Syntax of events in EBNF (also using overline, underline, and subscript syntax).

Sometimes multiple histories may be possible for a given protocol or module execution. We capture the underlying structure with a *partial order on events*, as illustrated in Table 4. The temporal dependency between events is represented with the $\rightarrow$ operator. Concurrent events are represented with the $\wedge$ operator. A choice among multiple mutually-exclusive choices is represented with the $|$ operator. We use partial orders to describe a protocol as a sequence of events that captures all possible inter-leavings and possibilities concisely.

Syntax	**Examples**
$\textit{partial-order} = \textit{event}$	$\textit{abort}$
$\mid \textit{partial-order}, " \rightarrow ", \textit{partial-order}, \{" \rightarrow ", \textit{partial-order}\}$	$ask[\bar{x}_1] \rightarrow \bar{x}_1 := v_1$
$\mid \textit{partial-order}, " \wedge ", \textit{partial-order}$	$\bar{x}_1 := v_1 \wedge \bar{x}_1 := v_2$
$\mid \textit{partial-order}, " \mid ", \textit{partial-order}$	$abort[\bar{x}_1] \mid error[err, \bar{x}_1]$
$\mid "(", \textit{partial-order}, ")";$	$(abort)$

Table 4: Syntax of partial orders in Extended Backus-Naur form with examples. Operators in order of priority: $\rightarrow$, $\wedge$, $|$. $\rightarrow$, $\wedge$, $|$ are all associative.

Pull-stream modules react to external events by initiating new events. The syntax for describing the rules that describe their behavior is given in Table 5. It is built around an *antecedent* that is itself a partial order augmented with additional operations. *Relations* are used to reason about stream variables and values and decide whether a rule applies for a given history. The rest of the antecedent operators are essentially a first-order logic with conjunction $\wedge$ (which also has the meaning of concurrent), disjunction $|$ (which also has the meaning of mutual exclusion), negation $\neg$, and *quantifiers*, to reason about all possible events of a certain type, or finding events in a history that satisfy some conditions. A rule $\Rightarrow$ is a combination of an antecedent and the event it generates if the antecedent is true.

To conclude the syntax, when multiple choices are possible for events in a partial order or an antecedent, it may be hard to read and parse them. We therefore sometimes use a vertical choice operator as syntactic sugar, shown in Figure 5.

$$\begin{bmatrix} expr_1 \\ expr_2 \\ ... \\ expr_n \end{bmatrix} = "(", expr_1, " \mid ", expr_2, " \mid ", ..., " \mid ", expr_n, ")"$$

Figure 5: Syntactic sugar for a disjunction of antecedents or partial orders.

4.2 Semantics

We explain the semantics for the syntax described previously in this section. We simplify its description by first normalizing antecedents and partial orders with rewrite rules. We then describe what a history is and provide operations on it and how it is extended. We finish the section with a simple ping-pong protocol to illustrate how the semantics work.

Syntax	**Examples**
$quantifier = \text{"}\exists\text{"}_{\{variable\}} \mid \text{"}\forall\text{"}_{\{variable\}};$	$\exists_i$
$relation\text{-}operator = \text{" = "} \mid \text{" } \neq \text{ "} \mid \text{" < "} \mid \text{" } \leq \text{ "} \mid \text{" > "} \mid \text{" } \geq \text{ ";}$	$< \quad \geq$
$relation\text{-}operand = stream\text{-}index\text{-}variable \mid variable \mid number;$	$i \; n \; 1$
$relation = relation\text{-}operand, relation\text{-}operator, relation\text{-}operand,$ $\{relation\text{-}operator, relation\text{-}operand\};$	$r < i < n$
$antecedent = boolean \mid event \mid relation$	
$\mid antecedent, \text{" } \rightarrow \text{ "}, antecedent, \{\text{" } \rightarrow \text{ "}, antecedent\}$	
$\mid antecedent, \text{" } \wedge \text{ "}, antecedent$	
$\mid antecedent, \text{" } \mid \text{ "}, antecedent$	$i < n \wedge I : ask[\bar{x}_i]$
$\mid \text{"("}, antecedent, \text{")"}$	
$\mid \text{"}\neg\text{"}, antecedent$	
$\mid quantifier, antecedent;$	
$rule = antecedent, \text{" } \Rightarrow \text{ "}, event$ $\mid event, \text{" } \Leftarrow \text{ "}, antecedent;$	$C : ping[\bar{x}_i] \Rightarrow S : \bar{x}_i := v_i$

Table 5: Syntax of rules in Extended Backus-Naur form (also using overline, underline, subscript, and superscript syntax) with examples. Operators in order of priority: *relational-operator*, $\rightarrow$, $\neg$, *quantifier*, $\wedge$, |. $\rightarrow$, $\wedge$, | are all associative.

In the next sections, we do not describe the meaning of a partial order because it is the same as an antecedent. Any rules for antecedent that use the same syntax therefore also apply to partial orders. For conciseness, we use the following aliases for booleans, relations, antecedents, and events:

$$b ::= boolean, \quad r ::= relation, \quad a ::= antecedent, \quad e ::= event \text{ except } empty$$

4.2.1 Normalization

We normalize partial orders and antecedents to make it easier to reason about them. The process consists in rewriting the expressions such that there are only sequences of events separated by conjunctions ($\wedge$) and disjunctions (|). A sequence is an arbitrarily large but finite list of events that happen before one another defined as:

$$seq ::= \begin{cases} e \\ seq \rightarrow e \end{cases}$$

The rewriting rules are shown in Figure 6. The first rule says that a sequence seq_1 followed by two concurrent sequences seq_2 and seq_3 is the same as the conjunction of seq_1 followed by each of the sequences. The second rule is similar with two concurrent sequences followed by a single sequence. The third rule says that a sequence seq_1 followed by a choice of either two sequences seq_2 or seq_3 is the same as a choice between the sequence seq_1 immediately followed by seq_2 or seq_1 immediately followed by seq_3. All the other rewriting rules show how to remove an empty event from a sequence or an antecedent.

$$seq_1 \to (seq_2 \wedge seq_3) \rightsquigarrow (seq_1 \to seq_2) \wedge (seq_1 \to seq_3)$$
$$(seq_2 \wedge seq_3) \to seq_1 \rightsquigarrow (seq_2 \to seq_1) \wedge (seq_3 \to seq_1)$$
$$seq_1 \to \begin{bmatrix} seq_2 \\ seq_3 \end{bmatrix} \rightsquigarrow seq_1 \to (seq_2 \mid seq_3) \rightsquigarrow (seq_1 \to seq_2) \mid (seq_1 \to seq_3)$$
$$\begin{bmatrix} seq_1 \\ seq_2 \end{bmatrix} \to seq_3 \rightsquigarrow (seq_1 \mid seq_2) \to seq_3 \rightsquigarrow (seq_1 \to seq_3) \mid (seq_2 \to seq_3)$$
$$seq \to empty \rightsquigarrow seq \qquad empty \wedge a \rightsquigarrow a$$
$$empty \to seq \rightsquigarrow seq \qquad a \wedge empty \rightsquigarrow a$$
$$seq_1 \to empty \to seq_2 \rightsquigarrow seq_1 \to seq_2 \qquad a \mid empty \rightsquigarrow a$$
$$empty \mid a \rightsquigarrow a$$

Figure 6: Rewriting rules for normalization. The left-hand side of the $\rightsquigarrow$ operator is rewritten to its right-hand side.

We have verified that the normalization terminates by using the Aprove method [20], by testing the rules listed in Appendix A with an automated assistant [3]. This ensures that all possible antecedent or partial order can be rewritten in a finite number of steps.

4.2.2 History

The following definitions define what a *history* is and various operations on it.

A history H is a stream of past events that is either empty or has an event following a shorter history:

$$H ::= \begin{cases} \langle\rangle \\ H, e \end{cases}$$

The definition of a history is similar to a sequence of events seq. A history may contain additional events that are not part of a defined sequence while still correctly following that sequence.

Similar to the syntax definition in Figure 3, an event is either a function call, a request, or an answer (there are no empty event in a history).

Every event of a history H is syntactically unique. There is no variable in the syntax of events in a history, every *stream-index* and *port-index* is a number and not a variable. Therefore we define two events as equal if they are written with exactly the same symbols in the same positions.

Consequently, an event e is part of a history H, written $e \in H$, if an event that appears in H is equal to e. The opposite relation is the negation of the previous:

$$e \in H ::= \begin{cases} H = \langle\rangle & false \\ H = (H', e') & \begin{cases} e = e' & true \\ e \neq e' & e \in H' \end{cases} \end{cases} \qquad e \notin H ::= \neg(e \in H)$$

The *depth* of an event, written $depth(e, H)$, represents how far it appeared in the past. It is determined by counting the number of events that happened after until now. The operation is only defined if $e \in H$:

$$depth(e, H) ::= H = H', e'; \begin{cases} e = e' & 0 \\ e \neq e' & 1 + depth(e, H') \end{cases}$$

An event e_2 therefore *follows* an event e_1 if they are both part of the same history and e_1 is further in the past (i.e. its depth is greater). The $before_H$ relationship is not defined if one or both of the operands are not in the history:

$$e_1 \ before_H \ e_2 ::= e_1 \in H \wedge e_2 \in H \wedge (depth(e_1, H) > depth(e_2, H))$$

An event e_2 *always follows* an event e_1 if the relationship is true for all possible histories.

History H *entails* ($\vDash$) an expression a (antecedent), which can be a complex expression that represents events, boolean values, relation operations between integers, or logic operations on expressions, if the following inference rules are *true*[9]:

$$\frac{e \in H}{H \vDash e} \quad \frac{seq = e \quad e \ beforeH \ e'}{H \vDash seq \rightarrow e'} \quad \frac{seq = seq' \rightarrow e \quad H \vDash seq' \quad e \ before_H \ e'}{H \vDash seq \rightarrow e'} \quad \frac{b = \top}{H \vDash b}$$

$$\frac{\frac{r \ is \ true}{r}}{H \vDash r} \quad \frac{a = r \quad H \vDash r}{H \vDash a} \quad \frac{a = e \quad H \vDash e}{H \vDash a} \quad \frac{a = seq \quad H \vDash seq}{H \vDash a} \quad \frac{a = b \quad H \vDash b}{H \vDash a} \quad \frac{\neg H \vDash a}{H \vDash \neg a}$$

$$\frac{H \vDash a_1 \quad \neg H \vDash a_2}{H \vDash a_1 \mid a_2} \quad \frac{\neg H \vDash a_1 \quad H \vDash a_2}{H \vDash a_1 \mid a_2} \quad \frac{H \vDash a_1 \quad H \vDash a_2}{H \vDash a_1 \wedge a_2} \quad \frac{\exists_{vars} H \vDash a}{H \vDash \exists_{vars} a} \quad \frac{\forall_{vars} H \vDash a}{H \vDash \forall_{vars} a}$$

By convention, indexes for stream variables and stream values start at 1 and increase by one for each immediately following item. If an antecedent has free variables in it, i.e. variables not mentioned in a quantifier, it is assumed that the quantifier $\exists_{i \in \mathcal{N}}$ for each free variable i. For example, suppose a history $H = ask[\bar{x}_1], abort$. Then $H \vDash ask[\bar{x}_i]$ because i is a free variable in $ask[\bar{x}_i]$, which is equivalent to writing $H \vDash \exists_{i \in \mathcal{N}} ask[\bar{x}_i]$. For $i = 1$, $H \vDash ask[\bar{x}_1]$, because $ask[\bar{x}_1] \in H$ (it is the first element by definition). The use of stream index variables therefore enables matching *series* of events in a history.

4.2.3 History Progression

The specification for modules uses a single rule that defines the behavior of the implication $a \Rightarrow e$. Intuitively, if the antecedent expression a is entailed by the the current history $H \vDash a$ and the event e has not already happened[10], then e should eventually happen and the new history H' will be the old history H extended by e. During the execution, it is possible that multiple implication rules may match at the same time, which enables concurrency. In this case, an implementation is free to execute them in any order. To be correct, an implementation needs to correctly follow the pull-stream protocol regardless of which rule was executed first.

[9] The bottom of an inference rule is true if all the conditions on top are true.

[10] Remember that each event is syntactically unique.

Formally, we therefore define the behavior of the implication ($\Rightarrow$) as:

$$H.(a \Rightarrow e) \succ H' \quad \frac{H \vDash a \quad e \notin H}{H.(a \Rightarrow e) \succ H, e} \quad \frac{\neg H \vDash a}{H.(a \Rightarrow e) \succ H} \quad \frac{e \in H}{H.(a \Rightarrow e) \succ H}$$

Finally, note that by definition, $a \Rightarrow e$ implies that for any event e' in a, $H \vDash e' \rightarrow e$. As a final remark, it is interesting to note that a history is itself a stream of events and can therefore be represented and processed using pull-streams!

4.2.4 Step-by-Step Ping Pong Example

We illustrate the syntax and semantics with a simple ping-pong protocol based on a single request event $ping[\bar{x}_i]$ and a single answer event $\bar{x}_i := pong$[11]. Let's assume a client C initiates a ping and a server S answers with a pong. The client and the server are respectively ports on which events happen.

The client sends a first ping, waits for an answer from the server and then sends the next ping, infinitely often. The behavior of the client can be described with the following rules:

$$C : ping[\bar{x}_1] \Leftarrow \neg C : ping[\bar{x}_1]$$
$$C : ping[\bar{x}_i] \Leftarrow S : \bar{x}_{i-1} := pong$$

The server answers each ping with a pong infinitely often. Its behavior can be described with the following rule:

$$S : \bar{x}_i := pong \Leftarrow C : ping[\bar{x}_i]$$

As you have seen, by convention we write the event initiated by the port on the left and the events on which it depends on the right. An execution of that protocol according to the semantics would follow those steps. Initially, the history is empty ($H = \langle\rangle$). The only rule that can extend the history is the first client rule and since there are no event in the history, it is true that $C : ping[\bar{x}_1] \notin H$. All other rules are missing a past event on which they depend and therefore cannot extend the history.

The first rule therefore applies, and now $H = \langle\rangle, C : ping[\bar{x}_1]$. The first rule of C no longer applies (and will never again). The second rule of C still needs an answer. The only rule that applies is that of S. H is again extended and is now $H = \langle\rangle, C : ping[\bar{x}_1], S : \bar{x}_1 := pong$. The rule of S no longer applies. But the second rule of C now does, because $\exists_{i \in \mathcal{N}}$ when $i = 2$. In such a case, the rule is equivalent to:

$$C : ping[\bar{x}_2] \Leftarrow S : \bar{x}_1 := pong$$

It applies because $S : \bar{x}_1 := pong \in H$. Then H is extended with the new ping, and so on and so forth forever. The invariant of the ping-pong protocol may be abstracted as the following partial-order:

$$C : ping[\bar{x}_i] \rightarrow S : \bar{x}_i := pong$$

And a corresponding generator function that creates sequences of events according to the protocol for a finite number of steps can be written:

We now use a similar exposition to present the pull-stream protocol, first by introducing the protocol and then reference modules that generate valid sequences of events.

[11] In this protocol, $\forall_i v_i = pong$.

Signature: $ping\_pong\_sequence(\underline{n}, C, S)$
Parameters:
$\underline{n}$ is the number of ping requests.
Implementation:

$$\underline{events}(\underline{i}) ::= \begin{cases} i \leq n & C : ping[\bar{x}_i] \rightarrow S : \bar{x}_i := pong \rightarrow \underline{events}(i+1)) \\ i > n & empty \end{cases}$$

returns $\underline{events}(1)$

Figure 7: Procedure to generate a partial order of events for the ping-pong protocol.

5 Pull-Stream Protocol

In this section, we define what sequences of events follow the pull-stream protocol at a given interface between two modules. There was no previous formal specification that described it, we obtained them by asking questions to the community about the usual expectations on the behavior of pull-stream modules[12] and by testing existing modules. We reformulated them to be concise, precise, and complete.

5.1 Overview

The protocol covers two possible sequences, a *normal sequence* in which the upstream module produces all its values and then stops, and an *early termination sequence* in which the upstream module is terminated by the downstream module before all values have been produced. We cover both in turn. Note that the protocol only specifies the sequence of events between two modules on a single interface: it does not specify constraints *between* two interfaces, such as between the input and output of a transformer. These constraints are instead part of the *specification* of a module and therefore allow module designers a maximum amount of flexibility.

To present both the normal sequence and the early termination sequence, we start with concrete sequences of events. Since many sequences are similar to one another apart from the specific ordering of events, we then capture the regularity in a partial order using the *happens before* ($e_1 \rightarrow e_2$) relation between events. Finally, we generalize the partial orders to functions that generates them for specific cases given some parameters that are specific to each of the two cases. These functions are described in Figure 9 and 10.

Note that the current protocol mostly forbids concurrent requests[13] or out-of-order answers. These additional cases are not supported to simplify the implementation of modules in JavaScript. We discuss the additional possibilities that could be offered by the protocol in Appendix B, which may be easier to use in languages with dataflow variables such as Oz. These additional cases illustrate the expressiveness of our notation for concurrency. Readers solely interested in the pull-stream protocol implementation for JavaScript may safely skip these explanations.

[12]The entire discussion is in this issue on the main pull-stream repository on GitHub: `https://github.com/pull-stream/pull-stream/issues/100`.

[13]The early termination is an exception to the rule to allow aborting before an answer has been provided.

5.2 Pull-Stream Events

Table 4b showed examples of the events of the pull-stream protocol. Figure 8 present them again regrouped in different categories to ease the presentation of the protocol and later specifications.

$$\underline{terminate}(\bar{x}_i) ::= \begin{bmatrix} abort[\bar{x}_i] \\ error[err, \bar{x}_i] \end{bmatrix} \qquad \underline{terminated}(\bar{x}_i) ::= \begin{bmatrix} \bar{x}_i := done \\ \bar{x}_i := err \end{bmatrix}$$

$$\underline{request}(\bar{x}_i) ::= \begin{bmatrix} ask[\bar{x}_i] \\ \underline{terminate}(\bar{x}_i) \end{bmatrix} \qquad \underline{answer}(\bar{x}_i) ::= \begin{bmatrix} \bar{x}_i := v_i \\ \underline{terminated}(\bar{x}_i) \end{bmatrix}$$

(a) Requests. (b) Answers.

Figure 8: All possible pull-stream events regrouped by category under the *request*, *terminate*, *answer*, and *terminated* functions.

All the *requests* are initiated downstream. All requests expect an answer in $\bar{x}_i$ ($request(\bar{x}_i)$). *Terminate* requests are used to terminate the stream *early* before all values have been produced. A normal termination, such as when no more values are needed, is performed with $abort[\bar{x}_i]$. An abnormal termination, such as when an internal error prevents a module from receiving more values, is performed with $error[err, \bar{x}_i]$. All *answers* are initiated upstream and provide either a *value* v_i or signal that the stream has *terminated* either normally with *done* or abnormally with *err*.

5.3 Normal Sequence

The *normal sequence* represents sequences of events in which the number of *ask* requests (r) from the module downstream is strictly greater than the number of values produced (n). Therefore the upstream module produces all its values and then terminates the stream. After receiving a terminated answer, the module downstream *must* never make additional requests, therefore $r = n+1$.

We describe the interaction between the input I of a downstream module and the output O of the module immediately upstream. For example, in the case where two requests are made in sequence ($I : ask[\bar{x}_1]$ and $I : ask[\bar{x}_2]$), and one value ($O : \bar{x}_1 := v_1$) and a terminated event ($O : terminated(\bar{x}_1)$) are produced, there are all six possible sequences of events.

The first two cases correspond to *co-routining* between the downstream module and the upstream module: the downstream module *asks* for a value, waits for an answer, then *asks* the next value.

$$I : ask[\bar{x}_1], \quad O : \bar{x}_1 := v_1, \quad I : ask[\bar{x}_2], \quad O : \bar{x}_2 := done$$
$$I : ask[\bar{x}_1], \quad O : \bar{x}_1 := v_1, \quad I : ask[\bar{x}_2], \quad O : \bar{x}_2 := err$$

We abstract the two termination cases ($\bar{x}_2 := done$ and $\bar{x}_2 := err$) in a single abstract event $terminated(\bar{x}_2)$ and express the ordering constraint as a sequence[14]:

$$I : ask[\bar{x}_1] \rightarrow O : \bar{x}_1 := v_1 \rightarrow I : ask[\bar{x}_2] \rightarrow O : terminated(\bar{x}_2)$$

[14] As explained in Section 4.2.1 using a sequence rather than a concrete history allows multiple concurrent streams to generate their events in a single global history of interleaved events.

The four other cases correspond to having two *concurrent asks* made before the answers have arrived, two cases in which the answers arrive in order and two out-of-order. These last four cases could possibly improve performance in some cases but implementations of pull-stream modules do not use them because the added implementation complexity is not worth the gain. They are not necessary to understand the protocol as it is currently used in practice but the interested reader may still find a complete description in Appendix B.1.

We generalize the *co-routining* sequence to an arbitrary number of values n and any pair of input and output ports I and O with the *normal_sequence* function of Figure 9. The function should only be used if $r = n + 1$, otherwise the other function of Figure 10 should be used[15]. The *normal_sequence* function generates a sequence one request and answer at a time starting from the first ($events(1)$). There are two different cases for generating a new request and answer. If $i \leq n$ than a request should be answered by a value before the next request is initiated. If $i = n + 1$ then there are no more values and the request will be answered by a *terminated* event.

Signature: $normal\_sequence(n, I, O)$
Parameters:
n is the number of values in the stream ($n \geq 0$)
Implementation:

$$\underline{events}(\underline{i}) ::= \begin{cases} i \leq n & I : ask[\bar{x}_i] \rightarrow O : \bar{x}_i := v_i \rightarrow \underline{events}(i+1)) \\ i = n+1 & I : ask[\bar{x}_i] \rightarrow O : \underline{terminated}(\bar{x}_i) \end{cases}$$

returns $\underline{events}(1)$

Figure 9: Procedure to generate a partial order of events for the normal sequence (in this case, the partial order is a sequence).

The function can be used to generate all possible sequences of events. For example, for $n = 1$ the partial order of events for the normal sequence generated by the function is:

$$I : ask[\bar{x}_1] \rightarrow O : \bar{x}_1 := v_1 \rightarrow I : ask[\bar{x}_2] \rightarrow O : \underline{terminated}(\bar{x}_2) \rightarrow empty$$

Making both $terminated(\bar{x}_i)$ cases explicit:

$$I : ask[\bar{x}_1] \rightarrow O : \bar{x}_1 := v_1 \rightarrow I : ask[\bar{x}_2] \rightarrow \begin{bmatrix} O : \bar{x}_2 := done \\ O : \bar{x}_2 := err \end{bmatrix}$$

Using the rewriting rules of Section 4.2.1 we obtain the two possibilities that correspond to the two examples given at the beginning of the section:

$$(I : ask[\bar{x}_1] \rightarrow O : \bar{x}_1 := v_1 \rightarrow I : ask[\bar{x}_2] \rightarrow O : \bar{x}_2 := done) \mid$$
$$(I : ask[\bar{x}_1] \rightarrow O : \bar{x}_1 := v_1 \rightarrow I : ask[\bar{x}_2] \rightarrow O : \bar{x}_2 := err)$$

5.4 Early-Terminated Sequence

The *early_terminated_sequence* represents sequences of events in which the number of *ask* requests is less or equal to the number of values that could be produced by the upstream module ($r \leq n$) .

[15] $r > n + 1$ is incorrect.

The last *ask* request is always followed by a *terminate* request. The upstream module is therefore *terminated* earlier than in the normal sequence and the stream is potentially shorter than it would have been otherwise[16].

For example, in the case of a stream of two values ($n = 2$), terminated early on the second request ($r = 1$), the following sequences of events are possible. In these sequences, we do not list the normal and abnormal termination requests and terminated answers explicitly, we use $I : \underline{terminate}(\bar{x}_i)$ and $O : \underline{terminated}(\bar{x}_i)$ instead.

The *co-routining* case is analogous to the *normal sequence*: an answer is expected before the next request is initiated, but the last request is a termination request instead of an *ask* request and the second value is ignored:

$$I : ask[\bar{x}_1],\ O : \bar{x}_1 := v_1,\ I : \underline{terminate}(\bar{x}_2),\ O : \underline{terminated}(\bar{x}_2)$$

At first glance, it may seem that it is sufficient to support early termination. However, because an answer is expected for each request before issuing the next, it is not possible to abort the upstream module if an answer takes too long to arrive. Because of that limitation, a limited form of *in-order* concurrency is supported: a terminate request may be concurrently initiated if the last answer takes too long to arrive. The previous answer is terminated as soon as possible, and the answer for the termination request comes right after:

$$I : ask[\bar{x}_1],\ I : \underline{terminate}(\bar{x}_2),\ O : \underline{terminated}(\bar{x}_1)\ O : \underline{terminated}(\bar{x}_2)$$

As for the normal sequence, there are other possible concurrent cases that are not used. The interested reader may still find a complete description in Appendix B.2.

As for the normal sequence, we generalize the sequence to an arbitrary number of values n, number of *ask* requests r, with an additional parameter w is used to distinguish between the case where the last answer was waited for or not. The generalization for the *early_terminated_sequence* function is shown in Figure 10. This function applies in cases where $r \leq n$, otherwise the *normal_sequence* applies.

The function generates a partial order, one request and answer at a time starting from the first ($events(1, empty)$). Note that it uses T as a parameter to save the terminated answer in case $w = \top$ to place it correctly later in the sequence. There are four different cases in the generation of events. If $i \leq r - 1$ then a value matching the *ask* is produced. If $i = r$ and the answer is waited for ($w = \top$), it is similar to the first case. If $i = r$ but the answer is not waited for ($w = \top$), then an *ask* with a terminated answer is produced and saved for later in T. In the last case where $i = r + 1$, the terminate request is generated and an additional terminated answer T, empty or not depending on whether w was $\top$ or $\bot$, and finally the answer for the terminate request.

5.5 Correctness

The sequence of events (*history* H) observed at the interface of two modules with input (I) and output (O) is *correct* if it follows either case of the pull-stream protocol and does not generate more events than those. In practice, we check the conformity of actual executions with the following invariants implemented in a transformer module [28] placed between two other modules:

[16] For the case where $r = n$ the upstream module actually terminate on the same request index it would have in the normal sequence. However, it terminates on a *terminate* request rather than an *ask* request.

Signature: $early\_terminated\_sequence(n, r, w, I, O)$
Parameters:
n is the number of values in the stream ($n \geq 0$)
r is the number of ask requests performed from downstream before terminating ($0 \leq r \leq n$)
w is whether the last value had been waited for before terminating ($w = \top$) or not ($w = \bot$)
Implementation:

$$\underline{events}(\underline{i}, \underline{T}) ::= \begin{cases} i \leq r-1 & I : ask[\bar{x}_i] \rightarrow O : \bar{x}_i := v_i \rightarrow \underline{events}(i+1, T) \\ i = r \wedge w & I : ask[\bar{x}_i] \rightarrow O : \bar{x}_i := v_i \rightarrow \underline{events}(i+1, T) \\ i = r \wedge \neg w & I : ask[\bar{x}_i] \rightarrow \underline{events}(i+1, O : \underline{terminated}(\bar{x}_i)) \\ i = r+1 & I : \underline{terminate}(\bar{x}_i) \rightarrow T \rightarrow O : \underline{terminated}(\bar{x}_i) \end{cases}$$

returns $\underline{events}(1, empty)$

Figure 10: Procedure to generate a partial order of events for early terminated sequences.

1. No additional request after a *terminate* request or a *terminated* answer;
2. Every expected answer ($\bar{x}_i$) eventually happens;
3. Every expected answer ($\bar{x}_i$) happens only once;
4. Expected answers happen in the creation order of their stream variable ($\bar{x}_i$);
5. No concurrent ask requests;
6. (If the stream is finite), the stream is eventually terminated.

6 Reference Modules

In this section, we provide reference modules that can guide a correct and complete implementation. They are useful both as an illustration of our notation for specifying the behavior of modules as well as for testing other modules. For the latter case, we implemented them in JavaScript [29][17] and we report on our experiments in testing community modules with them in Section 7.

To present the next modules, we use some conventions to make each interface between pairs of modules unique and provide some intuitions about how values flow through them. Figure 11 shows the interfaces between a transformer module and the output of the module immediately upstream (UO) and the input of the module immediately downstream (DI). The module upstream may be a source or a transformer and the module downstream may be a sink or a transformer also. The interface upstream (UO-TI) receives requests which create an implicit stream of $\bar{x}_i$ variables and produces answers of values v_i. To show the progression of values through the modules, we write v_i' the value in a downstream interface with the same index, after some processing has occurred. Correspondingly, the variable in the request for that answer is noted $\bar{x}_i'$.

[17]The specifications given in this section follow the pull-stream protocol presented in the last section. The JavaScript implementations might have additional parameters compared to the specifications to change the order of event execution and intentionally generate incorrect behaviors to see how other modules react to them. They may also use different conventions that are more idiomatic to JavaScript or that follow the existing naming conventions of the pull-stream community.

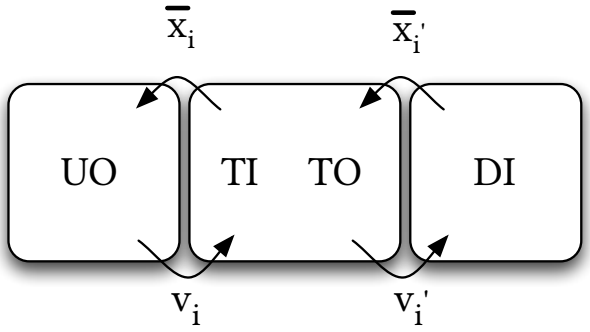


Figure 11: Abstract representation of the interfaces between a transformer and the output of the module upstream (UO-TI), which could be a source or another transformer, and the input of the module downstream (TO-DI), which could be a sink or another transformer. Each interface generates a variable stream with a unique name (ex: $\bar{x}_i$). Each new variable is implicitly added by a request coming from the interface's downstream port (ex: TI). The answers to those requests produce corresponding values that flow in the opposite direction (ex: v_i and v_i').

A *port* is used to specify the particular side of an interface where an event is initiated. When the events are implemented with function calls as in JavaScript, it represents the caller side. The destination of a request or an answer is implicit and not ambiguous because it is always the complementary port of the same interface (ex: UO for a request initiated on TI). The rules that define the behavior of the modules are given according to the ports of the abstract representation given in Figure 11. In the rest of this section, the upstream module is a source with an output UO, the middle module is a transformer with an input TI and an output TO, and the downstream module is a sink with an input DI. The three are connected in a single pipeline one immediately after the other ($UO - TI$ and $TO - DI$). We present all three modules in sequence and conclude with a discussion about completeness and correctness of specifications.

The reference source presented in Figure 12 answers requests with either a stream value (v_i), a termination marker (*done*), or an error (*err*). The rules are parameterized by the number of stream values to produce (n) and a boolean flag (err) to simulate the behavior of modules that may fail to correctly produce a value instead of normally terminating.

Parameters:
n ($n \geq 0$): number of values to produce;
err (*boolean*): terminate with a done ($err = \bot$) or an error ($err = \top$).

$$
\begin{aligned}
UO : \bar{x}_i := v_i \quad & \Leftarrow TI : ask[\bar{x}_i] \wedge \ i \leq n \\
UO : \bar{x}_i := done \quad & \Leftarrow \begin{bmatrix} TI : ask[\bar{x}_i] \wedge \ \neg err \wedge i = n + 1 \\ TI : \underline{terminate}(\bar{x}_i) \end{bmatrix} \\
UO : \bar{x}_i := err \quad & \Leftarrow TI : ask[\bar{x}_i] \wedge \ err \wedge i = n + 1
\end{aligned}
$$

Figure 12: Rules for a reference source.

The behavior of the module is defined as implication rules ($e \Leftarrow a$) that are reversed to emphasize

on the left side all the possible events initiated by the module. The antecedent a is a conjunction of posssibly multiple things: (1) events initiated downstream from the transformer input (TI); (2) boolean relations between the stream index of a particular request (i) and one of the source parameters (n); (3) boolean flags (err). In the rules, i in a $TI : request(\bar{x}_i)$ is a free variable that matches any event in the history regardless of its concrete index $(1, 2, ...)$. If the antecedent is true according to the semantics (Section 4.2) and the event e in the consequent of the implication has not happened yet ($e \notin H$), then the implementation of the source should eventually initiate e. As an example for the first rule, if the transformer input has asked for a value in the past ($TI : ask[\bar{x}_i] \in H$), there are still values to produce ($i \leq n$), and v_i has not been produced yet ($UO : \bar{x}_i := v_i \notin H$), then a value v_i should eventually be produced and assigned to $\bar{x}_i$. The other rules are straight-forward to derive from the pull-stream protocol. The rules for the source have been designed to be mutually exclusive, therefore only one rule applies at a time[18].

The organization of the rules in the figure makes it easy to visually inspect that all possible answers have rules associated with them. The same rules may also be rewritten in the other direction to verify that all cases of requests from the module immediately downstream are covered. Knowing that all requests and answers are both covered, we can be confident that the behavior of the module is fully specified according to the pull-stream protocol. The rules may be easily adapted to a different configuration. For example, if the source was connected directly to the sink, all rules with the prefix TI could be replaced with the prefix DI and would apply in the same way.

Parameters:
r ($r \geq 0$): number of *ask* requests to perform;
err (*boolean*): terminate with an abort ($err = \bot$) or with an error ($err = \top$);
w (*boolean*): wait for the previous value before terminating ($w = \top$) or not ($w = \bot$).

$$
\begin{aligned}
\underline{wait}(j) ::= & \begin{bmatrix} w \wedge TO : \bar{x}'_j := v_j \\ \neg w \wedge DI : ask[\bar{x}'_j] \end{bmatrix} \\
DI : ask[\bar{x}'_1] \Leftarrow & \; r > 0 \\
DI : ask[\bar{x}'_i] \Leftarrow & \; i \leq r \wedge TO : \bar{x}'_{i-1} := v'_{i-1} \\
DI : abort[\bar{x}'_1] \Leftarrow & \; r = 0 \wedge \neg err \\
DI : abort[\bar{x}'_i] \Leftarrow & \; i = r + 1 \wedge \neg err \wedge \underline{wait}(i-1) \\
DI : error[err, \bar{x}'_1] \Leftarrow & \; r = 0 \wedge err \\
DI : error[err, \bar{x}'_i] \Leftarrow & \; i = r + 1 \wedge err \wedge \underline{wait}(i-1)
\end{aligned}
$$

Figure 13: Rules for a reference sink.

The reference sink in Figure 13 makes at most r *ask* requests for values and an extra *terminate* request if the source has not terminated yet. The sink may make fewer requests than there are values available. If fewer values are requested than are available ($r < n$), the sink terminates *early*. Two other parameters control the termination behavior: (1) it may terminate normally ($err = \bot$) or abnormally ($err = \top$); (2) it may terminate after having received an answer ($w = \top$) or immediately after having issued an *ask* request ($w = \bot$). If the upstream module returns a terminated answer in

[18] This is not true in general: some modules may coordinate multiple concurrent streams and therefore have many rules that apply at the same time.

response to an *ask* request rather than a value, the sink stops emitting events, as expected by the pull-stream protocol.

The rules are organized similarly as for the reference source. Since the event initiated by the sink are requests, they are put on the left side of the implication rules. The stream of requests is defined inductively. The initial request depends only on the module parameters. For example, the first request will be $DI : ask[\bar{x}'_1]$ if there is at least one *ask* request to perform. Subsequent requests, such as $DI : ask[\bar{x}'_i]$, happen after an answer has been initiated from the transformer output (ex: $TO : \bar{x}'_{i-1} := v'_{i-1}$). In case the transformer module initiated a terminated answer, none of the rules apply and therefore the sink stops. Note that the rules use $'$ to indicate the stream variables ($\bar{x}'_i$) and stream values (v'_i) follow one stage of processing that happens before the sink. If the sink was connected directly to the source with no transformer in-between, all $'$ should be removed from the rules.

Parameters:
r ($r \geq 0$): number of *ask* requests to perform;
err (*boolean*): terminate with an abort ($err = \bot$) or with an error ($err = \top$).

$$
\begin{aligned}
TI : ask[\bar{x}_i] &\Leftarrow DI : ask[\bar{x}'_i] \wedge i \leq r \\
TI : abort[\bar{x}_i] &\Leftarrow \begin{bmatrix} DI : ask[\bar{x}'_i] \wedge i = r + 1 \wedge \neg err \\ DI : abort[\bar{x}'_i] \wedge \neg TI : \underline{terminate}(\bar{x}_{i-1}) \end{bmatrix} \\
TI : error[err, \bar{x}_i] &\Leftarrow \begin{bmatrix} DI : ask[\bar{x}'_i] \wedge i = r + 1 \wedge err \\ DI : error[err, \bar{x}'_i] \wedge \neg TI : \underline{terminate}(\bar{x}_{i-1}) \end{bmatrix} \\
TO : \bar{x}'_i := v'_i &\Leftarrow UO : \bar{x}_i := v_i \\
TO : \bar{x}'_i := done &\Leftarrow \begin{bmatrix} UO : \bar{x}_i := done \\ DI : \underline{terminate}(\bar{x}'_i) \wedge TI : \underline{terminate}(\bar{x}_{i-1}) \wedge UO : \bar{x}_{i-1} := done \end{bmatrix} \\
TO : \bar{x}'_i := err &\Leftarrow \begin{bmatrix} UO : \bar{x}_i := err \\ DI : \underline{terminate}(\bar{x}'_i) \wedge TI : \underline{terminate}(\bar{x}_{i-1}) \wedge UO : \bar{x}_{i-1} := err \end{bmatrix}
\end{aligned}
$$

Figure 14: Rules for a reference transformer.

The reference transformer in Figure 14 propagates requests from its output to its input and similarly transforms the values received on its input and send them on its output. The parameters are similar to those of the sink: r for the number of *ask* requests and err to control whether the termination is done normally or abnormally. It does not need a wait (w) parameter because its waiting behavior actually is the same as that of the sink downstream.

The rules presented are similar to the rules for the reference source and sink. They syntactically capture the transformation behavior by relating the non-transformed upstream stage ($\bar{x}_i, v_i$) and the transformed downstream stage ($\bar{x}'_i, v'_i$). The rules are similar to those of the sink with a subtle addition to avoid terminating upstream more than once. The second case of $TI : abort[\bar{x}_i]$ and $TI : error[err, \bar{x}_i]$ ensure that if the transformer aborted early a second abort will not be issued. The second case of $TO : \bar{x}'_i := done$ and $TO : \bar{x}'_i := err$ ensure that the termination request from downstream will receive an answer after the answer upstream has been received, even if a request had not been issued to avoid terminating twice.

Our specification assumes the downstream module behaves correctly (ex: no additional $ask[\bar{x}'_i]$ after an *abort*). If the downstream module were incorrect, it is possible that the incorrect behavior could be propagated upstream. In practice a protocol checker [28] may be used to catch those incorrect behaviors without having to complicate the transformer specification to defensively handle incorrect behavior.

6.1 Completeness and Correctness

The specification of a module is *complete* if it defines rules for all possible events that may happen externally. The specification is *correct* (follows the pull-stream protocol) if when combined with one of the reference modules on all its interfaces, any possible history generated by their combined specification at each of these interfaces follows the pull-stream protocol. To test implementations, we use a JavaScript implementation of the reference modules [29] to generate all possible sequences of events on streams of a finite size and check that the events on all interfaces follow the pull-stream protocol with a protocol checker [28]. We provide more details in the next section.

7 Evaluation of Community Modules

We tested our understanding of the protocol using the modules of the core library [13] and checked them for conformity at the same time. To do so, we generated all possible sequences of events using our references modules [29] and checked that the events generated at the interface of any two modules followed the pull-stream protocol with a checker module [28]. Our experiments are available in a GitHub repository [27]. We have found that in almost all cases the implementations correctly follow the pull-stream protocol as described in this paper, which is a testament to the quality of the core library. However, we did find one inconsistency and one incorrect behavior according to our specification.

The inconsistency was found in the way some source modules can be terminated without a callback but not others[19]. That behavior was part of our initial specification of the protocol but we since removed it after finding that many sources do not actually support it.

The incorrect behavior was found in a corner case for the *take* transformer module in which an early termination downstream triggers two abort events upstream[20]. The erroneous behavior correspond to the behavior our reference transformer in Section 6 would have if the abort condition did not check for an existing termination request upstream (i.e. if the second case of $TI : abort[\bar{x}_i]$ were simply $DI : abort[\bar{x}'_i]$). The expected behavior was not documented but following a discussion with Dominic Tarr[21], it was confirmed it was incorrect. In practice, it seems most sources and transformers handle multiple termination requests just fine so that was not a major problem for interoperability. However, it did illustrate that even seemingly simple protocols can have surprising corner cases and that an effort at formalizing is beneficial to identify them.

[19] Reported here `https://github.com/pull-stream/pull-stream/issues/101` and applicable to version 3.6.1.
[20] Reported here `https://github.com/pull-stream/pull-stream/issues/104`, same version.
[21] *Idem.*

8 Related Work

Pull-stream documentation. The community that created and maintains pull-stream modules has also produced informal documentation about their expected behavior [4, 11, 33]. Our description of expected sequences of events in the pull-stream protocol and our reference modules are consistent with that documentation and fill some gaps. To the best of our knowledge this paper is the first academic paper that describes the pattern formally and suggests a specification language for pull-stream modules.

Streaming design patterns as libraries. The pull-stream design pattern itself is similar to at least another open source JavaScript compositional library called Reducers written by Irakli Gozalishvili [21], which was independently conceived around the same time. There are some other examples in academic publications of streaming libraries for other languages. Spark Streaming exists for Scala and was used for large-scale stream processing [38]. Biboudis et al. [5] have proposed a domain-specific streaming language for Java as a library that was faster in some cases than the native Java streaming API. More generally though there seems to be little recent published work on streaming design patterns. We therefore provide a larger historical context on streaming or streaming-related languages by providing some older papers from the programming language community that are representative of ideas that are related.

Stream processing in programming languages and dataflow programming. A good overview of the earliest developments of stream processing from the 60s to the 90s has been written by Stephens [32]. It covers among others representative papers from the dataflow, functional, and logic approaches to stream programming, as well as reactive approaches and hardware design and verifications applications. From these different approaches we think the dataflow approach is closest to the pull-stream programming model. The first dataflow language is Lucid [36]. In Lucid, variables are infinite streams and variable transformations, called *filters*, are expressed as equations between variables. Filters therefore continuously compute new values based on the latest available values, similar to the pull-stream transformers. Lucid as a notation is more expressive than using pipelines of pull-stream modules: for example, it is easier in Lucid to express a complex dataflow network where let's say the output of a filter becomes in own input and is combined with multiple other input streams.

Our own experience with stream processing with dedicated language support, which led to the insights we mentioned in Section 3, came from our experience with Oz [22, 34], a multi-paradigm language built around a core language of orthogonal language features which together provide support for all the major programming paradigms. In Oz, stream programming is done with explicit streams of single-assignment dataflow variables. This makes the additional concurrency possibilities of the pull-stream protocol much easier to use in practice since the stream data structure takes care of the synchronization issues, which is not the case in JavaScript.

Another more recent survey from Johnston et al. details later development in the 90s and early 2000s on dataflow approaches, languages, visual dataflow programming [23]. Recently, the dataflow programming paradigm was revived around distributed programming with languages such as Agapia [30].

Functional reactive programming. From another perspective, functional reactive programming [37] also integrated similar streaming notions as a library rather than as programming language primitives, with the Functional Reactive Animation library and its an associated denotational semantics, as a prime example [16, 15]. One key difference compared to the dataflow approaches we mentioned

previously is in the treatment of time as a continuous quantity and the associated problem of sampling to realize animations with a discrete number of frames.

Formal specifications. Our own expertise does not lie in formal specification and our knowledge of the existing literature is limited. It is therefore likely that our event-based protocol language is similar to prior existing work. Accordingly, we do not make any claim of originality on it and recognize that better alternatives to specify pull-stream modules may exist. Nonetheless, our language definition has shown to be *sufficient* to describe both the pull-stream protocol and specify reference modules and *concise enough* to make the paper self-contained. We believe its main contribution will be to make it easier for experts in the respective fields to use this paper as a case study for their own work on formal methods and suggest better notation alternatives that could be used to specify pull-stream modules.

That being said, our event-based protocol language uses predicate calculus [17] and models time implicitly by extending a history in a discrete step for each new event added. We were inspired to formalize pull-streams around events after reading existing work on formalizing distributed systems [6, 24]. A state machine-based formalism could have been another valid alternative. Lamport's Temporal Logic of Action [25] and its associated language TLA+ [26] are a foundation to reason about concurrent programs with a strong mathematical foundation which could have been another alternative. We do not have experience in using either but the introduction of TLA+ [26] claims that it is well suited to specify discrete asynchronous systems. This suggests they would be a good formal basis for our semantics. Another possible option would be temporal logic [31, 8].

Automated testing. Our testing strategy, is similar to the property-based testing approach of Claessens and Hughes [7] in which a large number of test cases are automatically generated and some properties are checked at run-time to hold over all test cases. However, in contrast to their approach, we systematically test all cases for small finite streams rather than performing random testing. This is especially effective in our case as the behavior of pull-stream modules can often be defined inductively: testing for the base cases and a few inductive ones is usually enough to uncover most bugs. Another difference is that we do not test the module for functional correctness (i.e. that their output is correct given their input), we test them to ensure that whatever event they generate, their sequence correctly follow the pull-stream protocol. Our testing approach therefore focuses on *interoperability* between modules.

Others. Recent work in the programming language community has focused on stream processing. Vaziri et al. have shown an approach based on extending spreadsheets with stream support and formalized the core of their language extension [35].

9 Conclusion and Future Work

In this paper, we provided a formal treatment of the pull-stream design pattern that originated within the JavaScript developer community. We provided a new insight that the effectiveness of this design pattern comes from the *declarative concurrent programming model* that it uses. We then presented an event-based protocol language that is independent of the original JavaScript implementation, which we then used to formally and concisely specify the pull-stream callback protocol and reference modules that generate all possible events. This formalization ultimately led to new pull-stream module implementations in JavaScript that in turn helped automatically identify some corner-cases and possible inconsistencies in actual implementations of the most widely used modules. Our

approach therefore helps to better understand the pull-stream protocol, ensure interoperability of community modules, and concisely and precisely specify new pull-stream abstractions in papers.

We envision many direct applications to our work: (1) test more community modules for conformity; (2) implement the event-based protocol language in JavaScript and compare the execution of the specification to the implementation for consistency; (3) extend the run-time checking approach to test all possible inter-leavings of concurrent executions; (4) provide an ascii-based equivalent notation to document the behavior of community modules; (5) document other stream protocols using a similar approach.

In addition, the pull-stream protocol itself is relatively simple and could serve as a great case study for ensuring that the latest verification techniques based on type checking are expressive enough to be applicable to protocol compliance.

Acknowledgements. We would like to thank Francisco Ferreira for his tremendous help in clarifying the presentation of our event-based protocol language and providing comments on drafts of this paper.

A Normalization Rules

Input for the proof of termination with Aprove:

```
(VAR x y e e1 e2 a)
(RULES
    before(e,and(x,y)) -> and(before(e,x),before(e,y))
    before(and(x,y),e) -> and(before(x,e),before(y,e))
    before(e,or(x,y))  -> or(before(e,x),before(e,y))
    before(or(x,y),e)  -> or(before(x,e),before(y,e))
    before(e,empty)    -> e
    before(empty,e)    -> e
    before(before(e1, empty), e2) -> before(e1, e2)
    before(e1, before(empty, e2)) -> before(e1, e2)
    and(a, empty) -> a
    and(empty, a) -> a
    or(a, empty) -> a
    or(empty, a) -> a
)
```

B Concurrent Variations of the Pull-Stream Protocol

The pull-stream protocol could have used additional concurrency while keeping the same function signature for modules. We detail here the two additional cases that would have been possible but were ultimately ruled out because they incurred additional complexity in the implementation of modules to handle the concurrency while the benefits were not worth it. We present the two cases both for the *normal sequence* and the *early-termination sequence*.

B.1 Normal Sequence

There are two additional cases. The first case correspond to *concurrent* asks with *in-order* answers. The downstream module asks for multiple values, and the upstream module answers in the same order as the asks:

$$I : ask[\bar{x}_1], \quad I : ask[\bar{x}_2], \quad O : \bar{x}_1 := v_1, \quad O : \underline{terminated}(\bar{x}_2)$$

The second case corresponds to *concurrent* asks with *out-of-order* answers. The downstream module asks for values concurrently and the upstream module answers in any order. This can be used to ensure minimum latency as answers are made as fast as they are available:

$$I : ask[\bar{x}_1], \quad I : ask[\bar{x}_2], \quad O : \underline{terminated}(\bar{x}_2), \quad O : \bar{x}_1 := v_1$$

In these two additional cases, the ordering of the stream values is still captured by the order in which the *asks* were made and the downstream module can remember in which order the stream variables ($\bar{x}_1$ and $\bar{x}_2$) were created. While the additional implementation complexity is usually not worth it

in JavaScript, these possibilities are actually quite natural to use in a dataflow language such as Oz because the stream of variables is explicitly represented in memory and all the synchronization happens implicitly as the values are bound to the variables.

B.2 Early-Terminated Sequence

Similar to the normal sequence analysis, there are two additional cases to consider: the *concurrent in-order* and the *concurrent out-of-order* cases.

The early-termination sequence can already issue a *terminate* request concurrently with a single *ask* request, so it can be considered as a limited of in-order concurrency. In the *full concurrent in-order* case, more than one *ask* are initiated concurrently before the *terminate* request and therefore the *terminated* answers are also received in-order. For example:

$$I : ask[\bar{x}_1],\ I : ask[\bar{x}_2],\ I : \underline{terminate}(\bar{x}_3),\ O : \underline{terminated}(\bar{x}_1),\ O : \underline{terminated}(\bar{x}_2),\\ O : \underline{terminated}(\bar{x}_3)$$

The *concurrent out-of-order* case is similar to the *in-order* case except that answers may be received out-of-order which also applies to the termination answers. For example:

$$I : ask[\bar{x}_1],\ I : ask[\bar{x}_2],\ I : \underline{terminate}(\bar{x}_3),\ O : \underline{terminated}(\bar{x}_3),\ O : \underline{terminated}(\bar{x}_1),\\ O : \underline{terminated}(\bar{x}_2)$$

References

[1] Information technology - Syntactic metalanguage - Extended BNF. http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip, 1996.

[2] Pull-stream community modules, 2017. URL: https://pull-stream.github.io/.

[3] Automated Program Verification Environment., 2018. URL: http://aprove.informatik.rwth-aachen.de/.

[4] Pull-Stream Specification., 2018. URL: https://github.com/pull-stream/pull-stream/blob/master/docs/spec.md.

[5] Aggelos Biboudis, Nick Palladinos, George Fourtounis, and Yannis Smaragdakis. Streams à la carte: Extensible pipelines with object algebras. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 37. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[6] Manfred Broy and Max Fuchs. The Design of Distributed Systems - An Introduction to FOCUS. Technical report, 1992.

[7] Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. *Acm sigplan notices*, 46(4):53–64, 2011.

[8] Edmund M. Clarke, E Allen Emerson, and A Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[9] David Dias. The JavaScript Implementation of libp2p networking stack., 2017. URL: https://github.com/libp2p/js-libp2p.

[10] David Dias, Friedel Ziegelmayer, Juan Benet. IPFS implementation in JavaScript., 2017. URL: https://github.com/ipfs/js-ipfs.

[11] Dominic Tarr. Pull-stream. http://dominictarr.com/post/149248845122/pull-streams-pull-streams-are-a-very-simple, 2016.

[12] Dominic Tarr. Pull-Many: Combine many streams into one stream, as they come, while respecting back pressure., 2017. URL: https://github.com/pull-stream/pull-many.

[13] Dominic Tarr. Pull-stream repository, 2017. URL: https://github.com/pull-stream/pull-stream.

[14] Dominic Tarr. Secure-Scuttlebutt: A database of unforgeable append-only feeds, optimized for efficient replication for peer to peer protocols., 2017. URL: https://github.com/ssbc/secure-scuttlebutt.

[15] Conal Elliott. Functional implementations of continuous modeled animation. *Principles of Declarative Programming*, pages 284–299, 1998.

[16] Conal Elliott and Paul Hudak. Functional reactive animation. In *ACM SIGPLAN Notices*, volume 32, pages 263–273. ACM, 1997.

[17] Herbert Enderton and Herbert B Enderton. *A mathematical introduction to logic*. Academic press, 2001.

[18] Erick Lavoie. Pull-lend-stream: A refinement of the parallel map module for dynamic, unbounded, and fault-tolerant parallel processing., 2017. URL: `https://github.com/elavoie/pull-lend-stream`.

[19] Erick Lavoie. Pull-stream unspecified behaviors. `https://github.com/pull-stream/pull-stream/issues/104` and `https://github.com/pull-stream/pull-stream/issues/101`, 2018.

[20] Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, et al. Analyzing program termination and complexity automatically with AProVE. *Journal of Automated Reasoning*, 58(1):3–31, 2017.

[21] Irakli Gozalishvili. Reducers: Library for higher-order manipulation of collections., 2018. URL: `https://github.com/Gozala/reducers`.

[22] Martin Henz, Gert Smolka, and Jörg Würtz. Oz-a programming language for multi-agent systems. In *IJCAI*, pages 404–409, 1993.

[23] Wesley M Johnston, JR Hanna, and Richard J Millar. Advances in dataflow programming languages. *ACM Computing Surveys (CSUR)*, 36(1):1–34, 2004.

[24] C. Klein, B. Rumpe, and M. Broy. A stream-based mathematical model for distributed information processing systems - SysLab system model. *ArXiv e-prints*, September 2014. `arXiv:1409.7236`.

[25] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, 1994.

[26] Leslie Lamport. Specifying concurrent systems with tlaˆ+. *NATO ASI SERIES F COMPUTER AND SYSTEMS SCIENCES*, 173:183–250, 1999.

[27] Erick Lavoie. ECOOP18 Pull-Stream Experiments., 2018. URL: `https://github.com/elavoie/ecoop18-pull-stream-experiments`.

[28] Erick Lavoie. Pull-stream protocol checker., 2018. URL: `https://github.com/elavoie/pull-stream-protocol-checker`.

[29] Erick Lavoie. Pull-stream reference modules., 2018. URL: `https://github.com/elavoie/pull-stream-reference-modules`.

[30] Ciprian I Paduraru. Dataflow Programming Using AGAPIA. In *Parallel and Distributed Computing (ISPDC), 2014 IEEE 13th International Symposium on*, pages 87–94. IEEE, 2014.

[31] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE, 1977.

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

[33] Dominic Tarr. Pull-stream-examples., 2018. URL: `https://github.com/dominictarr/pull-stream-examples`.

[34] Peter Van-Roy and Seif Haridi. *Concepts, techniques, and models of computer programming*. MIT press, 2004.

[35] Mandana Vaziri, Olivier Tardieu, Rodric Rabbah, Philippe Suter, and Martin Hirzel. Stream processing with a spreadsheet. In *European Conference on Object-Oriented Programming*, pages 360–384. Springer, 2014.

[36] William W Wadge and Edward A Ashcroft. *LUCID, the dataflow programming language*, volume 303. Academic Press London, 1985.

[37] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *Acm sigplan notices*, volume 35, pages 242–252. ACM, 2000.

[38] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438. ACM, 2013.