CloudKeeper

Florian Schoppmann*

January 27, 2016

This article introduces CloudKeeper: a domain-specific language and a corresponding runtime system for data flows. While motivated by Lifecode's needs of analyzing the human genome at scale, CloudKeeper is entirely general-purpose and abstracts away tasks like data transfer, serialization, scheduling, checkpointing, and package/dependency management. For Lifecode this means, e.g., that without any source-code modifications, our genome-analysis data flow can be run purely in-memory within a single JVM as well as in a distributed fashion in the cloud.

CloudKeeper is superficially similar to academic workflow management systems like Taverna or Pegasus, though it targets software engineers instead of users. The statically typed DSL piggybacks on existing IDE support for Java, Scala, or Groovy – it also allows seamless integration of data-flow programming into any JVM-based language. The runtime system is available as just a library, and it is lightweight enough to be used as alternative to lower-level parallelization concepts such as threads, Java executor services, actor systems, futures/promises etc. CloudKeeper is highly modular and versatile: E.g., intermediate results can be kept as in-memory Java objects as well as in the file system or in a cloud-storage service. Likewise, processing of individual tasks can be as different as using an existing thread pool or by submitting a job to a distributed resource manager like Grid Engine.

Besides explaining CloudKeeper's design philosophy and architecture, this article also introduces the sophisticated data-flow-execution and checkpointing algorithms use by CloudKeeper, including formal correctness proofs.

1 Introduction

Write an introduction and present related work!

*fschoppmann@lifecodehealth.com

2 The Interpreter Component

The *CloudKeeper interpreter* component is responsible for traversing the runtime representation of a workflow (in the terminology of programming languages and compilers, this is sometimes referred to as the *optimized abstract-syntax tree*) and for recursively starting interpreters for submodules. It provides different kinds of module interpreters corresponding to the different kinds of CloudKeeper modules, such as composite modules, loop modules, or simple modules.

Most of the complexity of the interpreter component stems from parent modules. Simple modules instead are atomic entities for the interpreter, as they consist of user-defined code written in Java or some other programming language. Therefore, a simple-module interpreter simply passes the simple module on to the *executor* component, which is responsible for executing user-defined code. An executor has a great degree of freedom in doing so: For instance, it may invoke the user-defined code on a remote machine, make use of external schedulers, etc. However, the details of executors are entirely hidden from a simple-module interpreter and therefore do not need to be addressed in this section.

2.1 Formal Model of CloudKeeper Modules

In order to reason about the concepts used by the CloudKeeper interpreter component as well as to prove correctness of the algorithms described in this section, we start by introducing a formal model of what constitutes CloudKeeper modules. This is followed by additional definitions needed later, as well as a concrete example.

Modules A *CloudKeeper module* is a 6-tuple $M = (I, O, L, \text{deps}, S, C)$ consisting of the following elements.

- A set I of in-ports. An in-port is a formal input parameter.
- A set O of out-ports. An out-port is a formal output parameter.
- A set L of in-port/out-port pairs each of which defines an I/O-port. If M is not a loop module, then necessarily $L = \emptyset$.
- A function $\text{deps} : I \to 2^O$ that maps each in-port $i \in I$ to a subset of out-ports. If $o \in \text{deps}(i)$ we say that out-port o *depends* on in-port i. While it is necessary that $o \in \text{deps}(i)$ if the value passed for i is needed for computing the value of o, there is no upper-bound constraint on $\text{deps}(i)$. In other words, $\text{deps}(i) = O$ is always valid.
- A set S of submodules, where every $N \in S$ is itself a module $N = (I_N, O_N, L_N, \text{deps}_N, S_N, C_N)$. If M is not a parent module, then necessarily $S = \emptyset$.
- A set C of connections. Each connection $c \in C$ is an ordered pair of ports in the union of I, O, and all in- and out-ports of (direct) submodules of M. When M is not a parent module, then necessarily $C = \emptyset$.

The set C can be partitioned into the set of *parent-in-to-child-in* connections $C_{\uparrow}$, the set of *sibling connections* $C^{\rightarrow}$, the set of *child-out-to-parent-out* connections $C^{\downarrow}$, and the set of *short-circuit* connections $C_{\rightarrow}$. Specifically, let $c = (p, q) \in C$. Then:

i) $c \in C_{\uparrow}$ if $p \in I$ and $\exists N \in S : q \in I_N$

ii) $c \in C^{\downarrow}$ if $\exists N \in S : p \in O_N$ and $q \in O$

iii) $c \in C^{\rightarrow}$ if $\exists N, P \in S : p \in O_N$ and $q \in I_P$

iv) $c \in C_{\rightarrow}$ if $p \in I$ and $q \in O$

Graphs A *directed graph* $G = (V, E)$ consists of a set of *vertices* V and a set of *edges* $E \subseteq V^2$. A vertex $v \in V$ is called a *source vertex* is it has no incoming edges, that is, there is no $w \in V$ with $(w, v) \in E$. Likewise, v is called a *sink vertex* if it has no outgoing edges, that is, there is no $w \in V$ with $(v, w) \in E$. A *path* of length $n \geq 0$ is a sequence of vertices $(v_0, \ldots, v_n)$ so that for every $i = 1, \ldots, n$, it holds that $(v_{i-1}, v_i) \in E$. The path is called *maximal* if v_0 is a source vertex and v_n is a sink vertex.

Given a vertex $v \in V$, we say that another $w \in V$ is *reachable from* v if there is a path from v to w. Corresponding to the previous definition of a path, the reachable relation is reflexive; that is, every vertex is reachable from itself.

Dependency Graphs Let $M = (I, O, L, \text{deps}, S, C)$ be a module. Define $V := S \cup I \cup O \cup (\bigcup_{N \in S} I_N)$ as the set of all submodules, all in-port, all out-ports, and all submodules' in-ports. The *dependency graph of* M is then defined as the directed graph $G = (V, E)$, where the edge set E contains (v, w) if and only if one of the following (mutually exclusive) conditions holds:

i) (v, w) is a parent-in-to-child-in or a short-circuit connection. Formally, $(v, w) \in C_{\uparrow} \cup C_{\rightarrow}$.

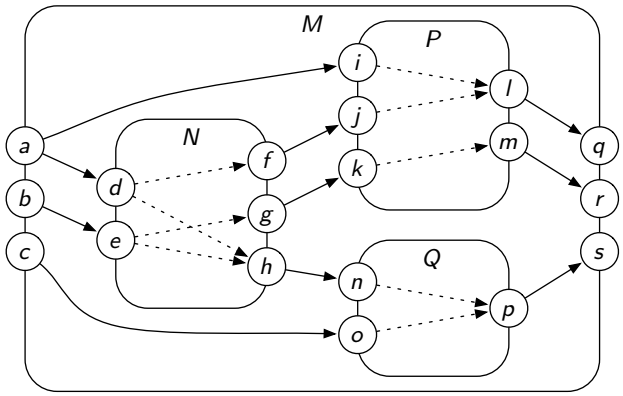
ii) v is the in-port of some submodule, and this submodule has an out-port that depends on v and that has a connection to w. Formally, $\exists N \in S : v \in I_N$ and $\exists o \in \text{deps}_N(v) : (o, w) \in C^{\rightarrow} \cup C^{\downarrow}$.

iii) v is a submodule, w is an out-port or a submodule's in-port, and v has an out-port that has a connection to w. Formally, $\exists N \in S : v = N$ and $\exists o \in O_N : (o, w) \in C^{\rightarrow} \cup C^{\downarrow}$.

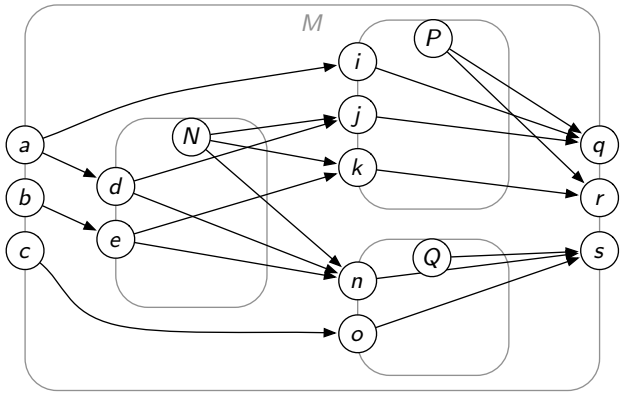
Let $R := \bigcup_{N \in S} O_N$ be the set of all submodules' out-ports. We define $\text{outports}(v, w) : E \to 2^R$ as the set of all out-ports that *witness* the connection between v and w in the dependency graph (in case of conditions (ii) and (iii)). Formally,

$$\text{outports}(v, w) := \begin{cases} \emptyset & \text{if } v \in I \\ \{o \in \text{deps}_N(v) \mid (o, w) \in C^{\rightarrow} \cup C^{\downarrow}\} & \text{if } v \in I_N \text{ for some } N \in S \\ \{o \in O_N \mid (o, w) \in C^{\rightarrow} \cup C^{\downarrow}\} & \text{if } v = N \in S \end{cases}$$

In the remainder of this section, we will only be interested in the *pruned* dependency graph with respect to a set of requested out-ports $O^* \subseteq O$. The pruned graph only contains those edges from which O^* is reachable. That is, all vertices that did not have a path to O^* become isolated vertices in the pruned graph.



(a) Parent module with three submodules



(b) Corresponding dependency graph

Figure 1: Example parent module and its dependency graph

Example 2.1. Let $M = (I, O, L, \text{deps}, S, C)$ be the CloudKeeper parent module shown in Figure 1a. The following list illustrates the previous definitions:

- Set of in-ports $I = \{a, b, c\}$, set of out-ports $O = \{q, r, s\}$, set of in-port/out-port pairs $L = \emptyset$
- Since M is a parent module, function deps could be computed from the other elements in M. There is some freedom here: Strictly speaking, computing a value for out-port q requires only a value for in-port a. That is, $q \in \text{deps}(a)$ needs to hold necessarily,

but it is optional that $q \in \text{deps}(b)$ or $q \in \text{deps}(c)$. We will see later that keeping sets deps minmal is most efficient, but it requires more preprocessing.

The dashed lines indicate functions deps_N, deps_P and deps_Q. For instance, $\text{deps}_N(d) = \{f, h\}$, $\text{deps}_N(e) = \{g, h\}$, $\text{deps}_P(i) = \{l\}$, etc.

- Set of submodules $S = \{N, P, Q\}$
- Set of parent-in-to-child-in connections $C_{\uparrow} = \{(a, i), (a, d), (b, e), (c, o)\}$
- Set of sibling connections $C^{\rightarrow} = \{(f, j), (g, k), (h, n)\}$
- Set of child-out-to-parent-out connections $C^{\downarrow} = \{(l, q), (m, r)\}, (p, s)\}$
- Set of short-circuit connections $C_{\rightarrow} = \emptyset$

The corresponding dependency graph is shown in Figure 1b. Witnesses between dependency-graph vertices are, e.g., $\text{outports}(a, i) = \emptyset$, $\text{outports}(d, j) = \{f\}$, $\text{outports}(N, j) = \{f\}$. $\Diamond$

2.2 The Basic Interpreter Protocol

The CloudKeeper interpreter component provides module interpreters for the different kinds of modules, such as composite, loop, or simple modules. Despite idiosyncrasies that will be explained in this and later sections, all module interpreters adhere to the same basic protocol: In particular, module interpreters are always invoked asynchronously, and they communicate with the entity invoking them through a very simple message interface. Specifically, a module-interpreter invocation:

- Receives *in-port-has-value* messages from its invoker whenever one of its in-ports receives a value.
- Sends *out-port-has-value* messages to its invoker whenever it has computed the value of an out-port.
- Receives a message during initialization that consists of two sets I^* and O^*, where I^* contains the in-ports for which in-port-has-value messages will be received, and O^* contains the out-ports for that out-port-has-value messages have to be sent.

Whenever unambiguous or irrelevant, we will simply write module interpreter instead of module-interpreter invocation in the following.

CloudKeeper module interpreters do not differentiate between starting and resuming. Instead, when a module interpretation is started, it is always assumed that ports may have previously computed values. For efficiency, as many as possible of these values should be reused. Yet, not all values *can* always be salvaged, because consistency requires that recomputing the value of a port also triggers reevaluation of all downstream ports.

2.3 Parent-Module Interpreters

The central piece of the interpreter component is the parent-module interpreter. It traverses a single parent module, such as a composite or a loop module, and is responsible for recursively starting interpreters for its submodules.

Consistent Checkpoints When a parent-module interpreter starts, it uses the sets I^* and O^* to compute a set of vertices A in the dependency graph that constitute a *consistent checkpoint*. Only then the traversal of the module will be started/resumed from this checkpoint. Specifically, consistent checkpoints used by CloudKeeper are characterized by the following properties:

i) Each vertex in A either represents a port that has a value or it represents a submodule.

ii) Resuming interpretation from all vertices in A and from all in-port vertices in I^* (once they receive a value) eventually leads to the computation of all required out-ports O^*. Formally, every maximal path ending in O^* contains a vertex from $A \cup I^*$.

iii) No port will receive a new value more than once. Formally, every path that ends in O^* contains at most one vertex from $A \cup I^*$.

iv) The number of reused values is maximal under the given constraints. Formally, the set of all vertices reachable from any other set B satisfying the previous constraints is a superset of the set of all vertices reachable from A.

The following Definition 2.2 formalizes and summarizes properties (i)–(iii). It will be established in the next Section 2.4 that the algorithm used by CloudKeeper even computes *optimal* consistent checkpoints in the sense of property (iv).

Definition 2.2. Let $M = (I, O, L, \text{deps}, S, C)$ be a parent module, $I^* \subseteq I$ be the set of recomputed in-ports, $O^* \subseteq O$ be the set of requested out-ports, and let $G = (V, E)$ be the pruned dependency graph. Let $H \subseteq V \setminus S$ be the set of ports that already have a value. A *consistent checkpoint* $A \subseteq H \cup S \setminus I^*$ is a set of vertices so that every maximal path ending in O^* contains exactly one vertex from $A \cup I^*$. $\Diamond$

Starting module interpreters Once a consistent checkpoint A has been computed, the parent-module interpreter proceeds as follows:

- It transmits the values from all in-ports contained in A to all connection targets from which O^* is reachable,
- it invokes a child interpreter for every submodule that has an in-port contained in A,
- it invokes a child interpreter for every submodule contained in A that does not have any in-ports, and

- it sends out-port-has-value messages to its invoker for all out-ports contained in A.

As discussed above, for each child-interpreter invocation the set of recomputed in-ports and the set of requested out-ports is needed. These two sets can be inferred from the consistent checkpoint A and the pruned dependency graph $G = (V, E)$. When invoking a child interpreter for submodule $N \in S$:

- The set of recomputed in-ports are those vertices in I_N that are reachable from A (as they will receive a value later), but not itself in A (as those already have a value).
- The set of requested out-ports are those vertices in O_N that are also in

$$W := \bigcup_{\substack{(v,w)\in E \\ v \text{ reachable from } A\cup I^*}} \text{outports}(v, w)\,. \tag{2.3}$$

Finishing a parent-module interpretation After every transmission of a value, one of the following actions takes place:

- When a submodule in-port receives a value, the child interpreter is invoked if it has not been invoked before. An in-port-has-value message is sent to the child interpreter.
- When an out-port receives a value, an out-port-has-value message is sent to the invoker of the current interpreter.

Whenever the parent-module interpreter receives an out-port-has-value message from one of its child interpreters, it transmits the value from that submodule's out-port to all connection targets from which O^* is reachable. This asynchronous sequence of events continues until all requested out-ports have a value.

2.4 Computing Consistent Checkpoints

While the previous section gave a definition for (and used) consistent checkpoints, we still need to verify that the definition is sound – that is, consistent checkpoints really exist – and that they can be computed efficiently. Both aspects are addressed by Algorithm `ComputeResumeState`. Its basic idea is to perform a modified breadth-first-search in order to classify all vertices of the dependency graph into one of three categories:

Ready These vertices are exactly those contained in the consistent checkpoint. That is, these vertices do not need further input, and interpretation can resume immediately from these vertices. It is guaranteed that they will never receive a new value while the current module is interpreted.

Recompute These are the vertices that are reachable from I^* or a "Ready" vertex. They are guaranteed to receive a new value while the current module is interpreted.

Irrelevant These are all other vertices, which are not needed during the current module interpretation. It is guaranteed that they will not receive a new value while the current module is interpreted.

The algorithm also computes the set W from (2.3), which contains all submodules' outports that witness connections between *ready* or *recompute* vertices to another vertex with state *recompute*. As stated previously, these are needed for recursive invocations of submodule interpreters.

The Formal Algorithm Apart from the inputs mentioned before, Algorithm `ComputeResumeState` also takes as input the *kind* of queue to be used; that is, a data structure with the following operations:

enqueue(x) Adds an element x to the queue.

dequeue Returns an element x from the queue, provided that dequeue has returned x fewer times than enqueue(x) has been called.

isEmpty Returns whether the queue is empty.

contains(x) Returns whether x is contained in the queue. This must be *true* if after the last enqueue(x), no dequeue operation has returned x. And this must return *false* if dequeue has returned x at least as often as enqueue(x) has been called. Given a queue Q, we will use the shorthand notation "$x \in Q$".

Note that this specification can be met by different kinds of queues: For instance, one kind of queue may contain the same element more than once, while another could employ set semantics. Likewise, the order in which elements are dequeued could just be the insertion order, or it could be determined by (implicit) element priorities.

Algorithm `ComputeResumeState`(M, H, I^*, O^*)
Input: parent module $M = (I, O, L, \text{deps}, S, C)$,
set $H \subseteq V \setminus S$ of ports that already have a value,
set $I^* \subseteq I$ of recomputed in-ports,
non-empty set $O^* \subseteq O$ of requested out-ports,
kind of queue
Output: map $state : V \to \{irrelevant, ready, recompute\}$,
set $W = \bigcup_{(v,w) \in E \mid state[v] \neq irrelevant} \text{outports}(v, w) \subseteq R$
Require: for all $i \in I$ with $\text{deps}(i) \cap O^* \neq \emptyset : i \in H \cup I^*$

1: Construct dependency graph $G = (V, E)$, pruned w.r.t. O^*
2: $Q \leftarrow$ new instance of the given kind of queue
3: **for** $v \in V \setminus (I^* \cup O^*)$ **do**
4: $\quad state[v] \leftarrow irrelevant$
5: **for** $i \in I^*$ **do**

```
 6:     state[i] ← recompute
 7:     enqueue i to Q
 8: for o ∈ O* do
 9:     state[o] ← ready
10:     enqueue o to Q
11: W ← ∅
12: while Q not empty do
13:     v ← dequeue from Q
14:     if state[v] = ready and v ∉ H ∪ S then
15:         state[v] ← recompute
16:     if state[v] = recompute then
17:         for w ∈ V with (w, v) ∈ E do
18:             if state[w] = irrelevant then
19:                 state[w] ← ready
20:                 enqueue w to Q
21:     for w ∈ V with (v, w) ∈ E do
22:         W ← W ∪ outports(v, w)
23:         if state[w] ≠ recompute then
24:             state[w] ← recompute
25:             enqueue w to Q
```

Example 2.4. Figure 2 shows the result of running Algorithm `ComputeResumeState` on the module from Example 2.1. The set H consists of all vertices with a gray background. The set I^* of recomputed in-ports is $\{a\}$, and the set O^* of requested out-ports is $\{r, s\}$. Note that the pruned dependency graph only contains edges that are on a path to one of the requested out-ports.

The result is indicated as follows: If a vertex $v \in V$ is shown with a thick stroke, then $state[v] = ready$, if it is shown with a dashed outline, then $state[v] = recompute$, and otherwise $state[v] = irrelevant$. ◊

Termination and Runtime Computing the pruned dependency graph in line 1 can be done using, e.g., a breadth-first-search starting from the vertices in O^*. The initialization steps in lines 2 to 11 perform $|V|$ plus constantly many assignments and at most $|V|$ enqueuing operations. In the following, we verify termination and running time of the while-loop that starts in line 12. Every vertex $v \in V$ can be added to the queue only if $state[v] = ready$ (lines 10 and 20) or if $state[v] = recompute$ (lines 7 and 25). Moreover, each enqueuing operation is contingent on $state[v]$ having been modified in the immediately preceding line. Since the state of a vertex can only change from *irrelevant* to *ready* (lines 9 and 19) or from *irrelevant* or *ready* to *recompute* (lines 6 and 24), this implies that every $v \in V$ can be enqueued at most twice – in which case it would first be enqueued in state

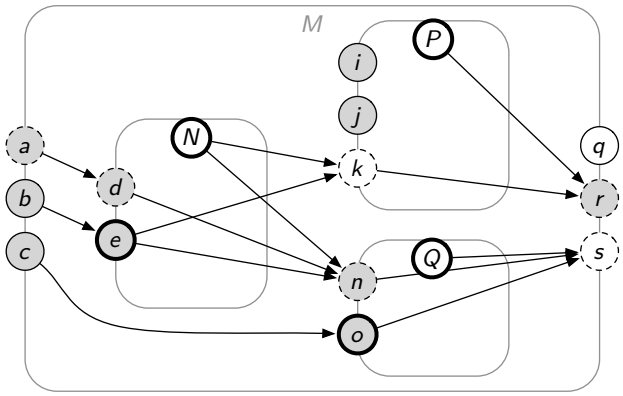


Figure 2: Result of running Algorithm `ComputeResumeState`

ready (line 10 or 20) and then again during some later iteration in state *recompute* (line 25). Since one vertex is dequeued during every iteration of the while-loop (line 13), the queue will necessarily become empty during some iteration, at which point the algorithm will terminate. Specifically, the previous observation guarantees that there will be at most $2|V|$ iterations of the while-loop.

Correctness We need to verify that, upon termination of Algorithm `ComputeResumeState`, the set $A := \{v \in V \mid state[v] = ready\}$ is a consistent checkpoint. This will proceed in a series of steps. First, we verify in Lemma 2.5 that A contains no "forbidden" vertices. Then, we show in Lemmas 2.6 and 2.8 and Corollary 2.9 that every maximal path ending in a required-out-port vertex contains exactly one vertex from A or exactly one recomputed-in-port vertex, but not both.

We also need to verify that Algorithm `ComputeResumeState` correctly computes set W (containing all submodules' out-ports that must receive a new value). Lemma 2.10 establishes that W is indeed as defined in (2.3).

Lemma 2.5. *Upon termination of algorithm* `ComputeResumeState`*, it holds for every* $v \in V$ *with* $state[v] = ready$ *that* $v \in H \cup S \setminus I^*$.

Proof. Let $v \in V$ with $state[v] = ready$. Setting $state[v]$ to *ready* can only have happened in lines 9 or 19. In either case, v was enqueued to Q in the immediately succeeding line. Since the algorithm terminated, v must have been dequeued in line 13 in some subsequent iteration i of the while-loop. The if-condition in the following line 14 was not satisfied, because otherwise $state[v]$ would have been updated to *recompute* (and would have remained *recompute* until the end of the algorithm). The first part of the if-condition, i.e., $state[v] = ready$, was satisfied in iteration i, so it must have been the second part that was not satisfied. That is, $v \in H \cup S$. Moreover, $v \notin I^*$ because otherwise $state[v]$ would have been set to *recompute* already in line 6. □

Lemma 2.6. *Let $(v_0, \dots, v_m)$ be a path in the dependency graph. Upon termination of Algorithm* `ComputeResumeState`*, the following equivalence holds for every $j \in \{1, \dots, m\}$:*

$$state[v_{j-1}] \neq irrelevant \iff state[v_j] = recompute\,.$$

Proof. Let n be the number of iterations of the while-loop before algorithm `ComputeResumeState` terminates. Denote by $Q^{(0)}$ the set of vertices contained in queue Q before entering the while-loop in line 12. Similarly, let $state^{(0)}$ be the value of *state* at this point in the algorithm. For $i = 1, \dots, n$, denote by $Q^{(i)}$ and $state^{(i)}$ the contents of Q and *state*, respectively, at the end of iteration i of the while-loop. Obviously, $Q^{(0)} = I^* \cup O^*$ and $Q^{(n)} = \emptyset$.

We will show by induction over $i = 0, 1, \dots, n$ that the following invariant holds before and after each loop iteration.

> For every path $(v_0, \dots, v_m)$ in the dependency graph and every $j = 1, \dots, m$, the following implications hold:
>
> i) $state^{(i)}[v_{j-1}] \neq irrelevant$ and $state^{(i)}[v_j] \neq recompute \implies v_{j-1} \in Q^{(i)}$
>
> ii) $state^{(i)}[v_{j-1}] = irrelevant$ and $state^{(i)}[v_j] = recompute \implies v_j \in Q^{(i)}$

If this invariant holds after iteration n, then this completes the proof because $Q^{(n)} = \emptyset$.

We first verify the base case $i = 0$. Let $(v_0, \dots, v_m)$ be a path and $j \in \{1, \dots, m\}$. To see (i), note that $state^{(0)}[v_{j-1}] \neq irrelevant$ implies $v_{j-1} \in I^* \cup O^*$. However, v_{j-1} is not a sink vertex, so $v_{j-1} \notin O^*$, but instead $v_{j-1} \in I^*$. Due to line 7, this implies $v_{j-1} \in Q^{(0)}$. Implication (ii) holds vacuously because the assumption is never satisfied: To see this, note that $state^{(0)}[v_j] = recompute$ implies $v_j \in I^*$, meaning that v_j is a source vertex – a contradiction.

Before continuing with the inductive step, we first need a small technical observation:

Claim 2.7. Let $v \in V$, $i \in \{1, \dots, n\}$, $v \notin Q^{(i-1)}$, and $v \notin Q^{(i)}$. Then $state^{(i)}[v] = state^{(i-1)}[v]$.

Proof (of Claim 2.7). Since $v \notin Q^{(i-1)}$, it holds that v was not dequeued from Q in iteration i, and therefore line 15 cannot have modified $state[v]$. Moreover, since v was not enqueued to Q in iteration i, lines 19 and 24 cannot have modified $state[v_j]$, either. In order to see this, note that in each case the immediately following line would have enqueued v. Consequently, $state^{(i)}[v] = state^{(i-1)}[v]$. ■

Now suppose that the invariant holds for an arbitrary iteration $i - 1 \in \{0, 1, \dots\}$. We show that the invariant also holds at the end of iteration i (which proves the inductive step $i - 1 \to i$). Let $(v_0, \dots, v_m)$ be an arbitrary path, and let $j \in \{1, \dots, m\}$.

i) By way of contradiction, assume that $v_{j-1} \notin Q^{(i)}$. If $v_{j-1} \in Q^{(i-1)}$, then v_{j-1} was dequeued in iteration i, and therefore $state^{(i)}[v_j] = recompute$ due to line 24, because $(v_{j-1}, v_j) \in E$. On the other hand, if $v_{j-1} \notin Q^{(i-1)}$, then at least one of the following holds because of the induction hypothesis (that is, because of the invariant after iteration $i - 1$):

- $state^{(i-1)}[v_{j-1}] = irrelevant$

  Since $v_{j-1} \notin Q^{(i)}$ and $v_{j-1} \notin Q^{(i-1)}$, Claim 2.7 implies that also $state^{(i)}[v_{j-1}] = irrelevant$.

- $state^{(i-1)}[v_j] = recompute$

  In this case, also $state^{(i)}[v_j] = recompute$ because Algorithm `ComputeResumeState` never changes the *state* of a vertex once it has been set to *recompute*.

Hence, there is a contradiction to the assumption of implication (i), as needed.

ii) By way of contradiction, assume that $v_j \notin Q^{(i)}$. If $v_j \in Q^{(i-1)}$, then v_j was dequeued in iteration i. Therefore, either $state^{(i)}[v_j] \neq recompute$ or $state[v_{j-1}]$ would have been set to *ready* in line 19, and thus $state^{(i)}[v_{j-1}] \neq irrelevant$, because $(v_{j-1}, v_j) \in E$. On the other hand, if $v_j \notin Q^{(i-1)}$, then at least one of the following holds because of the induction hypothesis:

- $state^{(i-1)}[v_{j-1}] \neq irrelevant$

  In this case, also $state^{(i)}[v_{j-1}] \neq irrelevant$ because Algorithm `ComputeResumeState` never changes the *state* of a vertex back to *irrelevant*.

- $state^{(i-1)}[v_j] \neq recompute$

  Since $v_j \notin Q^{(i)}$ and $v_j \notin Q^{(i-1)}$, Claim 2.7 implies that also $state^{(i)}[v_{j-1}] \neq recompute$.

Hence, there is a contradiction to the assumption of implication (ii), as needed. □

Lemma 2.8. *Upon termination of Algorithm `ComputeResumeState`, it holds for every $v \in V$ with $state[v] = recompute$ that v is reachable from a vertex w with $state[w] = ready$ or $w \in I^*$.*

Proof. Let $v \in V$ with $state[v] = recompute$, and let v_0 be a source vertex that v is reachable from; that is, $v_0 \in I \cup S$. (Note that such a vertex v_0 always exists.) Since the "reachable" relation is transitive, it is sufficient to show the claim for v_0.

If $state[v_0] \neq recompute$, then Lemma 2.6, together with a simple inductive argument, guarantees existence of a vertex w with $state[w] = ready$ such that v is reachable from w. Therefore suppose $state[v_0] = recompute$. One of the following must hold:

- $v_0 \in O^*$ (if $state[v_0]$ was assigned in line 6),
- $v_0 \notin H \cup S$ (if assigned in line 15), or
- $\exists w \in V$ with $(w, v_0) \in E$ (if assigned in line 24).

Of these three cases, only $v_0 \notin H \cup S$ is compatible with the previous observation that $v_0 \in I \cup S$. By construction of the dependency graph, and since $state[v_0] \neq irrelevant$, some vertex in O^* is reachable from v_0, implying $\text{deps}(v_0) \cap O^* \neq \emptyset$. The precondition of Algorithm `ComputeResumeState` implies that $v_0 \in H \cup I^*$. Put together, we get $v_0 \in I^*$.□

Corollary 2.9. *Upon termination of Algorithm* `ComputeResumeState`*, every maximal path ending in O^* contains exactly one vertex v with $state[v] = ready$ or $v \in I^*$.*

Proof. Let $(v_0, \dots, v_n)$ be a maximal path with $v_n \in O^*$. Existence of a $j \in \{0, \dots, n\}$ with $state[v_j] = ready$ or with $v_j \in I^*$ is established by Lemma 2.8 and the fact that for all $o \in O^*$ it holds that $state[o] \neq irrelevant$.

In order to show uniqueness, note that Lemma 2.6 implies there is at most one $j \in \{0, \dots, n\}$ with $state[v_j] = ready$. In that case, $v_j \notin I^*$ because of Lemma 2.5. By construction of the dependency graph, it moreover holds that $v_1, \dots, v_n \notin I^*$. This completes the proof. □

Lemma 2.10. *Upon termination of Algorithm* `ComputeResumeState`*, let $A := \{v \in V \mid state[v] = ready\}$ be the computed checkpoint. It holds that*

$$W = \bigcup_{\substack{(v,w)\in E \\ v \text{ reachable from } A\cup I^*}} \text{outports}(v, w)\,.$$

Proof. Due to Lemmas 2.6 and 2.8, we know that a vertex $v \in V$ is reachable from $A \cup I^*$ if and only if $state[v] \neq irrelevant$.

We first show "$\subseteq$". Let $o \in W$. Obviously, o was added to W in line 22, for some $v, w \in V$ with $o \in \text{outports}(v, w)$. At the beginning of that while-loop iteration, v was necessarily dequeued in line 13. This implies that $state[v] \neq irrelevant$ at that time. Since $state[v]$ can never change back to *irrelevant*, it also means that $state[v] \neq irrelevant$ after Algorithm `ComputeResumeState` terminated.

In order to show "$\supseteq$", let $(v, w) \in E$ with $state[v] \neq irrelevant$, meaning that $state[v]$ was updated at least once either in line 6, 9, 19, or 24. In each case, the subsequent line added v to the queue. Since Algorithm `ComputeResumeState` terminated, v was dequeued later, implying that $\text{outports}(v, w) \subseteq W$ due to line 22. □

Optimality So far, we have verified that Algorithm `ComputeResumeState` correctly computes a consistent checkpoint A. In the following, we show that A is moreover optimal in the following sense: If B is any other consistent checkpoint, then the set of vertices reachable from B is a superset of the set of vertices reachable from A.

Lemma 2.11. *Let B be a consistent checkpoint. Upon termination of algorithm* `ComputeResumeState`*, it holds for all $v \in V$ with $state[v] \neq irrelevant$ that v is reachable from $B \cup I^*$.*

Proof. Using the same notation as in the proof of Lemma 2.6, we show that the following invariant holds before and after every iteration of the while-loop. For every $v \in V$:

i) $state^{(i)}[v] \neq irrelevant \implies v$ reachable from $B \cup I^*$ and

ii) $state^{(i)}[v] = recompute \implies v \notin B$.

We first verify the base case $i = 0$ (that is, prove the invariant before the first iteration). Let $v \in V$. To see (i), note that $\mathit{state}^{(0)}[v] \neq \mathit{irrelevant}$ implies $v \in I^* \cup O^*$. Since B is a consistent checkpoint, all vertices in $I^* \cup O^*$ must be reachable from $B \cup I^*$. To see (ii), note that $\mathit{state}^{(0)}[v] = \mathit{recompute}$ implies $v \in I^*$. Again by definition of a consistent checkpoint, we know $I^* \cap B = \emptyset$.

For the inductive step $i - 1 \to i$, it is sufficient to consider any arbitrary w with $\mathit{state}^{(i-1)}[w] = \mathit{irrelevant}$ and $\mathit{state}^{(i)}[w] \neq \mathit{irrelevant}$. Let v be the vertex dequeued in line 13 of iteration i.

There are two cases. If the assignment to $\mathit{state}[w]$ happened in line 19, then v is reachable from $B \cup I^*$ and $v \notin B$, both by the induction hypothesis. Moreover, $v \notin I^*$ because v is not a source vertex. By way of contradiction, suppose now that w is not reachable from $B \cup I^*$. Then we can find a maximal path containing w and v that does not contain a vertex from $B \cup I^*$; a contradiction to the assumption that B is a consistent checkpoint. This proves (i). Since $\mathit{state}^{(i)}[w] = \mathit{ready}$, property (ii) holds vacuously.

If the assignment happened in line 24, then likewise w is reachable from $B \cup I^*$, because $(v, w) \in E$ and v is reachable from $B \cup I^*$ by the induction hypothesis. To see (ii), note that since v is reachable from $B \cup I^*$, it must hold that $w \notin B$. Otherwise, there would be a path through v and w that contains more than one vertex from $B \cup I^*$. This completes the proof. □

Implementation Set H is the set of all ports that already have a value. In practice, this set is relatively expensive to compute because doing so may involve accessing secondary storage (like the file system). It is therefore desirable to use H only lazily, in particular because the only operation needed is "$\notin$" in line 14. It is additionally desirable to allow processing more than one such operation at a single time. Hence, the operation should not block, and the question is instead if one may asynchronously check $v \in H$ and not dequeue v while the result is unknown. This would be possible if the order of dequeuing in line 13 was arbitrary, and we can simply dequeue another vertex $w \neq v$ while waiting for v. The following Lemma 2.12 shows this is indeed the case.

Lemma 2.12. *The output of Algorithm* `ComputeResumeState` *does not depend on the kind of queue used.*

Proof. Let M, H, I^*, O^* be arbitrary valid arguments for Algorithm `ComputeResumeState`. Define $\mathit{state}(\phi)$ as the output of Algorithm `ComputeResumeState` with the previous arguments and using a queue of kind ϕ. Let $A(\phi) := \{v \in V \mid \mathit{state}(\phi)[v] = \mathit{ready}\}$ and let $X(\phi) := \{v \in V \mid v \text{ reachable from } A(\phi) \cup I^*\}$.

Lemma 2.11 shows that for any two ϕ, ϕ', it holds that $X(\phi) \subseteq X(\phi')$. Consequently, $X(\phi)$ is constant in ϕ. Let Y denote this constant. We will show that not just $X(\phi)$ is

constant in ϕ, but also $state(\phi)$ is constant in ϕ. To this end, define $state'$ as follows:

$$state'[v] := \begin{cases} recompute & \text{if } v \in I^* \text{ or there is } w \in Y \text{ with } (w,v) \in E \\ ready & \text{if } v \in Y \text{ otherwise} \\ irrelevant & \text{otherwise.} \end{cases}$$

Now let ϕ be an arbitrary kind of queue. We will show that $state(\phi) = state'$. Let $v \in V$; there are three cases:

- Suppose $state'[v] = recompute$. It holds that $v \in I^*$ or there is a $w \in V$ with w reachable from $A(\phi) \cup I^*$ and $(w,v) \in E$. By Lemma 2.6, this implies $state(\phi)[v] = recompute$.
- Suppose $state'[v] = ready$. By definition of $state'$, it holds that $v \notin I^*$ and that there is no $w \in X$ with $(w,v) \in E$. That is, there is no $w \in V$ with $(w,v) \in E$ such that w is reachable from $A(\phi) \cup I^*$. However, $v \in Y = X(\phi)$, i.e., v is reachable from $A(\phi) \cup I^*$. This implies $v \in A(\phi) \cup I^*$. Put together, $v \in A(\phi)$, which is equivalent to $state'[v] = ready$.
- Suppose $state'[v] = irrelevant$. In this case, $v \notin Y$, i.e., v is not reachable from $A(\phi) \cup I^*$. By definition, $v \notin A(\phi)$ implies $state(\phi)[v] \neq ready$. Moreover, it follows from Lemma 2.8 that $state(\phi)[v] \neq irrelevant$. Hence, $state(\phi)[v] = irrelevant$.

Since $state(\phi) = state'$, and due to Lemma 2.10, the set W computed by Algorithm `ComputeResumeState` is likewise independent of ϕ. This completes the proof. □