Improving Workflow Fault Tolerance through Provenance-based Recovery

Sven Köhler, Sean Riddle, Daniel Zinn,
Timothy McPhillips, and Bertram Ludäscher

University of California, Davis

Abstract. Scientific workflow systems frequently are used to execute a variety of long-running computational pipelines prone to premature termination due to network failures, server outages, and other faults. Researchers have presented approaches for providing fault tolerance for portions of specific workflows, but no solution handles faults that terminate the workflow engine itself when executing a mix of stateless and stateful workflow components. Here we present a general framework for efficiently resuming workflow execution using information commonly captured by workflow systems to record data provenance. Our approach facilitates fast workflow *replay* using only such commonly recorded provenance data. We also propose a *checkpoint* extension to standard provenance models to significantly reduce the computation needed to reset the workflow to a consistent state, thus resulting in much shorter re-execution times. Our work generalizes the rescue-DAG approach used by DAGMan to richer workflow models that may contain stateless and stateful multi-invocation actors as well as workflow loops.

1 Introduction

Scientific workflow systems are increasingly used to perform scientific data analyses [1,2,3]. Often via a graphical user interface, scientists can compose, easily modify, and repeatedly run *workflows* over different input data. Besides automating program execution and data movement, scientific workflow systems strive to provide mechanisms for fault tolerance during workflow execution. There have been approaches that re-execute individual workflow components after a fault [4]. However, little research has been done on how to handle failures at the level of the workflow itself, e.g., when a faulty actor or a power failure takes down the workflow engine itself. Circumstances that lead to (involuntary) workflow failures—for example software errors, power outages or hardware failures—are common in large supercomputer environments. Also, a running workflow might be aborted voluntarily so that it can be migrated to another location, e.g., in case of unexpected system maintenance.

Since typical scientific workflows often contain compute- and data-intensive steps, a simple "restart-from-scratch" strategy to recover a crashed workflow is impractical. In this work, we develop two strategies (namely *replay* and *checkpoint*) that allow workflows to be resumed while mostly avoiding redundant

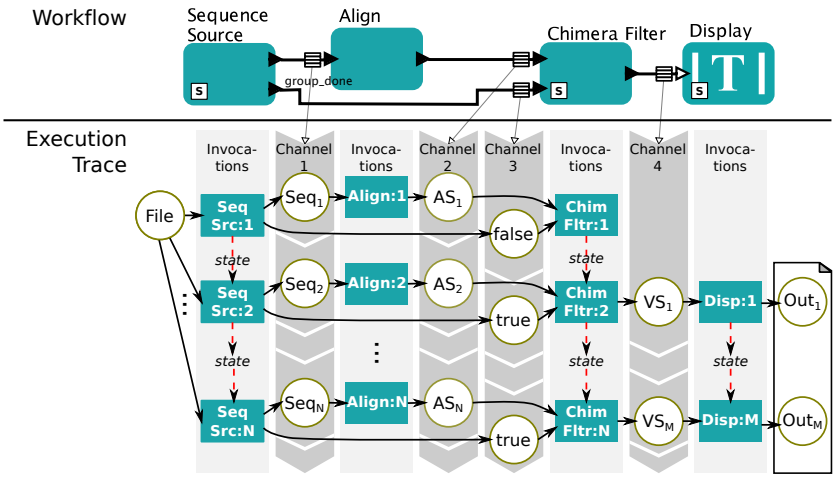

Fig. 1. Example workflow with stateful actors. To recover the workflow execution after a fault, unconsumed tokens inside workflow channels and internal states of all actors except the stateless `Align` have to be restored.

re-execution of work performed prior to the fault. The necessary book-keeping information to allow these optimizations is extracted from provenance information that scientific workflow systems often already record for data lineage reasons, allowing our approach to be deployed with minimal additional runtime overhead.

Workflows are typically modeled as dataflow networks. Computational entities (*actors*) perform scientific data analysis steps. These actors consume or produce data items (*tokens*) that are sent between actors over uni-directional FIFO queues (*channels*). In general, output tokens are created in response to input tokens. One round of consuming input tokens and producing output tokens is referred to as an actor *invocation*. For *stateful actors*, the values of tokens output during an invocation may depend on tokens received during previous invocations. The execution and data management semantics are defined by the *model of computation* (MoC).

Commonly used models for provenance are the Read/Write model [5], and the Open Provenance Model (OPM) [6]. In both provenance models, events are recorded when actors consume tokens (`read` or `used_by` events) and produce tokens (`write` or `generated_by` events). Thus, the stored provenance data effectively persists the tokens that have been flowing across workflow channels. We show how this data can be used to efficiently recover faulty workflow executions.

Example. Consider the small scientific pipeline shown in Fig. 1, which carries out two tasks automated by the WATERS workflow described in [1]. As in the full implementation of WATERS, streaming data and stateful multi-invocation actors make an efficient recovery process non-trivial.

The actor `SequenceSource` reads DNA sequences from a text file, emitting one DNA sequence token via the top-right port per invocation. The total num-

ber of invocations of `SequenceSource` is determined by the contents of the input file. On the `group_done` port, it outputs a 'true' token when the sequence output is the last of a predefined group of sequences, and 'false' otherwise. `Align` consumes one DNA sequence token per invocation, aligns it to a reference model, and outputs the aligned sequence. The `ChimeraFilter` actor receives the individually aligned sequences from `Align` and the information about grouping from the `SequenceSource`. In contrast to `Align`, `ChimeraFilter` accumulates input sequences, one sequence per invocation, without producing any output tokens until the last sequence of each group arrives. `ChimeraFilter` then checks the entire group for chimeras (spurious sequences often introduced during biochemical amplification of DNA), outputs the acceptable sequences, and clears its accumulated list of sequences.

All actors but `Align` are stateful across invocations: `SequenceSource` and `Display` maintain as state the position within the input file and the output produced thus far, respectively. `ChimeraFilter`'s state is the list of sequences that it has seen so far in the current group. If a fault occurred in the execution of the workflow, the following information will be lost: (1) the content of the queues between actors, i.e., tokens produced by actors but not yet consumed; (2) the point in the workflow execution schedule as observed by the workflow engine; and (3) the internal states of all actors. Correctly resuming workflow execution requires reconstructing all of this information. In many workflow systems, such as Kepler and Taverna it can be challenging to (1) recorde the main-memory actor-actor data transport, i.e. data flowing within the workflow engine without persistent storage[1]; (2) resume workflows that use non-trivial scheduling algorithms for multiple actor invocations based on data availability; and (3) capture the state of stateful actors that are invoked multiple times. In this paper, we show how to do so efficiently with low runtime overhead.

In particular, we make the following contributions:

- We present a general architecture for recovering from workflow crashes, and two concrete strategies (*replay* and *checkpoint*) that provide a balance between recovery speed and required provenance data.
- Our approach is applicable to workflows that can contain both stateful and stateless black-box actors. To the best of our knowledge, this is the first work to consider stateful actors in a fault tolerance context.
- Our approach is applicable to different models of computation commonly used in scientific workflow systems (namely DAG, SDF, PN, and DDF). We achieve this generality by mapping the different models of computations to a common model.
- Our *replay* strategy significantly improves performance over the naïve strategy (77% in our preliminary evaluation). Since this strategy is based on provenance data that is already recorded routinely for data lineage purposes, it adds no runtime overhead.

[1] Even if data is persisted to disk due to large data sizes, data handles are usually kept in main memory.

– Finally, we propose an extension to commonly used provenance models, i.e., to record actor states at appropriate points in time. This not only adds information valuable from a provenance point of view, but also enables our *checkpoint* strategy to recover workflows in a very short time span (98% improvement over a naïve re-execution) independent of the amount of work performed prior to the workflow crash.

The rest of the paper is organized as follows. Section 2 presents the fundamentals of our workflow recovery framework. In Section 3, we describe two recovery strategies and how to apply them to different models of computation. Section 4 reports on our prototypical implementation and preliminary evaluation. In Section 5, we provide a brief discussion of related work, and we conclude in Section 6.

2 Fault Tolerance Approach

Our approach generalizes the rescue-DAG method [7,8,9], which is used to recover DAGMan workflows after workflow crashes. DAGMan is a single-invocation model of computation, i.e., all actors are invoked only once with a "read-input—compute—write-output" behavior. The rescue-DAG is a sub-graph of the workflow DAG containing exactly those actors that have not yet finished executing successfully. After a crash, the rescue-DAG is executed by DAGMan, which completes the workflow execution.

To facilitate the execution of workflows on streaming data, several models of computation (e.g., Synchronous DataFlow (SDF) [10], Process Networks (PN) [11], Collection Oriented MOdeling and Design (COMAD) [12] and Taverna [13]) allow actors to have multiple invocations.

If the rescue-DAG approach were applied directly to workflows based on these models of computation, i.e., if all actors that had not completed all of their invocations were restarted, then in many cases a large fraction of the actors in a resumed workflow would be re-executed from the beginning. Instead, our approach aims to resume each actor after its last successful invocation. The difficulties of this approach are the following: (1) The unfolded *trace graph* (which roughly corresponds to the rescue-DAG) is not known a priori but is implicitly determined by the input data. (2) Actors can maintain internal state from invocation to invocation. This state must be restored. (3) The considered models of computation (e.g., SDF, PN, COMAD, Taverna) explicitly model the flow of data across channels, and the corresponding workflow engines perform these data transfers at run time. A successful recovery mechanism in such systems thus needs to re-initialize these internal communication channels to a consistent state. In contrast, data movement in DAGMan workflows is handled by the actors opaquely to the DAGMan scheduler (e.g., via naming conventions) or by a separate system called Stork [14]; materializing on disk all data passing between actors simplifies fault tolerance in these cases.

In the following, we present a simple relational model of workflow definitions and provenance information. We employ this model to define recovery strategies

using logic rules. Due to space restrictions, we concentrate here on the SDF and PN models of computation. SDF represents a model that is serially executed according to a statically defined schedule, while PN represents the other extreme of a parallel schedule only synchronized through the flow of data.

2.1 Basic Workflow Model

Scientific workflow systems use different languages to describe workflows and different semantics to execute them. However, since most scientific workflow systems are based on dataflow networks [15,11], a common core that describes the basic workflow structure can be found in every model of computation.

Core Model. Many workflow description languages allow nesting, i.e., embedding a sub-workflow within a workflow. The relation `subworkflow(W,Pa)` supports this nesting in our schema and stores a tuple containing the sub-workflow name `W` and the parent workflow name `Pa`. Each workflow in this hierarchy is associated with a model of computation (MoC) using the relation `moc(W,M)` that assigns the MoC `M` to the workflow `W`.

Actors represent computational functions that are either implemented using the language of the workflow system or performed by calling external programs. The separation of computations in multiple invocations sometimes requires that an actor maintains state across invocations.

Stateless Actor. The values of tokens output during an invocation depend only on tokens input during this invocation.

Stateful Actor. The values of tokens output during an invocation may depend on tokens received during previous invocations.

The predicate `actor(A,W,S)` embeds an actor with unique name `A` into the workflow `W`. The flag `S` specifies whether the actor is stateful or stateless.

Although the data shipping model is implemented differently in various workflow systems, it can be modeled uniformly as follows: Each actor has named ports, which send and receive data tokens. One output port can be connected to many input ports. In this situation, the token is cloned and sent to all receivers. Connecting multiple output ports to one channel is prohibited due to the otherwise resulting write conflicts. Ports are expressed with the predicate `port(A,P,D)` in our schema. The port with name `P` is attached to actor `A`. `D` specifies the direction in which data is sent, i.e., `in` or `out`. Ports are linked through the relation `link(A,P,L)` by sharing the same link identifier `L` (the third parameter of the link relation). A link from the port `p` of actor `a` to the port `q` of actor `b` is encoded as `link(a,p,l)` and `link(b,q,l)`.

Application to Process Networks with Firing. A Process Network (PN), as defined by [15], is a general model of computation for distributed systems. In Kahn PN, actors communicate with each other through unbounded unidirectional FIFO channels. Workflows of the model *PN with firings* [11], a refinement of Kahn PN, can be described with the four core relations `Subworkflow`, `Actor`, `Port`, and `Link`. The PN execution semantics allow a high level of parallelism,

i.e., all actors can be invoked at the same time. After an invocation ends, the actor will be invoked again, consuming more data. This procedure stops either when the actor explicitly declares completion or by reaching the end of the workflow execution. A PN workflow ends when all remaining running invocations are deadlocked on reading from an input port.

Application to Synchronous DataFlow (SDF). Besides the data captured by the four core relations (`Subworkflow`, `Actor`, `Port`, and `Link`), workflow models can provide additional information. As an example, SDF workflow descriptions require annotations on ports. In SDF, output ports are annotated with a fixed *token production rate* and input ports have a fixed *token consumption rate*. Both rates are associated with ports using the predicate `token_transfer(A,P,N)` in our model. During an invocation, each actor `A` is required to consume/produce `N` tokens from the input/output port `P`.

Another extension is the *firing count* of an actor that specifies the maximum number of actor invocations during a workflow execution. The predicate `firing_count(A,N)` provides this number (`N`) for an actor `A`.

Unlike in PN, where the actors synchronize themselves through channels, the execution of SDF is based on a static schedule that is repeatedly executed in *rounds*. The number of firings of each actor per round is determined by solving balance equations based on token production and consumption rates [10].

2.2 Review of Provenance Model

Another critical part of our approach is the definition of a simple, prototypical provenance model. It defines which observables are recorded during runtime. The Open Provenance Model (OPM) [6] captures the following basic observables: (1) artifact generation, i.e., token production; (2) artifact use, i.e., token consumption; (3) control-flow dependencies, i.e., `was_triggered_by` relation; and (4) data dependencies, i.e., `was_derived_from` relation. A more comprehensive provenance schema was defined by Crawl et al. in [16]. It captures the OPM observables in more detail, e.g., it provides timestamps for the beginning and end of invocations. In addition, it records metadata about the workflow execution as well as the evolution of the workflow. All provenance information is recorded by the workflow system transparently without modifications to actors.

Our provenance model uses the basic observables from OPM and adds additional details about events that occurred during an invocation cycle. As soon as an invocation starts, the actor name `A` and its corresponding invocation number `N` are stored in the relation `invocation(I,A,N,Z)` with the status attribute `Z` set to running. A unique identifier `I` is assigned to each invocation. Some models of computation allow an actor to indicate that all invocations are completed, for instance if the maximum firing count in SDF is reached. This information is captured in our provenance model as well. When an actor successfully completes an invocation and indicates that it will execute again, the status attribute in the corresponding provenance record is updated to iterating. Otherwise, this attribute status is set to done.

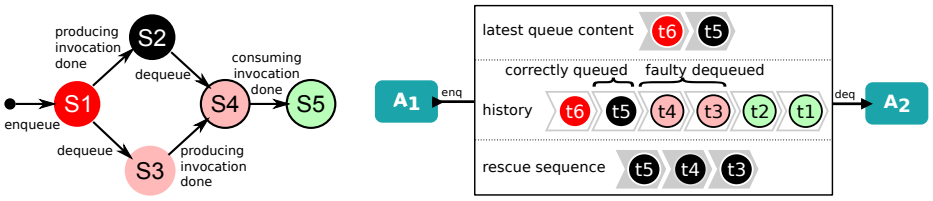

Fig. 2. Input queues with history and token state. Each token produced during workflow execution can be in one of five states. Events on the producing and consuming actors trigger transitions between token states, shown on the left. The right graph shows three views of a channel: (1) the current content of the queue during an execution in the first row, (2) the history of all tokens passed through this channel associated with their state in the middle row, and (3) the *rescue sequence* of tokens that needs to be restored in the third row.

The second observable process in our model is the flow of tokens. Many workflow engines treat channels that define the dataflow as first-class citizens of the model. The dependencies between data tokens are of general interest for provenance. They can be inferred from the core workflow model in combination with the token consumption (read) and production (write) events.

Our model stores read and write events in the `event(Y,T,I,Po,N)` relation. The first entry `Y` determines the event type, i.e., token production events are indicated by the constant `w` while consumption events are encoded with the value `r`. `T` is the data token to be stored. The following two attributes specify which actor invocation `I` triggered this event and on which port `Po` it was observed. The last element `N` in the tuple is an integer value that is used to establish an order of events during the same actor invocation on the same port. Establishing an order using timestamps is not practical because of limited resolution and time synchronization issues.

Based on the `event` relation, the queues of all channels can be reconstructed for any point in time. Figure 2 shows a queue at the time of a workflow failure. Using provenance we can restore the whole history of this queue (shown in the middle right). Based on this history, we can determine the *rescue sequence* of tokens that are independent of failed invocations, i.e., tokens in state S2 and S4.

3 Recovery Strategies

Depending on the model of computation and the available provenance data, different recovery approaches can be used. We will now present our two strategies *replay* and *checkpoint*.

3.1 The Replay Strategy: Fast-Forwarding Actors

Re-running the entire workflow from the beginning is a naïve recovery strategy, which is often impractical, especially when a long-running workflow fails a sig-

nificant period of time into its execution. The role of provenance in restoring a workflow execution is similar to that of log files used in database recovery.

Stage 1. In the first stage of the *replay* strategy, the point of a failure is determined using provenance information. Invocations of actors that were running when the fault occurred are considered faulty and their effects have to be undone. Query (1) retrieves the invocation identifiers `I` of faulty invocations.

```
faulty_invoc(I) :- invocation(I,_,_,running).                    (1)
```

Actors with invocation status `done` are not recovered, since they are not needed for further execution. All other actors `A` are retrieved by query (2) and they need to be recovered.

```
finished_actors(A) :- invocation(_,A,_,done).
restart_actors(A) :- actor(A,_,_), not finished_actors(A).       (2)
```

Stage 2. If an actor is stateless, it is ready to be resumed without further handling. However, if an actor is stateful, its internal state needs to be restored to its *pre-failure state*, i.e., the state after the last successful invocation. Each actor is executed individually by presenting it with all input data the actor received during successful invocations. This input data is retrieved from the provenance log, where it is readily available. The `replay(A,I)` query (3) extracts the identifiers of all actor invocations that need to be replayed. The tokens needed for those re-invocations are provided by (4). This query retrieves for each port `P` of actor `A` the tokens `T` that are needed to replay invocation `I`. `N` is the sequence number of token `T` at input port (queue) `P`. The replay does not need to be done in the same order as in the original workflow schedule. All actors can be re-executed in parallel using only the input data recorded as provenance. The actor output can either be discarded or checked against the recorded provenance to verify the workflow execution.

```
replay(A,I) :- actor(A,_,stateful), invocation(I,A,_,_),        (3)
      not faulty_invoc(I).
replay_token(A,P,I,T,N) :- replay(A,I), event(r,T,I,P,N).       (4)
```

In order to correctly recover a workflow execution, the problem of side-effects still needs to be addressed. Either stateful actors should be entirely free of side-effects or side-effects should be idempotent. That is, it must not matter whether the side-effect is performed once or multiple times. Examples of side-effects in scientific workflows include the creation or deletion of files, or sending emails. Deleting a file (without faulting if the file does not exist) is an idempotent operation. Further, creating a file is idempotent if an existing file is overwritten. Sending an email is, strictly speaking, not idempotent, since if done multiple times, multiple emails will be sent.

Stage 3. Once all actors are instantiated and in pre-failure state, the queues have to be initialized with the *restore sequence*, i.e., all valid tokens that were present before the execution failed. Tokens created by faulty invocations must be removed, and those consumed by a failed invocation are restored. This information is available in basic workflow provenance and can be queried using (5).

For each port `Po` of an actor `A` the query retrieves tokens `T` with the main order specified by the invocation order `N1`. However, if multiple tokens are produced in one invocation, the token order `N2` is used for further ordering.

The auxiliary view `invoc_read(A,P,T)` contains all actors `A` and the corresponding ports `P` that read token `T`. The view `connect(A1,P1,C,A2,P2)` returns all output ports `P1` of actor `A1` that are connected to actor `A2` over input port `P2` through channel `C`. The auxiliary rule (5.1) computes the queue content in state S2 (see Fig. 2), i.e., tokens that were written by another actor but not yet read by actor `A2` on port `P2`. The second rule (5.2) adds back the queue content in state S4, i.e., tokens that were read by a failed invocation of actor `A2`.

```
current_queue(A2,P2,T,N1,M1) :- queue_s2(A2,P2,T,N1,M1).          (5)

current_queue(A2,P2,T,N1,M1) :- queue_s4(A2,P2,T,N1,M1).

queue_s2(A2,P2,T,N1,M1) :- connect(A1,P1,C,A2,P2),                (5.1)
      invocation(I1,A1,N1,_), event(w,T,I1,P1,M1),
      not invoc_read(A2,P2,T), not faulty_invoc(I1).

queue_s4(A2,P2,T,N1,M1) :- connect(A1,P1,C,A2,P2),                (5.2)
      invocation(I1,A1,N1,_), event(w,T,I1,P1,M1),
      invocation(I2,A2,_,_),event(r,T,I2,P2,_),
      faulty_invoc(I2).
```

Stage 4. After restoring actors and recreating the queues, faulty invocations of actors that produced tokens which were in state S3 have to be repeated in a "sandbox". This ensures that tokens in state S3 are not sent to the output port after being produced but are discarded instead. If these tokens are sent, then invocation based on them are duplicated. Rule (6) determines tokens `T` that were in state S3 and it returns the invocation ID `I`, the port `P` this token was sent from, and the sequence number in which the token was produced. Query (7) determines which invocations produced tokens in state S3 and therefore have to be repeated in a sandbox environment.

```
queue_s3(I,P,T,N) :- invocation(I,A1,_,_),                        (6)
      faulty_invoc(I), event(w,T,I,P,N),
      connect(A1,P,C,A2,P2), invocation(I2,A2,_,_),
      not faulty_invoc(I2), event(r,T,I2,P2,_).
invoc_sandbox(I) :- faulty_invoc(I), queue_s3(I,_,_,_,_).         (7)
```

After executing the sandbox, the workflow is ready to be resumed. The recovery system provides information about where to begin execution (i.e., the actor at which the failure occurred) to the execution engine (e.g., the SDF scheduler) and then the appropriate model of computation controls execution from that point on.

To summarize, the most expensive operation in the *replay* strategy is the re-execution of stateful actors, which is required to reset the actor to its pre-failure state. Our *checkpoint* strategy provides a solution to avoid this excessive cost.

3.2 The Checkpoint Strategy: Using State Information

Many existing workflow systems are shipped with stateful actors or new actors are developed that maintain state. Because actors in scientific workflows usually have complex and long-running computations to perform, the *replay* strategy can be very time-consuming.

Current provenance models, such as the one used in [16], either do not include the state of actors or record limited information about state as in [17]. The Read-Write-Reset model (as presented in [18]), e.g., records only state reset events, which specify that an actor is in its initial state again. This can be seen as a special case of the *checkpoint* strategy we will present, where states are only recorded when they are equal to the initial state.

To support a faster recovery, we propose to make the actor's state a distinct observation for provenance. Recording state information not only helps to recover workflows, but also makes provenance traces more meaningful: Instead of linking an output token of a stateful actor to all input tokens across its entire history, our model links it to the state input and the current input only.

An actor's state can be recorded by the workflow engine at any arbitrary point in time when the actor is not currently invoked. To integrate checkpointing into the order of events, we store state information immediately after an invocation, using the invocation identifier as a reference for the state. The predicate `state(I,S)` stores the actor's state `S` together with the identifier of the preceding invocation `I` of that actor. The information required to represent an actor state depends on the workflow system implementation.

Given this additional state information the workflow recovery engine can speed up the recovery process. The *checkpoint* strategy is based on the *replay* strategy but extends it with checkpointing.

Stage 1. When normally executing the workflow, state is recorded in provenance. In case of a fault, the recovery system first detects the point of failure. Then the provenance is searched for all checkpoints written for stateful actors. Rule (8) retrieves the state `S` of each invocation `I` of a given actor `A`. If no state was recorded then the invocation will not be contained in this relation:

```
restored_state(A,I,N,S) :- actor(A,_,stateful),                    (8)
      invocation(I,A,N,_),state(I,S), not faulty_invoc(I).
```

If states were stored for an actor, this actor is updated with the latest available state. Rule (9) will determine the latest recoverable invocation `I` and the *restorable pre-failure state* `S` captured after that invocation.

```
restored_stateGTN(A,I,N2) :- restored_state(A,I,N,_), N > N2.
latest_state(A,I,S) :- restored_state(A,I,N,S),                    (9)
      not restored_stateGTN(A,I,N).
```

Stage 2. Now only those successfully completed invocations that started after the checkpoint have to be replayed. This will use the same methods described above for the *replay* strategy.

Stage 3 and 4. Same as in the *replay* strategy.

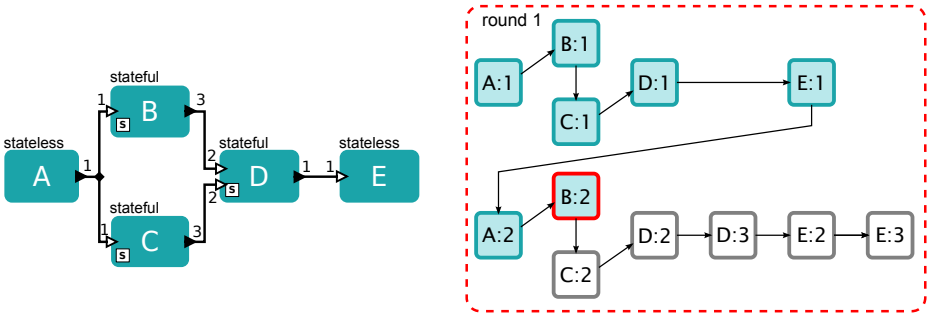

Fig. 3. SDF workflow example with state annotations on the left and a corresponding schedule on the right. The schedule describes the execution order. The red circle indicates the failure during the second invocation of B.

If provenance information is kept only for fault tolerance, not all data needs to be stored persistently. Only data tokens representing the active queues or consumed by stateful actors after the last checkpoint need to be persisted. All other data can be discarded or stored in a compressed representation.

3.3 Recovering SDF and PN Workflows

The SDF example in Figure 3 demonstrates our *checkpoint* strategy. Below, we explain the differences when dealing with PN models.

Synchronous Dataflow (SDF). Figure 3 shows a sample SDF workflow with ports annotated with consumption and production rates for input and output ports, respectively. Actors A and E are stateless while all other actors maintain state. A is a source and will output one data item (token) each time the actor is invoked. A also has a *firing count* of two, which limits the total number of its invocations. Actors B and C consume one token on their input ports and output three tokens per invocation. Outputs are in a fixed, but distinguishable, order, so that an execution can fail between the production of two tokens. Actor D will receive two tokens in each invocation from both B and C and will output one new token.

The schedule in Fig. 3 was computed, as usual, before the workflow execution begins, based on the token production and consumption rates in the model. Actors were then invoked according to the schedule until the workflow crash occurred. All invocations up to the second invocation of A (A:2) completed successfully, invocation B:2 was still running and all other invocations were scheduled for the future. The failed workflow execution, together with checkpointing and data shipping, is summarized in Fig. 4. For recovery, the workflow description as well as the recorded provenance is used by our *checkpoint* strategy. In the following, we will describe the details of the recovery process (shown in Figure 4).

Stateless actors are in the correct state (i.e., pre-failure state) immediately after initialization. That is why actor E is in its proper state after simple initial-

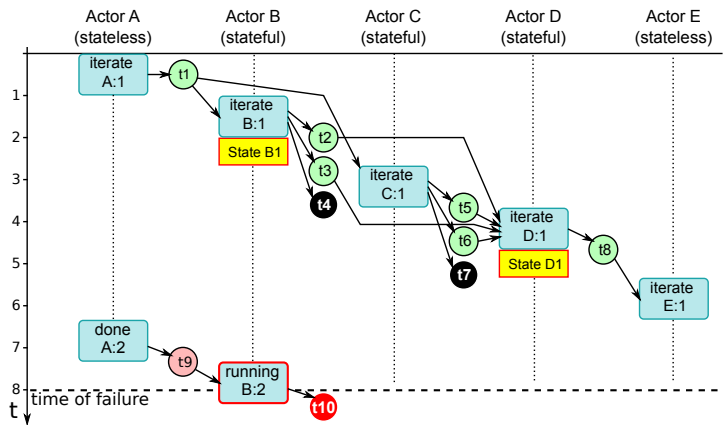

Fig. 4. Workflow execution up to failure in B:2. The states of actors B and D are stored, but no checkpoint exists for C. Token t1 is only send once, but is duplicated by link to both actors B and C. Tokens t4 and t7 are never read. Token t9 is read by a faulty invocation, and t10 is written by a faulty invocation.

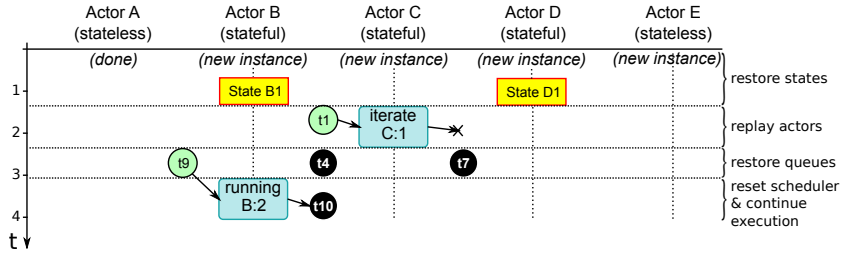

Fig. 5. Individual stages to recover the sample workflow with *checkpoint* strategy. Note how only a small amount of invocations are repeated (compared to Figure 4).

ization and actor A is identified as done and will be skipped. Both actors B and D are stateful and a checkpoint is found in the provenance. Therefore, the recovery system instantiates these actors and restores them to the latest recorded state. The state at this point in the recovery process is shown in Fig. 5 above the first horizontal line. Second, stateful actors that either have no checkpoints stored (e.g., actor C) or that have successful invocations after the last checkpoint need to have those invocations replayed. To do this for actor C:1, the input token t1 is retrieved from the provenance store. In the next stage, all queues are restored. Since the second invocation of actor B failed, the consumed token t9 is restored back to the queue. Additionally, all tokens that were produced but never consumed (e.g., t4 and t7) are restored to their respective queues. After all queues are rebuilt, the recovery system has to initialize the SDF scheduler to resume execution. This entails setting B as the next active actor, since its invocation was interrupted by the crash. After setting the next active actor, the recovery system can hand over the execution to the workflow execution engine. This final

recovery state is shown in Fig. 5. The recovery time is significantly improved compared to the original runtime shown in Fig. 4.

Process Networks (PN). The example SDF workflow shown in Fig. 3 can also be modeled using PN. Since actors under PN semantics have variable token production and consumption rates, these constraints cannot be leveraged to narrow the definition of a faulty invocation. Additionally, repeated invocations are not necessarily required for actors to perform their function. For instance, actor D can be invoked only once, while actor B is invoked multiple times. All invocations in PN run concurrently, and tokens on a port have to be consumed after they are produced and only in the order they were produced. Finally, there are no defined firing limits. Many systems allow an actor to explicitly declare when it is done with all computations, which is recorded in provenance. Actors without that information are invoked until all actors in the workflow are waiting to receive data.

These characteristics have some implications on the recovery process. First, since all actors are executed in parallel, a crash can affect all actors in a workflow. Since actors are invoked in parallel during workflow execution, the recovery engine can safely restore actors in parallel. All actors are instantiated simultaneously at the beginning of the workflow run, in contrast to Fig. 4. Long-running actor invocations reduce the availability of checkpoints and cause longer replay times. Finally, PN uses deadlock detection to define the end of a workflow, which makes it difficult to determine whether a particular actor is actually done (unless it explicitly says so) or just temporary deadlocked. Anything short of all actors being deadlocked by blocking reads (meaning the workflow is done) gives no useful information about which actors will exhibit future activity.

4 Evaluation

To evaluate the performance of the different recovery strategies, we implemented a prototype of our proposed approach in Kepler [19]. The current implementation adds fault tolerance to non-hierarchical SDF workflows.

Implementation. Our fault tolerance framework implements all features necessary for the *checkpoint* recovery strategy (as well as the *replay* strategy) in a separate workflow restore class that is instrumented from the director. We used the provenance system of Crawl et al. [16], which was altered to allow the storage of actor states and tokens. Instead of storing a string representation of a token, which may be lossy, we store the whole serialized token in the provenance database. When using the standard token types, this increases the amount of data stored for each token only slightly. Actors can be explicitly marked as stateless using an annotation on the implementing Java class. Thus, we avoid checkpointing and replay for stateless actors.

During a normal workflow execution, the system records all tokens and actor invocations. Currently, checkpoints for all stateful actors are saved after each execution of the complete SDF schedule. An actor's state is represented by a

serialization of selected fields of the Java class that implements the actor. There are two different mechanisms that can be chosen: (1) a blacklist mode that checks fields against a list of certain transient fields that should not be serialized, and (2) a whitelist mode that only saves fields explicitly annotated as state. The serialized state is then stored together with the last invocation id of the actor in the state relation of Kepler's provenance database. The serialization process is based on Java's object serialization and also includes selected fields of super classes.

During a recovery, the latest recorded checkpoint of an actor is restored. All stored actor fields are deserialized and overwrite the actor's fields. This leaves transient member fields intact and ensures that the restored actor is still properly integrated into its parent workflow. Successful invocations completed after a checkpoint or where no checkpoint exists are replayed to restore the correct pre-failure state. For the replay, all corresponding serialized tokens are retrieved from the provenance database. Then, the input queues of an actor are filled with tokens necessary for one invocation and the actor is fired. Subsequently, input and output queues are cleared again before the next invocation of an actor is replayed. The current implementation replays actors serially.

Next, all the queues are restored. For each actor, all tokens are retrieved that were written to an input port of the actor and not read by the actor itself before the fault. These tokens are then placed in the proper queues, preserving the original order. Finally, the scheduling needs modifications to start at the proper point. This process is closely integrated with the normal execution behavior of the SDF director. The schedule is traversed in normal order, but all invocations are skipped until the failed invocation is reached. At this stage, the normal SDF execution of the schedule is resumed.

Preliminary Experimental Evaluation. For an initial evaluation of the practicality of a provenance-based recovery, we created the synthetic workflow shown in Fig. 6. This model simulates a typical scientific workflow with long-running computations and a mix of stateless and stateful actors. We ran this workflow to its completion to measure the running time for a successful execution. We then interrupted the execution during the third invocation of actor C. After this, we loaded the workflow again and resumed its execution using the three different strategies: re-execution from the beginning, *replay* and *checkpoint*.

We ran the experiments ten times for each strategy. In a typical scientific workflow with computation time domination over the data transfer time, provenance recording adds an overhead of 139 milliseconds (with a standard deviation σ of $157.22ms$) to the workflow execution time of 80.7 seconds ($\sigma = 0.16s$). The naïve approach of re-running the whole workflow takes about 80.8 seconds ($\sigma = 0.017s$), repeating 55.8 seconds ($\sigma = 0.037s$) of execution time from before the crash. The *replay* strategy based on standard provenance already achieves a major improvement. The total time for a recovery using this strategy of approximately 12.8 seconds ($\sigma = 0.17s$) is dominated by replaying two invocations of stateful actor C in 10 seconds. The remaining 2.8 seconds are the accumulated overhead for retrieving and deserializing tokens for the replay as well as

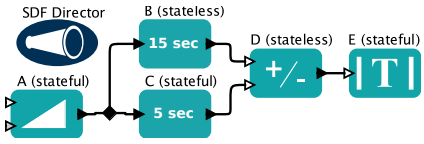

Fig. 6. Synthetic SDF workflow. Actor A is a stateful actor generating a sequence of increasing numbers starting from 0. B is a stateless actor that has a running time of 15 seconds. C is stateful and needs 5 seconds for each invocation. D is fast running stateless actor. E is a stateful "Display" actor.

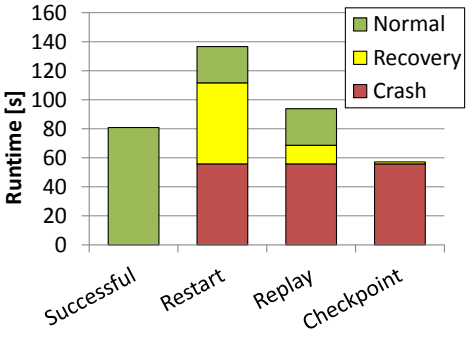

Fig. 7. Performance evaluation of different recovery strategies.

for restoring the queue content. After the recovery, the workflow execution finishes in 25.8 seconds ($\sigma = 0.02s$). The *replay* strategy reduced the recovery cost from 55.8 seconds to 12.8 seconds, or by 77%, in this workflow. The checkpoint strategy reduced the recovery time to only 1.3 seconds ($\sigma = 0.03s$), including the time for the deserialization process of state as well as the queue restoration. This strategy reduces the recovery time of the synthetic workflow by 97.6% compared to the naïve strategy. Checkpointing is so efficient because it does not scale linearly with the number of tokens sent like the naïve and *replay* strategies. This strategy also benefits from invocation runtimes that are significantly longer than the checkpointing overhead.

5 Related Work

In DAGMan [9], a basic fault tolerance mechanism (rescue-DAG) exists. Jobs are scheduled according to a directed graph that represents dependencies between those jobs. Initially the rescue-DAG contains the whole DAG but as jobs execute successfully, they are removed from the rescue-DAG. If a failure occurs, the workflow execution can thus be resumed using the rescue-DAG, only repeating jobs that were interrupted.

Feng et al. [17] present a mechanism for fault management within a simulation environment under real time conditions. Starting from a "checkpoint" in the execution of an actor, state changes are recorded incrementally and can then be undone in a "rollback". This backtracking approach allows to capture the state of an actor through a preprocessing step that adds special handlers for internal state changes wherever a field of an actor is modified. However, this solution can only be used during runtime of the workflow system. It does not provide checkpoints that cover the full state, and, more importantly, no persistent state storage is available for access after a workflow crash.

Dan Crawl et al. [16] employed provenance records for fault tolerance. Their Kepler framework allows the user to model the reactions upon invocation fail-

ures. The user can either specify a different actor that should be executed or that the same actor should be invoked again using input data stored in provenance records. However, they don't provide a fast recovery of the whole workflow system. Neither is the approach applicable for stateful actors.

Fault tolerance in scientific workflows has often been addressed using caching strategies. While still requiring a complete restart of the workflow execution, computation results of previous actor invocations are stored and reused. Swift [20] extends the rescue-DAG approach by adding such caching. During actor execution, a cache is consulted (indexed by the input data), and if an associated output is found, it will be used, avoiding redundant computation. Swift also employs this strategy for optimizing the re-execution of workflow with partially changed inputs. Conceptually, this can be seen as an extension of the rescue-DAG approach. Podhorszki et al. [3] described a checkpoint feature implemented in the ProcessFileRT actor. This actor uses a cache to avoid redundant computations. A very similar approach was implemented by Hartman et al. [1]. Both techniques are used to achieve higher efficiency for computation and allow a faster re-execution of workflows. However, these implementations are highly customized to their respective use cases and integrated in one or several actors rather being a feature of the framework. Also, [3] assumes that only external programs are compute intensive, which is not always the case, as can be seen in [1], where actors perform compute intensive calculations within the workflow system. Furthermore, caching strategies can only be applied to stateless actors, making this approach very limited. In contrast, our approach aims to integrate fault tolerane mechanisms into the workflow engine. Stateless actors are not re-executed during a recovery, since input and corresponding outputs are available in provenance, and the actor state does not need to be restored.

Wang et al. [21] presented a transactional approach for scientific workflows. Here, all effects of arbitrary subworkflows are either completed successfully or in case of a failure undone completely (the dataflow-oriented hierarchical atomicity model is described in [21]). In addition, it provides a dataflow-oriented provenance model for those workflows. The authors assumed that actors are white boxes, where data dependencies between input and output tokens can be observed. They describe a smart re-run approach similar to those presented by Podhorszki et al. and Hartman et al. [1]. Input data of actors is compared to previous inputs, and if an actor is fired with the same data, the output can easily be restored from provenance information rather than re-executing the actor. This white box approach differs from our black box approach that requires setting the internal state of stateful actors. Our system is more generally applicable, as not all actors are available in a white box form that allows for the direct observation of dependencies.

6 Conclusion

We introduced a simple relational representation of workflow descriptions and their provenance information in order to improve fault-tolerance in scientific

workflow systems. To the best of our knowledge, our approach is the first to handle not only individual actor failures, but (i) failures of the overall workflow, where workflows (ii) can have a stream-oriented, pipeline-parallel execution model, and (iii) can have loops, and where (iv) actors can be stateful and stateless. Another unique feature of our approach is that the workflow system itself, upon "smart resume" can handle the recovery, i.e., unlike other current approaches, neither actors nor the workflow are burdened with implementing parts of the recovery logic, since the system takes care of everything. To allow for checkpointing of internal state from stateful actors, we have developed an extension to the standard OPM-based provenance models. Information necessary to recover a failed execution of a scientific workflow is extracted from the relational representation via logic rules, allowing our approach to be easily deployed on various provenance stores. We defined and demonstrated a *replay* strategy that speeds up the recovery process by only re-executing stateful actors. Our *checkpoint* strategy improves on *replay* by using the saved checkpoints to significantly reduce actor re-execution. We implemented our approach in the Kepler system. In a preliminary evaluation, we compared our strategies to a naïve re-execution. Here, *replay* and *checkpoint* could reduce recovery times by 77% and 98%, respectively. This highlights the advantage of checkpointing in scientific workflows with compute intensive stateful actors.We plan to add support for other models of computation, e.g. dynamic dataflow (DDF) [22] to our Kepler implementation, in order to add fault tolerance to specific complex workflows [3]. We also plan to port our approach to other systems, e.g., RestFlow [23]. Another enhancement will be to parameterize the time between checkpoint saving as either a number of invocations, or in terms of wall-clock time to balance the overhead of provenance recording and recovery time.

Acknowledgments. Work supported through NSF grant OCI-0722079 and DOE grant DE-FC02-07ER25811.

References

1. Hartman, A., Riddle, S., McPhillips, T., Ludäscher, B., Eisen, J.: Introducing W.A.T.E.R.S.: a Workflow for the Alignment, Taxonomy, and Ecology of Ribosomal Sequences. BMC Bioinformatics **11**(1) (2010) 317
2. Ceyhan, E., Allen, G., White, C., Kosar, T.: A grid-enabled workflow system for reservoir uncertainty analysis. In: Proceedings of the 6th Int'l workshop on Challenges of large applications in distributed environments. CLADE '08 (2008)
3. Podhorszki, N., Ludäscher, B., Klasky, S.A.: Workflow automation for processing plasma fusion simulation data. In: Proceedings of the 2nd workshop on Workflows in support of large-scale science. WORKS '07, New York, NY, USA (2007) 35–44
4. Missier, P., Soiland-Reyes, S., Owen, S., Tan, W., Nenadic, A., Dunlop, I., Williams, A., Oinn, T., Goble, C.: Taverna, reloaded. In: Scientific and Statistical Database Management, Springer (2010) 471–481
5. Bowers, S., McPhillips, T., Ludäscher, B., Cohen, S., Davidson, S.: A Model for User-Oriented Data Provenance in Pipelined Scientific Workflows. In: Provenance and Annotation of Data. Volume 4145 of LNCS. Springer (2006) 133–147

6. Moreau, L., Freire, J., Futrelle, J., McGrath, R., Myers, J., Paulson, P.: The Open Provenance Model: An Overview. In: Provenance and Annotation of Data and Processes. Volume 5272 of LNCS. Springer (2008) 323–326
7. Frey, J.: Condor DAGMan: Handling inter-job dependencies. Technical report, University of Wisconsin, Dept. of Computer Science (2002)
8. Deelman, E., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Patil, S., Su, M.H., Vahi, K., Livny, M.: Pegasus: Mapping Scientific Workflows onto the Grid. In: Grid Computing. Volume 3165 of LNCS. Springer (2004) 131–140
9. Hernandez, I., Cole, M.: Reliable DAG scheduling on grids with rewinding and migration. In: Proceedings of the first Int'l conference on Networks for grid applications. GridNets '07, ICST (2007) 3:1–3:8
10. Lee, E.A., Messerschmitt, D.G.: Static scheduling of synchronous data flow programs for digital signal processing. IEEE Trans. Comput. **36** (1987) 24–35
11. Lee, E., Matsikoudis, E.: The semantics of dataflow with firing. From Semantics to Computer Science: Essays in memory of Gilles Kahn. Cambridge University Press, Cambridge (2008)
12. Dou, L., Zinn, D., McPhillips, T., Köhler, S., Riddle, S., Bowers, S., Ludäscher, B.: Scientific Workflow Design 2.0: Demonstrating Streaming Data Collections in Kepler. In: 27th IEEE Int'l Conference on Data Engineering. (2011)
13. Turi, D., Missier, P., Goble, C., De Roure, D., Oinn, T.: Taverna workflows: Syntax and semantics. In: IEEE Int'l Conference on e-Science and Grid Computing, IEEE (2008) 441–448
14. Kosar, T., Livny, M.: Stork: Making data placement a first class citizen in the grid. In: Proceedings of the 24th Int'l Conference on Distributed Computing Systems, 2004., IEEE (2005) 342–349
15. Kahn, G.: The Semantics of a Simple Language for Parallel Programming. In: Information Processing '74: Proceedings of the IFIP Congress. North-Holland, New York, NY (1974) 471–475
16. Crawl, D., Altintas, I.: A Provenance-Based Fault Tolerance Mechanism for Scientific Workflows. In: Provenance and Annotation of Data and Processes. Volume 5272 of LNCS. Springer (2008) 152–159
17. Feng, T., Lee, E.: Real-Time Distributed Discrete-Event Execution with Fault Tolerance. In: Real-Time and Embedded Technology and Applications Symposium, 2008. RTAS '08. IEEE. (2008) 205 –214
18. Ludäscher, B., Podhorszki, N., Altintas, I., Bowers, S., McPhillips, T.: From computation models to models of provenance: the RWS approach. Concurr. Comput.: Pract. Exper. **20** (2008) 507–518
19. Ludäscher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger, E., Jones, M., Lee, E.A., Tao, J., Zhao, Y.: Scientific workflow management and the Kepler system: Research Articles. Concurr. Comput. : Pract. Exper. **18** (2006) 1039–1065
20. Zhao, Y., Hategan, M., Clifford, B., Foster, I., Von Laszewski, G., Nefedova, V., Raicu, I., Stef-Praun, T., Wilde, M.: Swift: Fast, reliable, loosely coupled parallel computation. In: 2007 IEEE Congress on Services, IEEE (2007) 199–206
21. Wang, L., Lu, S., Fei, X., Chebotko, A., Bryant, H.V., Ram, J.L.: Atomicity and provenance support for pipelined scientific workflows. Future Generation Computer Systems **25**(5) (2009) 568 – 576
22. Zhou, G.: Dynamic dataflow modeling in Ptolemy II. PhD thesis, University of California (2004)
23. McPhillips, T., McPhillips, S.: RestFlow System and Tutorial. https://sites.google.com/site/restflowdocs (April 2011)