Fault-tolerant Resource Reasoning

Gian Ntzik, Pedro da Rocha Pinto, and Philippa Gardner

Imperial College London
{gn408,pmd09,pg}@doc.ic.ac.uk

Abstract. Separation logic has been successful at verifying that programs do not crash due to illegal use of resources. The underlying assumption, however, is that machines do not fail. In practice, machines can fail unpredictably for various reasons, e.g. power loss, corrupting resources. Critical software, e.g. file systems, employ recovery methods to mitigate these effects. We introduce an extension of the Views framework to reason about such methods. We use concurrent separation logic as an instance of the framework to illustrate our reasoning, and explore programs using write-ahead logging, e.g. an ARIES recovery algorithm.

1 Introduction

There are many ways that software can fail: either software itself can be the cause of the failure (e.g. memory overflow or null pointer dereferencing); or the failure can arise independently of the software. These unpredictable failures are either *transient faults*, such as when a bit is flipped by cosmic radiation, or *host failures* (also referred to as crashes). Host failures can be classified into *soft*, such as those arising from power loss which can be fixed by rebooting the host, and *hard*, such as permanent hardware failure.

Consider a simple transfer operation that moves money between bank accounts. Assuming that bank accounts can have overdrafts, the transfer can be regarded as a sequence of two steps: first, subtract the money from one bank account; and then add the money to the other account. In the absence of host failures, the operation should succeed. However, if a host failure occurs in the middle of the transfer, money is lost. Programmers employ various techniques to recover some consistency after a crash, such as write-ahead logging (WAL) and associated recovery code. In this paper, we develop the reasoning to verify programs that can recover from host failures, assuming hard failures do not happen.

Resource reasoning, as introduced by separation logic [15], is a method for verifying that programs do not fail. A triple $\{P\}\ \mathbb{C}\ \{Q\}$ is given a *fault-avoiding*, partial correctness interpretation. This means that, assuming the precondition P holds then, if program $\mathbb{C}$ terminates, it must be the case that $\mathbb{C}$ does not fail and has all the resource necessary to yield a result which satisfies postcondition Q. Such reasoning guarantees the correct behaviour of the program, ensuring that the software does not crash itself due to bugs, e.g. invalid memory access. However, it assumes that there are no other failures of any form. To reason about programs that can recover from host failures, we must change the underlying assumptions of resource reasoning.

We swap the traditional resource models with one that distinguishes between *volatile* and *durable* resource: the volatile resource (e.g. in RAM) does not survive crashes; whereas the durable resource (e.g. on the hard drive) does. Recovery operations use the durable state to repair any corruptions caused by the host failure. We introduce *fault-tolerant resource reasoning* to reason about programs in the presence of host failures and their associated recovery operations. We introduce a new fault-tolerant Hoare triple judgement of the form:

$$S \vdash \left\{P_V \mid P_D\right\} \; \mathbb{C} \; \left\{Q_V \mid Q_D\right\}$$

which has a partial-correctness, *resource fault-avoiding* and *host failing* interpretation. From the standard resource fault avoiding interpretation: assuming the precondition $P_V \mid P_D$ holds, where the volatile state satisfies P_V and the durable P_D, then if $\mathbb{C}$ terminates and there is no host failure, the volatile and durable resource will satisfy Q_V and Q_D respectively. From the host-failing interpretation: when there is a host failure, the volatile state is lost, and after potential recovery operations, the remaining durable state will satisfy the *fault-condition* S.

We extend the Views framework [3], which provides a general account of concurrent resource reasoning, with these fault-tolerant triples to provide a general framework for fault-tolerant resource reasoning. We instantiate our framework to give a fault-tolerant extension of concurrent separation logic [11] as an illustrative example. We use this instantiation to verify the correctness of programs that make use of recovery protocols to guarantee different levels of fault tolerance. In particular, we study a simple bank transaction using write-ahead logging and a simplified ARIES recovery algorithm [8], widely used in database systems.

2 Motivating Examples

We introduce fault-tolerant resource reasoning by showing how a simple bank transfer can be implemented and verified to be robust against host failures.

2.1 Naive Bank Transfer

Consider a simple transfer operation that moves money between bank accounts. Using a separation logic [15] triple, we can specify the transfer operation as:

$$\begin{array}{ll} & \{\mathsf{Account}(\mathtt{from}, v) * \mathsf{Account}(\mathtt{to}, w)\} \\ \vdash & \mathtt{transfer}(\mathtt{from}, \mathtt{to}, \mathtt{amount}) \\ & \{\mathsf{Account}(\mathtt{from}, v - \mathtt{amount}) * \mathsf{Account}(\mathtt{to}, w + \mathtt{amount})\} \end{array}$$

The internal structure of the account is abstracted using the abstract predicate [12], $\mathsf{Account}(x, v)$, which states that there is an account x with balance v. The specification says that, with access to the accounts `from` and `to`, the `transfer` will not fault. It will decrease the balance of account `from` by `amount` and increase the balance of account `to` by the same value. We can implement the transfer operation as follows:

```
function transfer(from, to, amount) {
  widthdraw(from, amount); deposit(to, amount);
}
```

Using separation logic, it is possible to prove that this implementation satisfies the specification, assuming no host failures. This implementation gives no guarantees in the presence of host failures. However, for this example, it is clearly desirable for the implementation to be aware that host failures occur. In addition, the implementation should guarantee that in the event of a host failure the operation is *atomic*: either it happened as a whole, or nothing happened. Note that the word atomic is also used in concurrency literature to describe an operation that takes effect at a single, discrete instant in time. In §3 we combine concurrency atomicity of concurrent separation logic with host failure atomicity: if an operation is concurrently atomic then it is also host-failure atomic.

2.2 Fault-tolerant Bank Transfer: Implementation

We want an implementation of `transfer` to be robust against host failures and guarantee atomicity. One way to achieve this is to use write-ahead logging (WAL) combined with a recovery operation. We assume a file-system module which provides standard atomic operations to create and delete files, test their existence, and write to and read from files. Since file systems are critical, their operations have associated internal recovery operations in the event of a host failure.

Given an arbitrary program $\mathbb{C}$, we use $[\mathbb{C}]$ to identify that the program is associated with a recovery. We can now rewrite the `transfer` operation, making use of the file-system operations to implement a stylised WAL protocol as follows:

```
function transfer(from, to, amount) {
  fromAmount := getAmount(from);
  toAmount := getAmount(to);
  [create(log)] ;
  [write(log, (from, to, fromAmount, toAmount))] ;
  setAmount(from, fromAmount − amount);
  setAmount(to, toAmount + amount);
  [delete(log)] ;
}
```

The operation works by first reading the amounts stored in each account. It then creates a log file, `log`, where it stores the amounts for each account. It then updates each account, and finally deletes the log file. If a host failure occurs the log provides enough information to implement a recovery operation. In particular, its absence from the durable state means the transfer either happened or not, while its presence indicates the operation has not completed. In the latter case, we restore the initial balance by reading the log. An example of a recovery operation is the following:

```
function transferRecovery() {
  b := [exists(log)] ;
  if (b) {
    (from, to, fromAmount, toAmount) := [read(log)] ;
    if (from ≠ nil && to ≠ nil) {
      setAmount(from, fromAmount); setAmount(to, toAmount);
    }
    [delete(log)] ;
  }
}
```

The operation tests if the log file exists. If it does not, the recovery completes immediately since the balance is already consistent. Otherwise, the values of the accounts are reset to those stored in the log file which correspond to the initial balance. While the recovery operation is running, a host failure may occur, which means that upon reboot the recovery operation will run again. Eventually the recovery operation completes, at which point the transfer either occurred or did not. This guarantees that `transfer` is atomic with respect to host failures.

2.3 Fault-tolerant Bank Transfer: Verification

We introduce the following new Hoare triple for specifying programs that run in a machine where host failures can occur:

$$S \vdash \left\{ P_V \mid P_D \right\} \ \mathbb{C} \ \left\{ Q_V \mid Q_D \right\}$$

where P_V, P_D, Q_V, Q_D and S are assertions in the style of separation logic and $\mathbb{C}$ is a program. P_V and Q_V describe the volatile resource, and P_D and Q_D describe the durable resource. The judgement is read as a normal Hoare triple when there are no host failures. The interpretation of the triples is partial *resource fault avoiding* and *host failing*. Given an initial $P_V \mid P_D$, it is safe to execute $\mathbb{C}$ without causing a resource fault. If no host failure occurs, and $\mathbb{C}$ terminates, the resulting state will satisfy $Q_V \mid Q_D$. If a host failure occurs, then the durable state will satisfy the *fault-condition* S.

Given the new judgement we can describe the resulting state after a host failure. Protocols designed to make programs robust against host failures make use of the durable resource to return to a consistent state after reboot. We must be able to describe programs that have a recovery operation running after reboot. We introduce the following triple:

$$R \vdash \left\{ P_V \mid P_D \right\} \ [\mathbb{C}] \ \left\{ Q_V \mid Q_D \right\}$$

The notation $[\mathbb{C}]$ is used to identify a program with an associated recovery. The assertion R describes the durable resource after the recovery takes place.

$$\mathsf{emp} \lor \mathsf{file}(\mathtt{name}, []) \vdash \{\mathsf{emp} \mid \mathsf{emp}\}\ [\mathtt{create}(\mathtt{name})]\ \{\mathsf{emp} \mid \mathsf{file}(\mathtt{name}, [])\}$$

$$\mathsf{emp} \lor \mathsf{file}(\mathtt{name}, xs) \vdash \{\mathsf{emp} \mid \mathsf{file}(\mathtt{name}, xs)\}\ [\mathtt{delete}(\mathtt{name})]\ \{\mathsf{emp} \mid \mathsf{emp}\}$$

$$\mathsf{emp} \vdash \{\mathsf{emp} \mid \mathsf{emp}\}\ [\mathtt{exists}(\mathtt{name})]\ \{\mathsf{ret} = \mathsf{false} \land \mathsf{emp} \mid \mathsf{emp}\}$$

$$\mathsf{file}(\mathtt{name}, xs) \vdash \{\mathsf{emp} \mid \mathsf{file}(\mathtt{name}, xs)\}\ [\mathtt{exists}(\mathtt{name})]\ \{\mathsf{ret} = \mathsf{true} \land \mathsf{emp} \mid \mathsf{file}(\mathtt{name}, xs)\}$$

$$\mathsf{file}(\mathtt{name}, xs) \lor \mathsf{file}(\mathtt{name}, xs +\!\!+ [\mathtt{x}]) \vdash \begin{array}{c} \{\mathsf{emp} \mid \mathsf{file}(\mathtt{name}, xs)\} \\ [\mathtt{write}(\mathtt{name}, \mathtt{x})] \\ \{\mathsf{emp} \mid \mathsf{file}(\mathtt{name}, xs +\!\!+ [\mathtt{x}])\} \end{array}$$

$$\mathsf{file}(\mathtt{name}, []) \vdash \{\mathsf{emp} \mid \mathsf{file}(\mathtt{name}, [])\}\ [\mathtt{read}(\mathtt{name})]\ \{\mathsf{ret} = \mathsf{nil} \land \mathsf{emp} \mid \mathsf{file}(\mathtt{name}, [])\}$$

$$\mathsf{file}(\mathtt{name}, [x] +\!\!+ xs) \vdash \begin{array}{c} \{\mathsf{emp} \mid \mathsf{file}(\mathtt{name}, [x] +\!\!+ xs)\} \\ [\mathtt{read}(\mathtt{name})] \\ \{\mathsf{ret} = x \land \mathsf{emp} \mid \mathsf{file}(\mathtt{name}, [x] +\!\!+ xs)\} \end{array}$$

Fig. 1. Specification of a simplified journaling file system.

We can now use the new judgements to verify the write-ahead logging `transfer` and its recovery. In their implementation, we use a simplified journaling file system as the durable resource with the operations specified in figure 1. We specify the write-ahead logging `transfer` with the following triple:

$$S \vdash \begin{array}{c} \left\{ \dfrac{\mathtt{from} = f \land \mathtt{to} = t \land \mathtt{amount} = a \land \mathsf{emp}}{\mathsf{Account}(f, v) * \mathsf{Account}(t, w)} \right\} \\ \mathtt{transfer}(\mathtt{from}, \mathtt{to}, \mathtt{amount}) \\ \left\{ \dfrac{\mathtt{from} = f \land \mathtt{to} = t \land \mathtt{amount} = a \land \mathsf{emp}}{\mathsf{Account}(f, v - a) * \mathsf{Account}(t, w + a)} \right\} \end{array}$$

where the fault-condition S describes all the possible durable states if a host failure occurs:

$$\begin{aligned} S = {} & (\mathsf{Account}(f, v) * \mathsf{Account}(t, w)) \\ \lor {} & (\mathsf{Account}(f, v) * \mathsf{Account}(t, w) * \mathsf{file}(\mathtt{log}, [])) \\ \lor {} & (\mathsf{Account}(f, v) * \mathsf{Account}(t, w) * \mathsf{file}(\mathtt{log}, [(f, t, v, w)])) \\ \lor {} & (\mathsf{Account}(f, v - a) * \mathsf{Account}(t, w) * \mathsf{file}(\mathtt{log}, [(f, t, v, w)])) \\ \lor {} & (\mathsf{Account}(f, v - a) * \mathsf{Account}(t, w + a) * \mathsf{file}(\mathtt{log}, [(f, t, v, w)]) \\ \lor {} & (\mathsf{Account}(f, v - a) * \mathsf{Account}(t, w + a)) \end{aligned}$$

A proof that the implementation satisfies the specification is shown in figure 2. If there is a host failure, the current specification of `transfer` only guarantees that the durable resource satisfies S. This includes the case where money is lost. This is undesirable. What we want is a guarantee that the operation is atomic. In order to add this guarantee, we must combine reasoning about the operation with reasoning about its recovery to establish that undesirable states are fixed after recovery. We formalise the combination of an operation and its recovery

$$S \vdash$$

$$\left\{\frac{\mathsf{from} = f \land \mathsf{to} = t \land \mathsf{amount} = a \land \mathsf{emp}}{\mathsf{Account}(f, v) * \mathsf{Account}(t, w)}\right\}$$

$$\texttt{fromAmount} := \texttt{getAmount}(\texttt{from});$$

$$\texttt{toAmount} := \texttt{getAmount}(\texttt{to});$$

$$\left\{\frac{\mathsf{from} = f \land \mathsf{to} = t \land \mathsf{amount} = a \land \texttt{fromAmount} = v \land \texttt{toAmount} = w \land \mathsf{emp}}{\mathsf{Account}(f, v) * \mathsf{Account}(t, w))}\right\}$$

$$[\texttt{create}(\texttt{log})]\,;$$

$$\left\{\frac{\mathsf{from} = f \land \mathsf{to} = t \land \mathsf{amount} = a \land \texttt{fromAmount} = v \land \texttt{toAmount} = w \land \mathsf{emp}}{\mathsf{Account}(f, v) * \mathsf{Account}(t, w) * \mathsf{file}(\texttt{log}, [])}\right\}$$

$$[\texttt{write}(\texttt{log}, (\texttt{from}, \texttt{to}, \texttt{fromAmount}, \texttt{toAmount}))]\,;$$

$$\left\{\frac{\mathsf{from} = f \land \mathsf{to} = t \land \mathsf{amount} = a \land \texttt{fromAmount} = v \land \texttt{toAmount} = w \land \mathsf{emp}}{\mathsf{Account}(f, v) * \mathsf{Account}(t, w) * \mathsf{file}(\texttt{log}, [(f, t, v, w)])}\right\}$$

$$\texttt{setAmount}(\texttt{from}, \texttt{fromAmount} - \texttt{amount});$$

$$\left\{\frac{\mathsf{from} = f \land \mathsf{to} = t \land \mathsf{amount} = a \land \texttt{fromAmount} = v \land \texttt{toAmount} = w \land \mathsf{emp}}{\mathsf{Account}(f, v - a) * \mathsf{Account}(t, w) * \mathsf{file}(\texttt{log}, [(f, t, v, w)])}\right\}$$

$$\texttt{setAmount}(\texttt{to}, \texttt{toAmount} + \texttt{amount});$$

$$\left\{\frac{\mathsf{from} = f \land \mathsf{to} = t \land \mathsf{amount} = a \land \texttt{fromAmount} = v \land \texttt{toAmount} = w \land \mathsf{emp}}{\mathsf{Account}(f, v - a) * \mathsf{Account}(t, w + a) * \mathsf{file}(\texttt{log}, [(f, t, v, w)])}\right\}$$

$$[\texttt{delete}(\texttt{log})]\,;$$

$$\left\{\frac{\mathsf{from} = f \land \mathsf{to} = t \land \mathsf{amount} = a \land \mathsf{emp}}{\mathsf{Account}(f, v - a) * \mathsf{Account}(t, w + a)}\right\}$$

Fig. 2. Proof of transfer operation using write-ahead logging.

in order to provide robustness guarantees against host failures in the *recovery abstraction* rule:

$$\frac{\begin{array}{c}\mathbb{C}_R \textbf{ recovers } \mathbb{C} \\ S \vdash \{P_V \mid P_D\}\ \mathbb{C}\ \{Q_V \mid Q_D\} \\ S \vdash \{\mathsf{emp} \mid S\}\ \mathbb{C}_R\ \{\mathsf{true} \mid R\}\end{array}}{R \vdash \{P_V \mid P_D\}\ [\mathbb{C}]\ \{Q_V \mid Q_D\}}$$

When implementing a new operation, we use the *recovery abstraction* rule to establish the fault-condition R we wish to expose to the client. In the second premiss we must first derive what the durable resource S will be immediately after a host-failure. In the third premiss, we establish that given S, the associated recovery operation will change the durable resource to the desired R. Note that because the recovery $\mathbb{C}_R$ runs immediately after the host failure, the volatile resource of its precondition is empty. Furthermore, we require the fault-condition of the recovery to be the same as the resource that is being recovered, since the recovery operation itself may fail due to a host-failure; i.e. recovery operations must be able to recover themselves.

We allow recovery abstraction to derive any fault-condition that is established by the recovery operation. If that fault-condition is a disjunction between the durable pre- and postconditions, $P_D \lor Q_D$, then the operation $[\mathbb{C}]$ appears to be atomic with respect to host failures. Either the operation's (durable) resource updates completely, or not at all. No intermediate states are visible to the client.

In order for `transfer` to be atomic, according to the recovery abstraction rule, `transferRecovery` must satisfy the following specification:

$$
\begin{array}{ll}
& \left\{ \dfrac{\mathsf{emp}}{S} \right\} \\
S \vdash & \mathtt{transferRecovery}() \\
& \left\{ \dfrac{\mathsf{true}}{(\mathsf{Account}(f, v) * \mathsf{Account}(t, w)) \lor (\mathsf{Account}(f, v - a) * \mathsf{Account}(t, w + a))} \right\}
\end{array}
$$

The proof that the implementation satisfies this specification is given in figure 3. By applying the abstraction recovery rule we get the following specification for `transfer` which guarantees atomicity in case of a host-failure:

$$
\begin{array}{ll}
& \left\{ \dfrac{\mathtt{from} = f \land \mathtt{to} = t \land \mathtt{amount} = a \land \mathsf{emp}}{\mathsf{Account}(f, v) * \mathsf{Account}(t, w)} \right\} \\
R \vdash & [\mathtt{transfer}(\mathtt{from}, \mathtt{to}, \mathtt{amount})] \\
& \left\{ \dfrac{\mathtt{from} = f \land \mathtt{to} = t \land \mathtt{amount} = a \land \mathsf{emp}}{\mathsf{Account}(f, v - a) * \mathsf{Account}(t, w + a)} \right\}
\end{array}
$$

where the fault-condition R describes the recovered durable state:

$$R = (\mathsf{Account}(f, v) * \mathsf{Account}(t, w)) \lor (\mathsf{Account}(f, v - a) * \mathsf{Account}(t, w + a))$$

With this example, we have seen how to guarantee atomicity by logging the information required to undo operations. Advanced WAL protocols also store information allowing to redo operations and use concurrency control. We do not go into depth on how to enforce concurrency control in our examples other than the example shown in §3.1. It follows the common techniques used in concurrent separation logic.[1] However, in §4 we show ARIES, an advanced algorithm that uses write-ahead logging. A different style of write-ahead logging is used by file systems called journaling [14], which we discuss in the technical report [10].

3 Program Logic

Until now, we have only seen how to reason about sequential programs. For concurrent programs, we use resource invariants, in the style of concurrent separation logic [11], that are updated by primitive atomic operations. Here primitive atomic is used to mean that the operation takes effect at a single, discrete instant in time, and that it is atomic with respect to host failures.

The general judgement that enables us to reason about host failing concurrent programs is:

$$\boxed{J_V \mid J_D}; S \vdash \{P_V \mid P_D\} \; \mathbb{C} \; \{Q_V \mid Q_D\}$$

[1] For an introduction to concurrent separation logic see [18].

```
$S \vdash$
$\left\{\frac{\mathsf{emp}}{S}\right\}$
b := [exists(log)];
$\left\{\frac{\mathsf{b} = b \land \mathsf{emp}}{S \land (b \implies \mathsf{file}(\mathtt{log}, []) * \mathsf{true} \lor \mathsf{file}(\mathtt{log}, [(f,t,v,w)]) * \mathsf{true}) \land (\neg b \implies \mathsf{Account}(f,v) * \mathsf{Account}(t,w) \lor \mathsf{Account}(f, v-a) * \mathsf{Account}(t, w+a))}\right\}$
if (b) {
    $\left\{\frac{\mathsf{b} = b \land \mathsf{emp}}{S \land (\mathsf{file}(\mathtt{log}, []) * \mathsf{true} \lor \mathsf{file}(\mathtt{log}, [(f,t,v,w)]) * \mathsf{true})}\right\}$
    (from, to, fromAmount, toAmount) := [read(log)];
    if (from ≠ nil && to ≠ nil) {
        $\left\{\frac{\mathsf{b} = b \land \mathtt{from} = f \land \mathtt{to} = t \land \mathtt{fromAmount} = v \land \mathtt{toAmount} = w \land \mathsf{emp}}{S \land (\mathsf{file}(\mathtt{log}, [(f,t,v,w)]) * \mathsf{true})}\right\}$
        setAmount(from, fromAmount); setAmount(to, toAmount);
        $\left\{\frac{\mathsf{b} = b \land \mathtt{from} = f \land \mathtt{to} = t \land \mathtt{fromAmount} = v \land \mathtt{toAmount} = w \land \mathsf{emp}}{S \land (\mathsf{file}(\mathtt{log}, [(f,t,v,w)]) * \mathsf{true}) \land (\mathsf{Account}(f,v) * \mathsf{Account}(t,w) * \mathsf{true})}\right\}$
    }
    $\left\{\frac{\mathsf{b} = b \land \mathsf{emp}}{S \land ((\mathsf{file}(\mathtt{log}, []) * \mathsf{true}) \lor (\mathsf{file}(\mathtt{log}, [(f,t,v,w)]) * \mathsf{true})) \land (\mathsf{Account}(f,v) * \mathsf{Account}(t,w) * \mathsf{true})}\right\}$
    [delete(log)];
    $\left\{\frac{\mathsf{b} = b \land \mathsf{emp}}{\mathsf{Account}(f,v) * \mathsf{Account}(t,w)}\right\}$
}
$\left\{\frac{\mathsf{b} = b \land \mathsf{emp}}{\mathsf{Account}(f,v) * \mathsf{Account}(t,w) \lor \mathsf{Account}(f, v-a) * \mathsf{Account}(t, w+a)}\right\}$
```

Fig. 3. Proof that the transfer recovery operation guarantees atomicity.

Here, $P_V \mid P_D$ and $Q_V \mid Q_D$ are pre- and postconditions as usual and describe the volatile and durable resource. S is a durable assertion, which we refer to as the *fault-condition*, describing the durable resource of the program $\mathbb{C}$ after a host failure and possible recovery. The interpretation of these triples is partial *resource fault avoiding* and *host failing*. Starting from an initial state satisfying the precondition $P_V \mid P_D$, it is safe to execute $\mathbb{C}$ without causing a resource fault. If no host failure occurs and $\mathbb{C}$ terminates, the resulting state will satisfy the postcondition $Q_V \mid Q_D$. The shared resource invariant $J_V \mid J_D$ is maintained throughout the execution of $\mathbb{C}$. If a host failure occurs, all volatile resource is lost and the durable state will (after possible recoveries) satisfy $S * J_D$.

We give an overview of the key proof rules of Fault-tolerant Concurrent Separation Logic (FTCSL) in figure 4. Here we do not formally define the syntax of our assertions, although we describe the semantics in §5. In general, volatile and durable assertions can be parameterised by any separation algebra.

$$\textsf{sequence}$$

$$\frac{\begin{array}{c}\boxed{J_V \mid J_D}; S \vdash \left\{P_V \mid P_D\right\}\ \mathbb{C}_1\ \left\{R_V \mid R_D\right\} \\ \boxed{J_V \mid J_D}; S \vdash \left\{R_V \mid R_D\right\}\ \mathbb{C}_2\ \left\{Q_V \mid Q_D\right\}\end{array}}{\boxed{J_V \mid J_D}; S \vdash \left\{P_V \mid P_D\right\}\ \mathbb{C}_1; \mathbb{C}_2\ \left\{Q_V \mid Q_D\right\}}$$

$$\textsf{consequence}$$

$$\frac{\begin{array}{c}P_V \mid P_D \implies P'_V \mid P'_D \quad Q'_V \mid Q'_D \implies Q_V \mid Q_D \quad S' \implies S \\ \boxed{J_V \mid J_D}; S' \vdash \left\{P'_V \mid P'_D\right\}\ \mathbb{C}\ \left\{Q'_V \mid Q'_D\right\}\end{array}}{\boxed{J_V \mid J_D}; S \vdash \left\{P_V \mid P_D\right\}\ \mathbb{C}\ \left\{Q_V \mid Q_D\right\}}$$

$$\textsf{frame}$$

$$\frac{\boxed{J_V \mid J_D}; S \vdash \left\{P_V \mid P_D\right\}\ \mathbb{C}\ \left\{Q_V \mid Q_D\right\}}{\boxed{J_V \mid J_D}; S * R_D \vdash \left\{P_V * R_V \mid P_D * R_D\right\}\ \mathbb{C}\ \left\{Q_V * R_V \mid Q_D * R_D\right\}}$$

$$\textsf{atomic}$$

$$\frac{\boxed{\textsf{emp} \mid \textsf{emp}}; (P_D * J_D \vee Q_D * J_D) \vdash \left\{P_V * J_V \mid P_D * J_D\right\}\ \mathbb{C}\ \left\{Q_V * J_V \mid Q_D * J_D\right\}}{\boxed{J_V \mid J_D}; (P_D \vee Q_D) \vdash \left\{P_V \mid P_D\right\}\ \langle\mathbb{C}\rangle\ \left\{Q_V \mid Q_D\right\}}$$

$$\textsf{share}$$

$$\frac{\boxed{J_V * R_V \mid J_D * R_D}; S \vdash \left\{P_V \mid P_D\right\}\ \mathbb{C}\ \left\{Q_V \mid Q_D\right\}}{\boxed{J_V \mid J_D}; S * R_D \vdash \left\{P_V * R_V \mid P_D * R_D\right\}\ \mathbb{C}\ \left\{Q_V * R_V \mid Q_D * R_D\right\}}$$

$$\textsf{parallel}$$

$$\frac{\begin{array}{c}\boxed{J_V \mid J_D}; S_1 \vdash \left\{P_{V1} \mid P_{D1}\right\}\ \mathbb{C}_1\ \left\{Q_{V1} \mid Q_{D1}\right\} \\ \boxed{J_V \mid J_D}; S_2 \vdash \left\{P_{V2} \mid P_{D2}\right\}\ \mathbb{C}_2\ \left\{Q_{V2} \mid Q_{D2}\right\}\end{array}}{\boxed{J_V \mid J_D}; (S_1 * S_2) \vee (S_1 * Q_{D2}) \vee (Q_{D1} * S_2) \vdash \begin{array}{c}\left\{P_{V1} * P_{V2} \mid P_{D1} * P_{D2}\right\} \\ \mathbb{C}_1 \mid\mid \mathbb{C}_2 \\ \left\{Q_{V1} * Q_{V2} \mid Q_{D1} * Q_{D2}\right\}\end{array}}$$

$$\textsf{recovery abstraction}$$

$$\frac{\begin{array}{c}\mathbb{C}_R\ \textbf{recovers}\ \mathbb{C} \\ \boxed{J_V \mid J_D}; S \vdash \left\{P_V \mid P_D\right\}\ \mathbb{C}\ \left\{Q_V \mid Q_D\right\} \\ \boxed{\textsf{emp} \mid J_D}; S \vdash \left\{\textsf{emp} \mid S\right\}\ \mathbb{C}_R\ \left\{\textsf{true} \mid R\right\}\end{array}}{\boxed{J_V \mid J_D}; R \vdash \left\{P_V \mid P_D\right\}\ [\mathbb{C}]\ \left\{Q_V \mid Q_D\right\}}$$

Fig. 4. Selected proof rules of FTCSL

The *sequence rule* allows us to combine two programs in sequence as long as they have the same fault-condition and resource invariant. Typically, when the fault-conditions differ, we can weaken them using the *consequence rule*, which adds fault-condition weakening to the standard consequence rule of Hoare logic. The *frame rule*, as in separation logic, allows us to extend the pre- and postconditions with the same unmodified resource $R_V * R_D$. However, here the durable part, R_D, is also added to the fault-condition.

The *atomic rule* allows us to use the resource invariant $J_V \mid J_D$ using a primitive atomic operation. Since the operation executes in a single, discrete, moment in time, we can think of the operation temporarily owning the resources $J_V \mid J_D$. However, they must be reestablished at the end. This guarantees that the every primitive atomic operation maintains the resource invariant. Note that the rule enforces atomicity with respect to host failures. The *share rule* allows us to use local resources to extend the shared resource invariant.

The *parallel rule*, in terms of pre- and postconditions is as in concurrent separation logic. However, the fault-condition describes the possible durable resources that may result from a host failure while running $\mathbb{C}_1$ and $\mathbb{C}_2$ in parallel. In particular, a host-failure may occur while both $\mathbb{C}_1$ and $\mathbb{C}_2$ are running, in which case the fault-condition is $S_1 * S_2$, or when either one of $\mathbb{C}_1$, $\mathbb{C}_2$ has finished, in which case the fault-condition is $S_1 * Q_{D2}$ and $S_2 * Q_{D1}$ respectively.

Finally, the *recovery abstraction rule* allows us to prove that a recovery operation $\mathbb{C}_R$ establishes the fault-condition R we wish to expose to the client. The first premiss requires operation $\mathbb{C}_R$ to be the recovery of $\mathbb{C}$, i.e. it is executed on reboot after a host failure during execution of $\mathbb{C}$. The second premiss guarantees that in such case, the durable resources satisfy S and the shared resource invariant satisfies J_D, while the volatile state is lost after a host failure. The third premiss, takes the resource after the reboot and runs the recovery operation in order to establish R. Note that J_D is an invariant, as there can be potentially parallel recovery operations accessing it using primitive atomic operations. While the recovery operation $\mathbb{C}_R$ is running, there can be any number of host failures, which restart the recovery. This means that the recovery operation must be able to recover from itself. We allow recovery abstraction to derive any fault-condition that is established by the recovery operation. If the fault-condition is a disjunction between the durable pre- and post-conditions, $P_V \vee Q_D$, then the operation $[\mathbb{C}]$ appears to be atomic with respect to host failures.

3.1 Example: Concurrent Bank Transfer

Consider two threads that both perform a transfer operation from account f to account t as shown in §2. The parallel rule requires that each operation acts on disjoint resources in the precondition. Since both threads update the same accounts, we synchronise their use with the atomic blocks denoted by $\langle \_ \rangle$. A possible specification for the program is the following:

$$
\begin{array}{c}
\boxed{\mathsf{emp} \mid \mathsf{emp}}\,; (\exists v, w.\, \mathsf{Account}(f, v) * \mathsf{Account}(t, w)) \vdash \\
\left\{ \dfrac{\mathtt{from} = f \wedge \mathtt{to} = t \wedge \mathtt{amount} = a \wedge \mathtt{amount2} = b \wedge \mathsf{emp}}{\exists v, w.\, \mathsf{Account}(f, v) * \mathsf{Account}(t, w)} \right\} \\
\langle [\mathtt{transfer}(\mathtt{from}, \mathtt{to}, \mathtt{amount})] \rangle; \;\|\; \langle [\mathtt{transfer}(\mathtt{from}, \mathtt{to}, \mathtt{amount2})] \rangle; \\
\left\{ \dfrac{\mathtt{from} = f \wedge \mathtt{to} = t \wedge \mathtt{amount} = a \wedge \mathtt{amount2} = b \wedge \mathsf{emp}}{\exists v, w.\, \mathsf{Account}(f, v) * \mathsf{Account}(t, w)} \right\}
\end{array}
$$

A sketch proof of this specification is given in figure 5. We first move the shared resources of the two `transfer` operations to the shared invariant (share rule). We then prove each thread independently by making use of the atomic rule to gain temporary access to the shared invariant within the atomic block, and reuse the specification given in §2.3. It is possible to get stronger postconditions, that maintain exact information about the amounts of each bank account, using complementary approaches such as Owicki-Gries or other forms of resource ownership [18]. The sequential examples in this paper can be adapted to concurrent applications using these techniques.

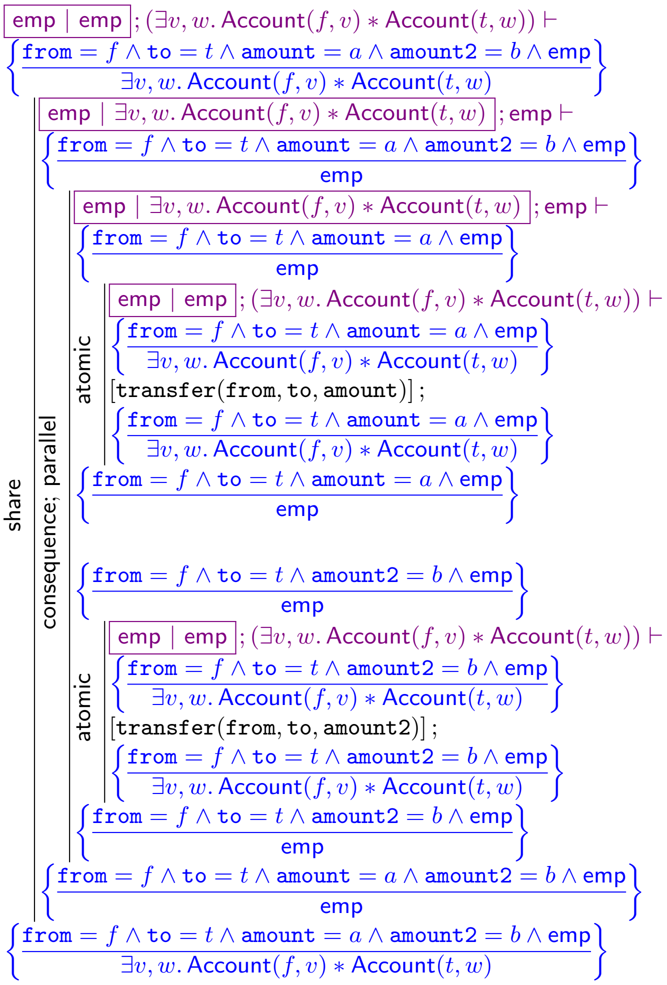
share
consequence; parallel
atomic
atomic
$\boxed{\mathsf{emp} \mid \mathsf{emp}}\,; (\exists v, w.\, \mathsf{Account}(f, v) * \mathsf{Account}(t, w)) \vdash$
$\left\{\dfrac{\texttt{from} = f \land \texttt{to} = t \land \texttt{amount} = a \land \texttt{amount2} = b \land \mathsf{emp}}{\exists v, w.\, \mathsf{Account}(f, v) * \mathsf{Account}(t, w)}\right\}$
$\boxed{\mathsf{emp} \mid \exists v, w.\, \mathsf{Account}(f, v) * \mathsf{Account}(t, w)}\,; \mathsf{emp} \vdash$
$\left\{\dfrac{\texttt{from} = f \land \texttt{to} = t \land \texttt{amount} = a \land \texttt{amount2} = b \land \mathsf{emp}}{\mathsf{emp}}\right\}$
$\boxed{\mathsf{emp} \mid \exists v, w.\, \mathsf{Account}(f, v) * \mathsf{Account}(t, w)}\,; \mathsf{emp} \vdash$
$\left\{\dfrac{\texttt{from} = f \land \texttt{to} = t \land \texttt{amount} = a \land \mathsf{emp}}{\mathsf{emp}}\right\}$
$\boxed{\mathsf{emp} \mid \mathsf{emp}}\,; (\exists v, w.\, \mathsf{Account}(f, v) * \mathsf{Account}(t, w)) \vdash$
$\left\{\dfrac{\texttt{from} = f \land \texttt{to} = t \land \texttt{amount} = a \land \mathsf{emp}}{\exists v, w.\, \mathsf{Account}(f, v) * \mathsf{Account}(t, w)}\right\}$
$[\texttt{transfer}(\texttt{from}, \texttt{to}, \texttt{amount})]\,;$
$\left\{\dfrac{\texttt{from} = f \land \texttt{to} = t \land \texttt{amount} = a \land \mathsf{emp}}{\exists v, w.\, \mathsf{Account}(f, v) * \mathsf{Account}(t, w)}\right\}$
$\left\{\dfrac{\texttt{from} = f \land \texttt{to} = t \land \texttt{amount} = a \land \mathsf{emp}}{\mathsf{emp}}\right\}$
$\left\{\dfrac{\texttt{from} = f \land \texttt{to} = t \land \texttt{amount2} = b \land \mathsf{emp}}{\mathsf{emp}}\right\}$
$\boxed{\mathsf{emp} \mid \mathsf{emp}}\,; (\exists v, w.\, \mathsf{Account}(f, v) * \mathsf{Account}(t, w)) \vdash$
$\left\{\dfrac{\texttt{from} = f \land \texttt{to} = t \land \texttt{amount2} = b \land \mathsf{emp}}{\exists v, w.\, \mathsf{Account}(f, v) * \mathsf{Account}(t, w)}\right\}$
$[\texttt{transfer}(\texttt{from}, \texttt{to}, \texttt{amount2})]\,;$
$\left\{\dfrac{\texttt{from} = f \land \texttt{to} = t \land \texttt{amount2} = b \land \mathsf{emp}}{\exists v, w.\, \mathsf{Account}(f, v) * \mathsf{Account}(t, w)}\right\}$
$\left\{\dfrac{\texttt{from} = f \land \texttt{to} = t \land \texttt{amount2} = b \land \mathsf{emp}}{\mathsf{emp}}\right\}$
$\left\{\dfrac{\texttt{from} = f \land \texttt{to} = t \land \texttt{amount} = a \land \texttt{amount2} = b \land \mathsf{emp}}{\mathsf{emp}}\right\}$
$\left\{\dfrac{\texttt{from} = f \land \texttt{to} = t \land \texttt{amount} = a \land \texttt{amount2} = b \land \mathsf{emp}}{\exists v, w.\, \mathsf{Account}(f, v) * \mathsf{Account}(t, w)}\right\}$

Fig. 5. Sketch proof of two concurrent transfers over the same accounts.

4 Case Study: ARIES

In §2 we saw an example of a very simple transaction and its associated recovery operation employing write-ahead logging. Relational databases support concurrent execution of complex transactions following the established ACID (Atomicity, Consistency, Isolation and Durability) set of properties. ARIES (Algorithms for Recovery and Isolation Exploiting Semantics) [8], is a collection of algorithms involving, concurrent execution, write-ahead-logging and failure recovery of transactions, that is widely-used to establish ACID properties.

It is beyond the scope of this paper to verify that the full set of ARIES algorithms guarantees ACID properties. Instead, we focus on a stylised version of the recovery algorithm of ARIES proving that: a) it is idempotent with respect to host failures, b) after recovery, all transactions recorded in the write-ahead log have either been completed, or were rolled-back.

Transactions update database records stored in durable memory, which for the purposes of this discussion we assume to be a single file in a file system. To increase performance, the database file is divided into fixed-size blocks, called pages, containing multiple records. Thus input/output to the database file, instead of records, is in terms of pages, which are also typically cached in volatile memory. A single transaction may update multiple pages. In the event of a host failure, there may be transactions that have not yet completed, or have completed but their updated pages have not yet been written back to the database file.

ARIES employs write-ahead logging for page updates performed by transactions. The log is stored on a durable fault-tolerant medium. The recovery uses the logged information in a sequence of three phases. First, the *analysis phase*, scans the log to determine the (volatile) state, of any active transactions (committed or not), at the point of host failure. Next, the *redo phase*, scans the log and redos each logged page update, unless the associated page in the database file is already updated. Finally, the *undo phase*, scans the log and undos each page update for each uncommitted transaction. To cope with a possible host failure during the ARIES recovery, each undo action is logged beforehand. Thus, in the event of a host failure the undo actions will be retried as part of the redo phase.

In figure 6, we define the log and database model and describe the predicates we use in our specifications and proofs. We refer the reader to our technical report [10] for the formal definitions. We model the database state, db, as a set of pages, where each page comprises the page identifier, the log sequence number (defined later) of the last update performed on the page, and the page data. The log, lg, is structured as a sequence of log records, ordered by a *log sequence number*, $lsn \in \mathbb{N}$, each of which records a particular action performed by a transaction. The ordering follows the order in which transaction actions are performed on the database. The logged action, $U[tid, pid, op]$, records that the transaction identifier tid, performs the update $op : \textsc{Data} \to \textsc{Data}$ on the page identified by pid. We use op^{-1} to denote the operation undoing the update op. $B[tid]$, records the start of a new transaction with identifier tid, and $C[tid]$, records that the transaction with id tid is committed. The information from the above actions is used to construct two auxiliary structures used by the recovery to determine the state of transactions and pages at the point of a host failure. The *transaction table* (TT), records the status of all active transactions (e.g. updating, committed) and the latest log sequence number associated with the transaction. The *dirty page table* (DPT), records which pages are modified but yet unwritten to the database together with the first log sequence number of the action that caused the first modification to each page. To avoid the cost of scanning the entire log, implementations regularly log snapshots of the TT and DPT in checkpoints, $CHK[tt, dpt]$. For simplicity, we assume the log contains exactly one checkpoint.

Let $lsn, tid, pid \in \mathbb{N}$, where we use lsn for log sequence numbers, tid for transaction identifiers, pid for page identifiers, d for page data and op for page-update operations. Let $\varnothing$ be an empty list.

Model:

Database state	$db \subseteq \mathbb{N} \times \mathbb{N} \times \textsc{Data}$, triples of pid, lsn, d
Logged actions	$act ::= U[tid, pid, op] \mid B[tid] \mid C[tid] \mid CHK[tt, dpt]$
Log state	$lg ::= \varnothing \mid (lsn, act) \mid lg \otimes lg$
Transaction table	$tt \subseteq \mathbb{N} \times \mathbb{N} \times \{C, U\}$, triples of lsn, pid and transaction status
Dirty page table	$dpt \subseteq \mathbb{N} \times \mathbb{N}$, tuples of pid, lsn

Predicates:

$\mathsf{log}\,(lg)$	the state of the log is given by lg (abstract predicate)
$\mathsf{db\_state}\,(db)$	the state of the database is given by db (abstract predicate)
$\mathsf{set}\,(\mathtt{x}, s)$	the set s identified by program variable $\mathtt{x}$ (abstract predicate)
$\mathsf{log\_tt}\,(lg, tt)$	log lg produces the TT entries in tt
$\mathsf{log\_dpt}\,(lg, dpt)$	log lg produces the DPT entries in dpt
$\mathsf{log\_rl}\,(lg, dpt, ops)$	given log lg and DPT dpt the list of redo updates is ops
$\mathsf{ul\_undo}\,(lg, tt, ops)$	given log lg and TT tt the list of undo updates is ops
$\mathsf{log\_undos}\,(ops, lg)$	given list of undos ops the additional log records are lg
$\mathsf{db\_acts}\,(db, ops, db')$	given the list of updates ops, the database db is updated to db'
$\mathsf{recovery\_log}\,(lg, lg')$	given log lg log records added by recovery are lg'
$\mathsf{recovery\_db}\,(db, lg, db)$	given database db and log lg the recovered database state is db'

Axioms:

$$\mathsf{log}\left(lg \otimes lg'\right) \iff \mathsf{log\_bseg}\,(lg) \otimes \mathsf{log\_fseg}\left(lg'\right)$$

Fig. 6. Abstract model of the database and ARIES log, and predicates.

The high level overview of the recovery algorithm in terms of its analysis, redo and undo phases is given in figure 7. The analysis phase first finds the checkpoint and restores the TT and DPT. Then, it proceeds to scan the log forwards from the checkpoint, updating the TT and DPT. Any new transaction is added to the TT. For any commit log record we update the TT to record that the transaction is committed. For any update log record, we add an entry for the associated page to the DPT, also recording the log sequence number, unless an entry for the same

```
function aries_recovery() {
  //ANALYSIS PHASE: restore dirty page table, transaction table
  //and undo list at point of host failure.
  tt, dpt := aries_analyse();
  //REDO PHASE: repeat actions to restore database state at host failure.
  aries_redo(dpt);
  //UNDO PHASE: Undo actions of uncommitted transactions.
  aries_undo(tt);
}
```

Fig. 7. ARIES recovery: high level structure.

page is already in it. We give the following specification for the analysis phase:

$$\mathsf{log}\,(lg_i \otimes (lsn, CHK[tt, dpt]) \otimes lg_c\ ) \vdash$$
$$\left\{\frac{\mathsf{emp}}{\mathsf{log}\,(lg_i \otimes (lsn, CHK[tt, dpt]) \otimes lg_c\ )}\right\}$$
$$\texttt{tt}, \texttt{dpt} := \texttt{aries\_analyse}()$$
$$\left\{\frac{\exists tt', dpt'.\ \mathsf{log\_tt}\,(lg_c, tt') \wedge \mathsf{log\_dpt}\,(lg_c, dpt') \wedge \mathsf{set}\,(\texttt{tt}, tt \oplus tt') * \mathsf{set}\,(\texttt{dpt}, dpt \uplus dpt')}{\mathsf{log}\,(lg_i \otimes (-, CHK[tt, dpt]) \otimes lg_c\ )}\right\}$$

The specification states that given the database log, the TT and DPT in the log's checkpoint are restored and updated according to the log records following the checkpoint. The analysis does not modify any durable state.

The redo phase, follows analysis and repeats the logged updates. Specifically, redo scans the log forward from the record with the lowest sequence number in the DPT. This is the very first update that is logged, but (potentially) not yet written to the database. The updates are redone unless the recorded page associated with that update is not present in the DPT, or a more recent update has modified it. We give the following specification to redo:

$$\exists ops, ops', ops''.\ (ops = ops' \otimes ops'') \wedge \mathsf{db\_acts}\,(db, ops', db'')$$
$$\wedge\ \mathsf{log\_fseg}\,((lsn, act) \otimes lg) * \mathsf{db\_state}\,(db'') \vdash$$
$$\left\{\frac{\mathsf{set}\,(\texttt{dpt}, dpt) \wedge lsn = min(dpt_{\downarrow 2})}{\mathsf{log\_fseg}\,((lsn, act) \otimes lg) * \mathsf{db\_state}\,(db)}\right\}$$
$$\texttt{aries\_redo}(\texttt{dpt})$$
$$\left\{\frac{\mathsf{set}\,(\texttt{dpt}, dpt) \wedge lsn = min(dpt_{\downarrow 2})}{\begin{array}{c}\mathsf{log\_fseg}\,((lsn, act) \otimes lg) * \mathsf{db\_state}\,(db') \wedge \mathsf{db\_acts}\,(db, ops, db') \\ \wedge\ \mathsf{log\_rl}\,((lsn, act) \otimes lg, dpt, ops)\end{array}}\right\}$$

The specification states that the database is updated according to the logged update records following the smallest log sequence number in the DPT. The fault-condition specifies that after a host failure, all, some or none of the redos have happened. Since redo does not log anything, the log is not affected.

The last phase is undo, which reverts the updates of any transaction that is not committed. In particular, undo scans the log backwards from the log record with the largest log sequence number in the TT. This is the log sequence number of the very last update. For each update record scanned, if the transaction exists in the TT and is not marked as committed, the update is reversed. However, each reverting update is logged beforehand. This ensures, that undos will happen even in case of host failure, since they will be re-done in the redo phase of the subsequent recovery run. We give the following specification for the undo phase:

$$\begin{gathered}
\exists lg', lg'', lg''', ops, ops', ops''.\, lg' = lg'' \otimes lg''' \wedge ops = ops' \otimes ops'' \\
\wedge\, \mathsf{db\_acts}\,(db, ops', db'') \wedge \mathsf{log\_bseg}\,(lg \otimes (lsn, act) \otimes lg'') * \mathsf{db\_state}\,(db'') \vdash \\
\left\{\frac{\mathsf{set}\,(\mathtt{tt}, tt) \wedge lsn = max(tt_{\downarrow 2})}{\mathsf{log\_bseg}\,(lg \otimes (lsn, act)) * \mathsf{db\_state}\,(db)}\right\} \\
\mathtt{aries\_undo}(\mathtt{tt}) \\
\left\{\frac{\mathsf{set}\,(\mathtt{tt}, tt) \wedge lsn = max(tt_{\downarrow 2}) \wedge \mathsf{ul\_undo}\,(tt, lg \otimes (lsn, act), ops)}{\begin{gathered}\mathsf{log\_bseg}\,(lg \otimes (lsn, act) \otimes lg') \wedge \mathsf{log\_undos}\,(ops, lg') \\ * \,\mathsf{db\_state}\,(db') \wedge \mathsf{db\_acts}\,(db, ops, db')\end{gathered}}\right\}
\end{gathered}$$

The specification states that the database is updated with actions reverting previous updates as obtained from the log. These undo actions are themselves logged. In the event of a host failure the fault-condition specifies that all, some, or none of the operations are undone and logged.

Using the specification for each phase and using our logic we can derive the following specification for this ARIES recovery algorithm:

$$\begin{gathered}
\exists lg', lg'', db'.\, \mathsf{log}\,(lg \otimes (lsn, CHK[tt, dpt]) \otimes lg') * \mathsf{db\_state}\,(db) \vdash \\
\left\{\frac{\mathsf{emp}}{\mathsf{log}\,(lg \otimes (lsn, CHK[tt, dpt]) \otimes lg') * \mathsf{db\_state}\,(db)}\right\} \\
\mathtt{aries\_recovery}() \\
\left\{\frac{\mathsf{true}}{\begin{gathered}\mathsf{log}\,(lg \otimes (lsn, CHK[tt, dpt]) \otimes lg' \otimes lg'') \\ \wedge\, \mathsf{recovery\_log}\,(lg \otimes (lsn, CHK[tt, dpt]) \otimes lg', lg'') \\ * \,\mathsf{db\_state}\,(db') \wedge \mathsf{recovery\_db}\,(db, lg \otimes (lsn, CHK[tt, dpt]) \otimes lg', db')\end{gathered}}\right\}
\end{gathered}$$

The proof that the high level structure of the ARIES algorithm satisfies this specification is given in figure 8. For the implementations of each phase and proofs they meet their specifications we refer the reader to our technical report[10]. The key property of the ARIES recovery specification is that the durable precondition is the same as the fault-condition. This guarantees that the recovery is idempotent with respect to host failures. This is crucial for any recovery operation, as witnessed in the recovery abstraction rule, guaranteeing that the recovery itself is robust against crashes. Furthermore, the specification states that any transaction logged as committed at the time of host failure, is committed after recovery. Otherwise transactions are rolled back.

5 Semantics and Soundness

We give a brief overview of the semantics of our reasoning and the intuitions behind its soundness. A detailed account is given in the technical report [10].

5.1 Fault-tolerant Views

We define a general fault-tolerant reasoning framework using Hoare triples with fault-conditions in the style of the Views framework [3]. Pre- and postcondition

```
$\exists lg', lg', db, rl'.\, \mathsf{log}\,(lg \otimes (lsn, CHK[tt, dpt]) \otimes lg') * \mathsf{db\_state}\,(db) \vdash$
$\{\mathsf{emp} \mid \mathsf{log}\,(lg \otimes (lsn, CHK[tt, dpt]) \otimes lg') * \mathsf{db\_state}\,(db)\}$
sequence
  frame
    //ANALYSIS PHASE
    $\mathsf{log}\,(lg \otimes (lsn, CHK[tt, dpt]) \otimes lg') \vdash$
    $\{\mathsf{emp} \mid \mathsf{log}\,(lg \otimes (lsn, CHK[tt, dpt]) \otimes lg')\}$
    tt, dpt := aries_analyse();
    $\left\{\frac{\exists tt', dpt'.\, \mathsf{log\_tt}\,(lg', tt') \land \mathsf{log\_dpt}\,(lg', dpt, dpt') \land \mathsf{set}\,(\mathtt{tt}, tt \oplus tt') * \mathsf{set}\,(\mathtt{dpt}, dpt \uplus dpt')}{\mathsf{log}\,(lg \otimes (-, CHK[tt, dpt]) \otimes lg')}\right\}$
  $\left\{\frac{\exists tt', dpt'.\, \mathsf{log\_tt}\,(lg', tt') \land \mathsf{log\_dpt}\,(lg', dpt, dpt') \land \mathsf{set}\,(\mathtt{tt}, tt \oplus tt') * \mathsf{set}\,(\mathtt{dpt}, dpt \uplus dpt')}{\mathsf{log}\,(lg \otimes (-, CHK[tt, dpt]) \otimes lg') * \mathsf{db\_state}\,(db)}\right\}$
  //REDO PHASE: repeat actions to restore database state at host failure.
  consequence
    $\exists lg_i, lg_c, lg, lg', db, db', db'', lsn_{\leq}, act, ops', ops''.\, (ops = ops' \otimes ops'') \land$
    $\mathsf{log\_bseg}\,(lg_i) * \mathsf{log\_fseg}\,((lsn_{\leq}, act) \otimes lg_c) * \mathsf{db\_state}\,(db'') \land \mathsf{db\_acts}\,(db, ops', db'') \vdash$
    $\left\{\frac{\exists tt', dpt'.\, lg \otimes (lsn, CHK[tt, dpt]) \otimes lg' = lg_i \otimes (lsn_{\leq}, act) \otimes lg_c \land lsn_{\leq} = min((dpt \uplus dpt')_{\downarrow 2}) \land \mathsf{log\_tt}\,(lg', tt') \land \mathsf{log\_dpt}\,(lg', dpt, dpt') \land \mathsf{log\_ul}\,(lg', ul) \land \mathsf{set}\,(\mathtt{tt}, tt \oplus tt') * \mathsf{set}\,(\mathtt{dpt}, dpt \uplus dpt')}{\mathsf{log\_bseg}\,(lg_i) * \mathsf{log\_fseg}\,((lsn_{\leq}, act) \otimes lg_c) * \mathsf{db\_state}\,(db)}\right\}$
    frame
      $\exists db'', ops', ops''.\, (ops = ops' \otimes ops'')$
      $\land \mathsf{log\_fseg}\,((lsn_{\leq}, act) \otimes lg) * \mathsf{db\_state}\,(db'') \land \mathsf{db\_acts}\,(db, ops', db'') \vdash$
      $\left\{\frac{\mathsf{set}\,(\mathtt{dpt}, dpt_u) \land lsn_{\leq} = min((dpt_u)_{\downarrow 2})}{\mathsf{log\_fseg}\,((lsn_{\leq}, act) \otimes lg_c) * \mathsf{db\_state}\,(db)}\right\}$
      aries_redo(dpt);
      $\left\{\frac{\mathsf{set}\,(\mathtt{dpt}, dpt_u) \land lsn_{\leq} = min((dpt_u)_{\downarrow 2})}{\mathsf{log\_fseg}\,((lsn_{\leq}, act) \otimes lg_c) * \mathsf{db\_state}\,(db') \land \mathsf{db\_acts}\,(db, ops, db') \land \mathsf{log\_rl}\,((lsn_{\leq}, act) \otimes lg_c, dpt_u, ops)}\right\}$
    $\left\{\frac{lg \otimes (lsn, CHK[tt, dpt]) \otimes lg' = lg_i \otimes (lsn_{\leq}, act) \otimes lg_c \land lsn_{\leq} = min((dpt \uplus dpt')_{\downarrow 2}) \land \mathsf{log\_tt}\,(lg', tt') \land \mathsf{log\_dpt}\,(lg', dpt, dpt') \land \mathsf{log\_ul}\,(lg', ul) \land \mathsf{set}\,(\mathtt{tt}, tt \oplus tt') * \mathsf{set}\,(\mathtt{dpt}, dpt \uplus dpt') * \mathsf{ulist}\,(\mathtt{ul}, ul)}{\mathsf{log\_bseg}\,(lg_i) * \mathsf{log\_fseg}\,((lsn_{\leq}, act) \otimes lg_c) * \mathsf{db\_state}\,(db') \land \mathsf{log\_rl}\,((lsn_{\leq}, act) \otimes lg_c, dpt_u, ops)}\right\}$
  $\left\{\frac{\exists lg_c, lsn_{\leq}.\, lg' = - \otimes (lsn_{\leq}, act) \otimes lg_c \land lsn_{\leq} = min((dpt \uplus dpt')_{\downarrow 2}) \land \mathsf{log\_tt}\,(lg', tt') \land \mathsf{log\_dpt}\,(lg', dpt, dpt') \land \mathsf{log\_ul}\,(lg', ul) \land \mathsf{set}\,(\mathtt{tt}, tt \oplus tt') * \mathsf{set}\,(\mathtt{dpt}, dpt \uplus dpt')}{\mathsf{log}\,(lg \otimes (lsn, CHK[tt, dpt]) \otimes lg') * \mathsf{db\_state}\,(db') \land \mathsf{db\_acts}\,(db, ops, db') \land \mathsf{log\_rl}\,((lsn_{\leq}, act) \otimes lg_c, dpt \uplus dpt', rl')}\right\}$
  consequence,frame
    //UNDO PHASE: Undo actions of uncommitted transactions.
    $\exists lg', lg'', lg''', ops, ops', ops''.\, lg' = lg'' \otimes lg''' \land ops = ops' \otimes ops''$
    $\land \mathsf{db\_acts}\,(db_r, ops', db''_r) \land \mathsf{log\_bseg}\,(lg \otimes (lsn, act) \otimes lg'') * \mathsf{db\_state}\,(db''_r) \vdash$
    $\{\mathsf{set}\,(\mathtt{tt}, tt) \land lsn_{\geq} = max(tt_{\downarrow 2}) \mid \mathsf{log\_bseg}\,(lg \otimes (lsn, act)) * \mathsf{db\_state}\,(db_r)\}$
    aries_undo(tt, dpt, ul);
    $\left\{\frac{\mathsf{set}\,(\mathtt{tt}, tt) \land lsn = max(tt_{\downarrow 2}) \land \mathsf{ul\_undo}\,(tt, lg \otimes (lsn, act), ops)}{\mathsf{log\_bseg}\,(lg \otimes (lsn, act) \otimes lg') \land \mathsf{log\_undos}\,(ops, lg') * \mathsf{db\_state}\,(db'_r) \land \mathsf{db\_acts}\,(db_r, ops, db'_r)}\right\}$
$\left\{\frac{\mathsf{true}}{\mathsf{log}\,(lg \otimes (lsn, CHK[tt, dpt]) \otimes lg' \otimes lg'') \land \mathsf{recovery\_log}\,(lg \otimes (lsn, CHK[tt, dpt]) \otimes lg', lg'') * \mathsf{db\_state}\,(db') \land \mathsf{recovery\_db}\,(db, lg \otimes (lsn, CHK[tt, dpt]) \otimes lg', db')}\right\}$
```

Fig. 8. Proof of the high level structure of ARIES recovery.

assertions are modelled as pairs of *volatile* and *durable views* (commutative monoids). Fault-condition assertions are modelled as durable views [2]. Volatile and durable views provide partial knowledge reified to concrete volatile and durable program states respectively. Concrete volatile states include the distinguished host-failed state $\lightning$. The semantic interpretation of a primitive operation is given as a state transformer function from concrete states to sets of concrete states.

To prove soundness, we encode our Fault-tolerant Views (FTV) framework into Views [3]. A judgement[3] $s \vdash \big\{(p_v, p_d)\big\} \; \mathbb{C} \; \big\{(q_v, q_d)\big\}$, where s, p_d, q_d are durable views and p_v, q_v are volatile views is encoded as the Views judgement: $\big\{(p_v, q_d)\big\} \; \mathbb{C} \; \big\{(q_v, q_d) \vee (\lightning, s)\big\}$, where volatile views are extended to include $\lightning$ and $\vee$ is disjunction of views. For the general abstraction recovery rule we encode $[\mathbb{C}]$ as a program which can test for host failures, beginning with $\mathbb{C}$ and followed by as many iterations of the recovery $\mathbb{C}_R$ as required in case of a host failure.

We require the following properties for a sound instance of the framework:

Host failure: For each primitive operation, its interpretation function must transform non host-failed states to states including a host-failed state. This guarantees that each operation can be abruptly interrupted by a host failure.

Host failure propagation: For each primitive operation, its interpretation function must leave all host-failed states intact. That is, when the state says there is a host failure, it stays a host failure.

Axiom soundness: The axiom soundness property (property [G] of Views [3]).

The first two are required to justify the general FTV rules, while the final property establishes soundness of the Views encoding itself. When all the parameters are instantiated and the above properties established then the instantiation of the framework is sound.

5.2 Fault-tolerant Concurrent Separation Logic

We justify the soundness of FTCSL by an encoding into the Fault-tolerant Views framework discussed earlier. The encoding is similar to the concurrent separation logic encoding into Views. We instantiate volatile and durable views as pairs of local views and shared invariants.

The FTCSL judgement $\boxed{(j_v, j_d)}\, ; s \vdash \big\{(p_v, p_d)\big\} \; \mathbb{C} \; \big\{(q_v, q_d)\big\}$ is encoded as:

$$s \vdash \big\{((p_v, j_v), (p_d, j_d))\big\} \; \mathbb{C} \; \big\{((q_v, j_v), (q_d, j_d))\big\}$$

The proof rules in figure 4 are justified by soundness of the encoding and simple application of FTV proof rules. Soundness of the encoding is established by proving the properties stated in §5.1.

Theorem 1 (FTCSL Soundness).
If the judgement $\boxed{J_V \mid J_D}\, ; S \vdash \big\{P_V \mid P_D\big\} \; \mathbb{C} \; \big\{Q_V \mid Q_D\big\}$ *is derivable in the*

[2] We use "Views" to refer to the Views framework of Dinsdale-Young et al. [3], and "views" to refer to the monoid structures used within it.

[3] Note that judgements, such as those in figure 4, using assertions (capital P, Q, S) are equivalent to judgements using views (models of assertions, little p, q, s).

program logic, then if we run the program $\mathbb{C}$ from state satisfying $P_V * J_V \mid P_D * J_D$, then $\mathbb{C}$ will either not terminate, or terminate in state satisfying $Q_V * J_V \mid Q_D * J_D$, or a host failure will occur destroying any volatile state and the remaining durable state (after potential recoveries) will satisfy $S * J_D$. The resource invariant $J_V \mid J_D$ holds throughout the execution of $\mathbb{C}$.

6 Related Work

There has been a significant amount of work in critical systems, such as file systems and databases, to develop defensive methods against the types of failures covered in this paper [14,19,1,8]. The verification of these techniques has mainly been through testing [13,6] and model checking [21]. However, these techniques have been based on building models that are specific to the particular application and recovery strategy, and are difficult to reuse.

Program logics based on separation logic have been successful in reasoning about file systems [5,9] and concurrent indexes [16] on which database and file systems depend. However, as is typical with Hoare logics, their specifications avoid host failures, assuming that if a precondition holds then associated operations will not fail. Faulty Logic [7] by Meola and Walker is an exception. Faulty logic is designed to reason about transient faults, such as random bit flips due to background radiation, which are different in nature from host failure.

Zengin and Vafeiadis propose a purely functional programming language with an operational semantics providing tolerance against processor failures in parallel programs [22]. Computations are check-pointed to durable storage before execution and, upon detection of a failure, the failed computations are restarted. In general, this approach does not work for concurrent imperative programs which mutate the durable store.

In independent work, Chen et al. introduced Crash Hoare Logic (CHL) to reason about host failures and applied it to a substantial sequential journaling file system (FSCQ) written in Coq [2]. CHL extends Hoare triples with fault-conditions and provides highly automated reasoning about host failures. FSCQ performs physical journaling, meaning it uses a write-ahead log for both data and metadata, so that the recovery can guarantee atomicity with respect to host failures. The authors use CHL to prove that this property is indeed true. The resource stored in the disk is treated as durable. Since FSCQ is implemented in the functional language of Coq, which lacks the traditional process heap, the volatile state is stored in immutable variables.

The aim of FSCQ and CHL is to provide a verified implementation of a sequential file system which tolerates host failures. In contrast, our aim is to provide a general methodology for fault-tolerant resource reasoning about concurrent programs. We extend the Views framework [3] to provide a general concurrent framework for reasoning about host failure and recovery. Like CHL, we extend Hoare triples with fault-conditions. We instantiate our framework to concurrent separation logic, and demonstrate that an ARIES recovery algorithm uses the write-ahead log correctly to guarantee the atomicity of transactions.

In the technical report [10], we explore the differences in the specifications of fault-tolerance guarantees in physical and logical journaling file systems.

As we are defining a framework, our reasoning of the durable and volatile state (given by arbitrary view monoids) is general. In contrast, CHL reasoning is specific to the durable state on the disk and the volatile state in the immutable variable store. CHL is able to reason modularly about different layers of abstraction of a file-system implementation, using *logical address spaces* which give a systematic pattern of use for standard predicates. We do not explore modular reasoning about layers of abstractions in this paper, since it is orthogonal to reasoning about host failures, and examples have already been studied in instances of the Views framework and other separation logic literature [12,4,17,20,18].

We can certainly benefit from the practical CHL approach to mechanisation and proof automation. We also believe that future work on CHL, especially on extending the reasoning to heap-manipulating concurrent programs, can benefit from our general approach.

7 Conclusions and Future Work

We have developed fault-tolerant resource reasoning, extending the Views framework [3] to reason about programs in the presence of host failures. We have proved a general soundness result. For this paper, we have focused on fault-tolerant concurrent separation logic, a particular instance of the framework. We have demonstrated our reasoning by studying an ARIES recovery algorithm, showing that it is idempotent and that it guarantees atomicity of database transactions in the event of a host failure.

There has been recent work on concurrent program logics with the ability to reason about abstract atomicity [17]. This involves proving that even though the implementation of an operation takes multiple steps, from the client's point of view they can be seen as a single step. Currently, this is enforced by syntactic primitive atomic blocks ($\langle \_ \rangle$) in the programming language. In future, we want to combine abstract atomicity from concurrency with host failure atomicity.

Another direction for future work involves extending existing specifications for file systems [5,9] with our framework. This will allow both the verification of interesting clients programs, such as fault-tolerant software installers or persisted message queues, and the verification of fault-tolerant databases and file systems.

Acknowledgements. We thank Thomas Dinsdale-Young for discussions and useful feedback. This research was supported by EPSRC Programme Grants EP/H008373/1 and EP/K008528/1. Supplementary material and proofs are available in the technical report [10].

References

1. Bonwick, J., Ahrens, M., Henson, V., Maybee, M., Shellenbaum, M.: The zettabyte file system. In: Proc. of the 2nd Usenix Conference on File and Storage Technologies (2003)

2. Chen, H., Ziegler, D., Chlipala, A., Kaashoek, M.F., Kohler, E., Zeldovich, N.: Using Crash Hoare Logic for Certifying the FSCQ File System. SOSP (2015)
3. Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M., Yang, H.: Views: compositional reasoning for concurrent programs. In: POPL. pp. 287–300 (2013)
4. Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V.: Concurrent abstract predicates. In: ECOOP. pp. 504–528 (2010)
5. Gardner, P., Ntzik, G., Wright, A.: Local Reasoning for the POSIX File System. ESOP (2014)
6. Kropp, N., Koopman, P., Siewiorek, D.: Automated robustness testing of off-the-shelf software components. In: Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on. pp. 230–239 (1998)
7. Meola, M., Walker, D.: Faulty Logic: Reasoning about Fault Tolerant Programs. In: Gordon, A. (ed.) Programming Languages and Systems, Lecture Notes in Computer Science, vol. 6012, pp. 468–487. Springer Berlin Heidelberg (2010)
8. Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., Schwarz, P.: ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. ACM Trans. Database Syst. pp. 94–162
9. Ntzik, G., Gardner, P.: Reasoning about the POSIX File System: Local Update and Global Pathnames. OOPLSA (2015)
10. Ntzik, G., da Rocha Pinto, P., Gardner, P.: Fault-tolerant Resource Reasoning. Tech. rep., Imperial College London (2015)
11. O'Hearn, P.W.: Resources, concurrency, and local reasoning. Theor. Comput. Sci. 375(1-3), 271–307 (Apr 2007)
12. Parkinson, M., Bierman, G.: Separation logic and abstraction. In: POPL. pp. 247–258 (2005)
13. Prabhakaran, V., Arpaci-Dusseau, A., Arpaci-Dusseau, R.: Model-based failure analysis of journaling file systems. In: Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on. pp. 802–811 (June 2005)
14. Prabhakaran, V., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: Analysis and Evolution of Journaling File Systems. In: USENIX Annual Technical Conference, General Track (2005)
15. Reynolds, J.: Separation logic: a logic for shared mutable data structures. In: Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on. pp. 55–74 (2002)
16. da Rocha Pinto, P., Dinsdale-Young, T., Dodds, M., Gardner, P., Wheelhouse, M.: A simple abstraction for complex concurrent indexes. In: OOPSLA. pp. 845–864 (2011)
17. da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P.: TaDA: A Logic for Time and Data Abstraction. In: ECOOP. pp. 207–231 (2014)
18. da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P.: Steps in modular specifications for concurrent modules. In: MFPS (2015)
19. Rosenblum, M., Ousterhout, J.K.: The Design and Implementation of a Log-structured File System. ACM Trans. Comput. Syst. pp. 26–52 (1992)
20. Svendsen, K., Birkedal, L.: Impredicative Concurrent Abstract Predicates. In: ESOP. pp. 149–168 (2014)
21. Yang, J., Twohey, P., Engler, D., Musuvathi, M.: Using Model Checking to Find Serious File System Errors. ACM Trans. Comput. Syst. 24(4), 393–423 (Nov 2006)
22. Zengin, M., Vafeiadis, V.: A Programming Language Approach to Fault Tolerance for Fork-Join Parallelism. In: Theoretical Aspects of Software Engineering (TASE), 2013 International Symposium on. pp. 105–112 (July 2013)