Fault-Tolerant Resource Reasoning

Gian Ntzik, Pedro da Rocha Pinto and Philippa Gardner

Imperial College London

{gn408,pmd09,pg}@imperial.ac.uk

January 13, 2016

Example: A naive bank transfer

```
widthdraw(from, amount);
deposit(to, amount);
```

Host-failure

$$\texttt{Host-failure} \rightarrow \begin{array}{l} \texttt{widthdraw(from, amount);} \\ \texttt{deposit(to, amount);} \end{array}$$

- Host-failure in the middle of the execution.
- Impossible to avoid
- Breaks state invariants

  e.g. the sum of the accounts is the same before and after
- Mitigated with recovery procedures fixing things

Resource Reasoning

- Separation Logic style reasoning
  - Sequential & concurrent programs
  - Library reasoning: DOM, POSIX, indexes, stacks, queues, ...
- Used to verify that programs use resources correctly

Example: Naive bank transfer specification

$$\vdash \begin{array}{c} \{\mathsf{Account}(\texttt{from}, v) * \mathsf{Account}(\texttt{to}, w)\} \\ \texttt{widthdraw}(\texttt{from}, \texttt{amount}); \\ \texttt{deposit}(\texttt{to}, \texttt{amount}); \\ \{\mathsf{Account}(\texttt{from}, v - \texttt{amount}) * \mathsf{Account}(\texttt{to}, w + \texttt{amount})\} \end{array}$$

Fault avoiding interpretation of Separation Logic

$$\vdash \{P\}\ \mathbb{C}\ \{Q\}$$

Starting with resource satisfying P, running $\mathbb{C}$ **will not fault**, and if the $\mathbb{C}$ terminates, the resource will satisfy Q.

- *fault* means illegal use of resource
- Assumes no host-failures
- Incomplete behaviour specification

Reasoning about host-failure: volatile & durable resources

$$\vdash \{P_V | P_D\}\ \mathbb{C}\ \{Q_V | Q_D\}$$

- Distinguish volatile and durable resource:
  P_V, Q_V volatile (e.g. heap), P_D, Q_D durable (e.g. disk)

Reasoning about host-failure: fault-condition

$$S \vdash \{P_V | P_D\}\ \mathbb{C}\ \{Q_V | Q_D\}$$

- Distinguish volatile and durable resource:
  P_V, Q_V volatile (e.g. heap), P_D, Q_D durable (e.g. disk)
- Fault-condition S:
  durable resource after a host-failure and potential recovery

Resource-fault avoiding & host-failing interpretation

$$S \vdash \{P_V | P_D\}\ \mathbb{C}\ \{Q_V | Q_D\}$$

- Resource-fault avoiding:

  $\mathbb{C}$ will not fail due to illegal resource use

- Host-failing:

  If a host-failure occurs when running $\mathbb{C}$, the volatile resource is lost, and after potential recovery the durable resource will satisfy S.

Example: Naive bank transfer with host-failure

$$S \vdash \left\{ \frac{\texttt{from} = f \wedge \texttt{to} = t \wedge \texttt{amount} = a \wedge \mathsf{emp}}{\mathsf{Account}(f, v) * \mathsf{Account}(t, w)} \right\}$$
$$\texttt{widthdraw}(\texttt{from}, \texttt{amount});$$
$$\texttt{deposit}(\texttt{to}, \texttt{amount});$$
$$\left\{ \frac{\texttt{from} = f \wedge \texttt{to} = t \wedge \texttt{amount} = a \wedge \mathsf{emp}}{\mathsf{Account}(f, v - a) * \mathsf{Account}(t, w + a)} \right\}$$

where:

$$\begin{aligned} S = {} & (\mathsf{Account}(f, v) * \mathsf{Account}(t, w)) \\ \vee {} & (\mathsf{Account}(f, v - a) * \mathsf{Account}(t, w + a)) \\ \vee {} & (\mathsf{Account}(f, v - a) * \mathsf{Account}(t, w)) \end{aligned}$$

Fault-tolerant bank transfer specification

- Atomic with respect to host-failure:

  The transfer either completes, or does not happen at all.

Fault-tolerant bank transfer specification

- Atomic with respect to host-failure:

  The transfer either completes, or does not happen at all.

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

where:

$$\begin{aligned} R = {} & (\mathsf{Account}(f, v) * \mathsf{Account}(t, w)) \\ \vee {} & (\mathsf{Account}(f, v - a) * \mathsf{Account}(t, w + a)) \end{aligned}$$

Recovery Abstraction Rule

$$\mathbb{C}_R \textbf{ recovers } \mathbb{C}$$

$$\frac{S \vdash \{P_V \mid P_D\}\ \mathbb{C}\ \{Q_V \mid Q_D\} \qquad S \vdash \{\mathsf{emp} \mid S\}\ \mathbb{C}_R\ \{\mathsf{true} \mid R\}}{R \vdash \{P_V \mid P_D\}\ [\mathbb{C}]\ \{Q_V \mid Q_D\}}$$

Recovery Abstraction Rule

$$\mathbb{C}_R \textbf{ recovers } \mathbb{C}$$

$$\frac{S \vdash \{P_V \mid P_D\}\ \mathbb{C}\ \{Q_V \mid Q_D\} \qquad S \vdash \{\mathsf{emp} \mid S\}\ \mathbb{C}_R\ \{\mathsf{true} \mid R\}}{R \vdash \{P_V \mid P_D\}\ [\mathbb{C}]\ \{Q_V \mid Q_D\}}$$

- If $R = P_D \vee Q_D$ then $[\mathbb{C}]$ is atomic w.r.t host-failure.
- Weaker fault-tolerance guarantees can be established
  e.g. for journaling file systems

Bank transfer with Write-Ahead Logging

- Write-ahead Logging (WAL):
  - Before any update, write information to a (durable) log
  - During recovery, use log to detect and fix broken state

Bank transfer with WAL implementation

```
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)];
}
```

Bank transfer with WAL: Host-failure specification

$$S \vdash \left\{\frac{\texttt{from} = f \wedge \texttt{to} = t \wedge \texttt{amount} = a \wedge \mathsf{emp}}{\mathsf{Account}(f, v) * \mathsf{Account}(t, w)}\right\} \; \texttt{transfer}(\texttt{from}, \texttt{to}, \texttt{amount}) \; \left\{\frac{\texttt{from} = f \wedge \texttt{to} = t \wedge \texttt{amount} = a \wedge \mathsf{emp}}{\mathsf{Account}(f, v - a) * \mathsf{Account}(t, w + a)}\right\}$$

where:

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

Bank transfer with WAL: Recovery

```
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)];
  }
}
```

Bank transfer with WAL: Recovery Specification

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

$$
S \vdash \left\{ \frac{\left\{ \dfrac{\mathsf{emp}}{S} \right\}}{\texttt{transferRecovery}()} \; \frac{\mathsf{true}}{\begin{array}{c}(\mathsf{Account}(f, v) * \mathsf{Account}(t, w)) \\ \vee (\mathsf{Account}(f, v - a) * \mathsf{Account}(t, w + a))\end{array}} \right\}
$$

Fault-tolerant bank transfer specification

$$R \vdash \left\{\frac{\texttt{from} = f \wedge \texttt{to} = t \wedge \texttt{amount} = a \wedge \mathsf{emp}}{\mathsf{Account}(f, v) * \mathsf{Account}(t, w)}\right\} [\texttt{transfer}(\texttt{from}, \texttt{to}, \texttt{amount})] \left\{\frac{\texttt{from} = f \wedge \texttt{to} = t \wedge \texttt{amount} = a \wedge \mathsf{emp}}{\mathsf{Account}(f, v - a) * \mathsf{Account}(t, w + a)}\right\}$$

where:

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

Semantics

- Views framework:

  General soundness framework for concurrent program logics & type systems: e.g. SL, CSL, RGSep, CAP, RG
- Extended to: Fault-tolerant Views framework

  General soundness framework for fault-tolerant concurrent program logics
- Fault-tolerant Concurrent Separation Logic (FTCSL)

Case Study: ARIES Recovery

- ARIES: collection of algorithms for ACID in databases
- Studied an ARIES-style recovery algorithm
- Based on WAL:

  Based on log and database state after host-failure, commit transactions logged to be committed & roll-back uncommitted transactions
- Complex durable and volatile state

  Log checkpoints, transaction table, dirty page table, forward and backward scanning
- Verified that:
  - Recovery is idempotent (recovery abstraction rule)
  - Transactions are committed or rolled-back as intended (A,D in ACID)

Related Work: Separation Logics with faults

- Meola M., Walker D.: Faulty Logic: Reasoning about Fault Tolerant Programs, PLS 2010

  Separation Logic extension to reason about *transient faults*, e.g. random bit flips on the heap.
- Chen et al. Using Crash Hoare Logic for Certifying the FSCQ File System, SOSP 2015

  Coq implementation of a sequential fault-tolerant file system.

Conclusions & Future Work

- General framework for reasoning about host-failures
- Reasoning about fault-tolerance with WAL.
- ARIES recovery verification
- Future:
  - Link with atomicity in concurrency
  - Fault-tolerant file-system specifications
  - Examples: persisted message-queues, transactions (full ACID)
  - Automation