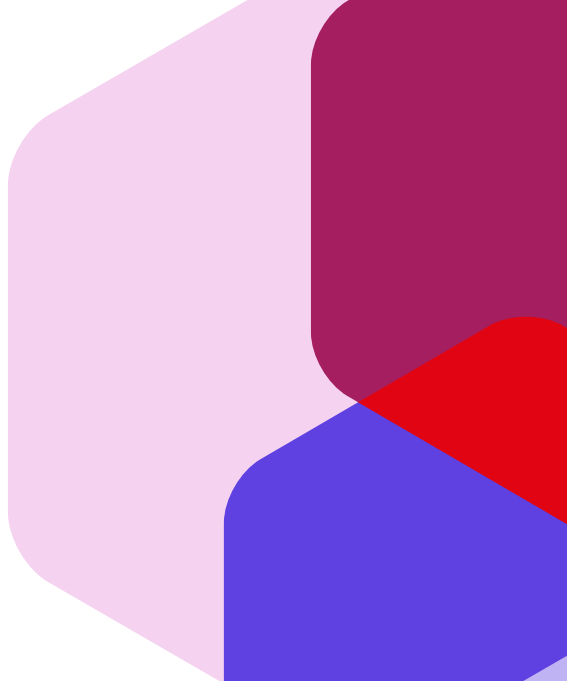


Principles, Algorithms and Models of Concurrency and Synchronization

06 Software Transactional Memory



Overview

- 1 Introduction
- 2 ACID Properties
- 3 Operations of TM systems
- 4 Features of TM systems
- 5 Correctness concepts for STM systems
- 6 The TL2 Algorithm
- 7 STM Haskell
- 8 Examples
- 9 Combining Transactions
- 10 Implementation

- Quite new programming approach (first mentioned around 1995, more intensive research around 2005)
- Goal: Programmers write more or less sequential code
- System guarantees correct concurrent execution (e.g., no unwanted deadlocks, no race conditions, etc.)

Motivation: Example

- Transfers between account A and B
- Solution: Lock accounts according to a total order

Motivation: Example

- Transfers between account A and B
- Solution: Lock accounts according to a total order
- Extension: Only debit if account has sufficient funds

Motivation: Example

- Transfers between account A and B
- Solution: Lock accounts according to a total order
- Extension: Only debit if account has sufficient funds
- Solution 1: Abort and restart: If account is not covered, start again. Carries the risk of having to restart again and again.

Motivation: Example

- Transfers between account A and B
- Solution: Lock accounts according to a total order
- Extension: Only debit if account has sufficient funds
- Solution 1: Abort and restart: If account is not covered, start again. Carries the risk of having to restart again and again.
- Solution 2: Wait until account is covered (locks must be released!)

Motivation: Example

- Transfers between account A and B
- Solution: Lock accounts according to a total order
- Extension: Only debit if account has sufficient funds
- Solution 1: Abort and restart: If account is not covered, start again. Carries the risk of having to restart again and again.
- Solution 2: Wait until account is covered (locks must be released!)
- Conclusion: Small extensions require major changes

Lock-Based Programming is Bad ...

(Arguments by S. L. Peyton Jones)

- **Setting too few locks:** Programmer forgets to set a lock

Consequence: Race condition

- **Setting too many locks:** Programmer is “overcautious”:

Consequences:

- Program unnecessarily sequential (best case)
 - Deadlock (worst case)
- **Setting the wrong locks:**
 - Only the programmer knows the relationship between locks and data.
 - Compilers or other programmers may not know the relationship.

Lock-Based Programming is Bad ... (2)

- **Setting locks in the wrong order:**
 - The total order theorem can only be satisfied if every programmer knows what the order of the locks looks like.
- **Error handling is difficult** because locks must be unlocked during error handling.
- **Forgotten** signal operations or forgotten rechecking of conditions lead to faulty systems.

A Compelling Argument

Lock-based programming is **not modular**

Example:

- Transfer from account A_1 or A_2 to account B , depending on which account has sufficient funds.

A Compelling Argument

Lock-based programming is **not modular**

Example:

- Transfer from account A_1 or A_2 to account B , depending on which account has sufficient funds.
- It cannot be composed from the existing programs
- but it requires a new program

- Database systems are concurrent
- From the user's point of view, however: No locking or similar measures are necessary
- Users write queries that are executed as **transactions** on the database
- Idea: "Transfer this model for parallel programming"
- **transactions** on the **shared memory**

- There are some (prototypical) implementations
- There is **software transactional memory** but also **hardware transactional memory**
- Lots of design differences
- Transaction management is time-consuming and space-intensive
- Haskell and Clojure use Software Transactional Memory
- Frameworks ZIO (Scala) and Effect (TypeScript) are based on STM.
- Hardware: Intel TSX (security issues, default: deactivated)

ACID Properties

First, let's look at database transactions.

These must fulfill the ACID properties:

- **A**tomicity: All operations of a transaction are executed, or **none** are executed.
Prohibited: Operation fails, but transaction is successful
Prohibited: Failed transaction leaves observable differences
A successful transaction **commits**, a failed transaction **aborts**



ACID Properties (2)

- Consistency: A transaction changes the state of the database. This change must be consistent.
The consistency of a committed transaction depends on the application. (For example, the balance of a bank account cannot be arbitrarily negative, or a newly added account must be assigned a unique account number, etc.).



ACID Properties (3)

- **Isolation**: A transaction delivers a correct result regardless of how many other concurrent transactions are being performed.



ACID Properties (4)

- Durability: The result of a committed transaction is permanent. This means that it is written permanently to the hard disk or similar before the transaction can be marked as committed.



- A,C,I desirable
- Durability not possible because main memory is volatile.
- Atomicity only guaranteed if all operations are performed as transactions.

Further requirements for TM:

- Databases can offset access times on hard disks with computing time, but this is not possible in main memory because access times are much shorter
- TM must be integrated into existing programming languages.

atomic blocks:

```
atomic {  
    Transaction code  
}
```

- Code is executed as a transaction
- Advantage over monitors: Variables do not have to be explicitly enumerated.
- Problem: What happens if the same variable is also changed from outside?



Atomicity Cannot be Guaranteed

Transactions must abort or commit, but:

```
atomic {  
    while True do skip;  
}
```

Therefore, different semantics (unlike databases):

The execution of a transaction (atomic block) has three possible outcomes:

- commit, if the transaction was successful
- abort, if the transaction was aborted
- undefined if the transaction did not terminate

Aborting Transactions

Transactions abort when

- a conflict occurs and the transaction manager decides to abort (and roll back).

Normal behavior: Manager attempts to execute the transaction again

- an explicit abort command is called.

Normal behavior: Transaction is aborted

```
atomic {  
  Balance := Balance + Interest;  
  if Balance < 0 then abort;  
}
```



- Program behaves as if a global lock is used for all `atomic` blocks.
- Accessing transaction variables outside of transactions can cause arbitrary nonsense
- Write access: variable is changed
- Read access: Can observe values that have not yet been committed!

Basic Operations (2)

The retry command

- Enables coordination of transactions
- `retry`: Transaction is aborted (rollback) and restarted



Example: Bounded Buffer:

```
atomic{  
    if isEmpty(buffer) then retry;  
    Read first element of buffer, etc.  
}
```

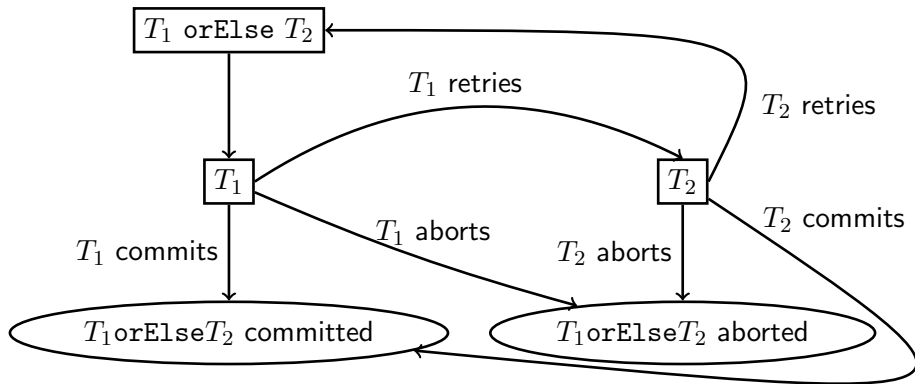
Basic Operations (3)

The orElse command

- Specifies alternatives when transactions abort
- T_1 orElse T_2 .
 - First, T_1 is executed. If T_1 is committed or T_1 is explicitly aborted via command, then T_2 is **not** executed. The transaction T_1 orElse T_2 commits or aborts, depending on the outcome of T_1 .
 - If T_1 is aborted by the transaction manager or retry is executed, then the transaction manager does not restart T_1 , but attempts to execute T_2 .
 - If T_2 is explicitly aborted or committed, then the entire transaction is terminated.
 - If T_2 performs a retry (or is aborted by the system), then T_1 orElse T_2 is restarted.



orElse Diagram



orElse Example

```
atomic {  
  {  
    B := B + amount;  
    A1 := A1 - amount;  
    if A1 ≤ 0 then retry;  
  }  
  orElse  
  {  
    B := B + amount;  
    A2 := A2 - amount;  
    if A2 ≤ 0 then retry;  
  }  
}
```

Features of TM Systems: Weak / Strong Isolation

Weak Isolation:

- Memory locations can also be manipulated outside of transactions.

Strong Isolation:

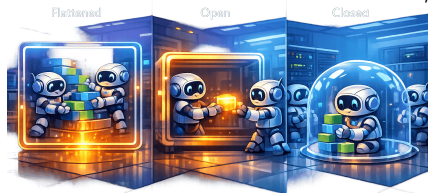
- Separation into transaction variables and other variables.
- System protects access to transaction variables: e.g., through
 - Automatic insertion of atomic blocks by the compiler
 - Type system prohibits access to transaction variables from outside (e.g., in Haskell)



Features of TM Systems: Nested Transactions

```
x := 1;  
atomic {  
  x := 2;  
  atomic {  
    x := 3;  
    abort;  
  }  
}
```

What does such nesting mean? → Three variants: flattened, open, closed transactions



Features of TM Systems: Nested Transactions (2)

Flattened transactions:

Behavior as if the inner atomic statement is missing.

<pre>x := 1; atomic { x := 2; atomic { x := 3; abort; } }</pre>	behaves like:	<pre>x := 1; atomic { x := 2; x := 3; abort; }</pre>
---	---------------	--

The example results in abort ($x=1$).

Features of TM Systems: Nested Transactions (3)

Closed transactions:

- Behaves as in flattened transactions, but:
- Internal abort does not lead to abort of the outer transaction.

<pre> x := 1; atomic { x := 2; atomic { x := 3; abort; } } </pre>	behaves like	<pre> x := 1; atomic { x := 2; } </pre>
(since inner transaction aborts)		

Example results in $x=2$.

Features of TM Systems: Nested Transactions (4)

Open transactions:

- Inner transactions are independent.
- Once an internal transaction is committed, its effect is visible to **all** and remains visible even if the outer transaction aborts.

```
x := 1
atomic {
  x:=2;
  atomic {
    x:= 3;
  }
  abort;
}
```

Result: $x = 3!$

- Which units are monitored for conflicts?
- Word/block granularity: If parallel access to memory words, then conflict
- Object granularity: Parallel access to an object leads to conflict

Note for object granularity: Conflict also occurs when **different** object attributes are changed

Direct update

- Executing transactions modify objects immediately.
- Requires protection against simultaneous access (concurrency control)
- Old values are stored in a log
- Transaction termination:
Rollback and recovery to restore data from the log

Delayed update

- Transactions receive local copies of the objects
- Modifications are first made to the copies
- Upon commit: Originals are replaced by local copies
- Logging: Who has which copies, who is allowed to commit?

Features of TM Systems: Direct/Delayed Update (3)

- Direct update is **optimistic**, delayed update **pessimistic**
- Optimistic: Assumption that conflicts rarely occur
- Pessimistic: Assumption that conflicts occur frequently

- **Early:** At the first access that leads to the conflict
(Required for this: Logging who accessed when)
- **Late:** Only when committing is it checked whether there is a conflict
- **Sometime:** Iterative checking (at intervals)

Features of TM Systems: Conflict Management

- In the event of a conflict: Which transaction is aborted?
- Many different strategies
- Goal: Progress of the overall system
- This prohibits: aborting all transactions

Correctness Criteria for STM Systems

- When is a TM system considered to be “correct”?
- Rather controversial, much research
- E.g., Dziuwa, Fatourou, Kanellou 2015: Overview with 16 different correctness criteria and relationships between them:
 - strict serializability
 - serializability
 - opacity
 - causal consistency
 - causal serializability
 - virtual world consistency
 - strong virtual world consistency
 - snapshot isolation

Each in an eager and a deferred update variant

Correctness Criteria for STM Systems (2)

Formal consideration of execution:

- Transaction execution:
 - Step-by-step processing, whereby each step
 - corresponds to a call on a concurrent object (e.g., atomic register, CAS object, ...) and
 - local computations
- We restrict ourselves to atomic registers as objects
- Executions are recorded as histories

Definition:

A **history** H is a **sequence of events**, where an event can be:

- Transaction T_i calls a command:
 - $T_i.\text{read}(x)$ (reading the transactional variable x)
 - $T_i.\text{write}(x, v)$ (writing the transactional variable x with the value v)
 - $T_i.\text{commit}$ (T_i wants to commit)
 - $T_i.\text{abort}$ (T_i wants to abort)
- Transaction T_i receives return value for a command:
 - $T_i.w$ for read (w is the value read)
 - $T_i.\text{Ok}$ for write, commit
 - $T_i.A_{T_i}$ indicates termination of transaction T_i
(possible for all operations, only possible return value for abort)

Example

E.g. $x = y = z = 1$ at the beginning

Transaction 1: $x := y + 1$

Transaction 2: $y := z + 1$

One possible history is:

$T_1.read(y)$

$T_2.read(z)$

$T_2.1$

$T_2.write(y, 2)$

$T_1.1$

$T_2.Ok$

$T_1.write(x, 2)$

$T_1.Ok$

$T_1.commit$

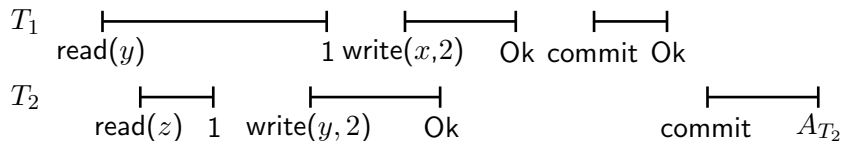
$T_1.Ok$

$T_2.commit$

$T_2.A_{T_2}$

Example (2)

Representation of history with intervals along the time axis



- First: Terms and necessary notation for histories
- Next: Selection of correctness terms using these formalisms
- We present only a few criteria
(from Guerraoui & Kapalka, PPOPP 2008)

Let H be a history.

- For transaction T , $H|T$ is the **restriction of H to T** :
Delete all events from H that do not belong to T .
- For $T_1, \dots, T_n$, $H|\{T_1, \dots, T_n\}$ is the **restriction of H to $T_1, \dots, T_n$** :
Delete all events from H that do not belong to $T_1, \dots, T_n$.
- For a transactional variable x , $H|x$ is the **restriction of H to x** :
Delete all events that do not belong to the variable x .

Examples

History H_0 :

$T_1.\text{read}(y)$

$T_2.\text{read}(z)$

$T_2.1$

$T_2.\text{write}(y,2)$

$T_1.1$

$T_2.\text{Ok}$

$T_1.\text{write}(x,2)$

$T_1.\text{Ok}$

$T_1.\text{commit}$

$T_1.\text{Ok}$

$T_2.\text{commit}$

$T_2.A_{T_2}$

Examples

History H_0 :

$T_1.\text{read}(y)$
 $T_2.\text{read}(z)$
 $T_2.1$
 $T_2.\text{write}(y,2)$
 $T_1.1$
 $T_2.\text{Ok}$
 $T_1.\text{write}(x,2)$
 $T_1.\text{Ok}$
 $T_1.\text{commit}$
 $T_1.\text{Ok}$
 $T_2.\text{commit}$
 $T_2.A_{T_2}$

History $H_0|T_2$:

$T_1.\text{read}(y)$
 $T_2.\text{read}(z)$
 $T_2.1$
 $T_2.\text{write}(y,2)$
 ~~$T_1.1$~~
 $T_2.\text{Ok}$
 ~~$T_1.\text{write}(x,2)$~~
 ~~$T_1.\text{Ok}$~~
 ~~$T_1.\text{commit}$~~
 ~~$T_1.\text{Ok}$~~
 $T_2.\text{commit}$
 $T_2.A_{T_2}$

Examples

History H_0 :

$T_1.\text{read}(y)$
 $T_2.\text{read}(z)$
 $T_2.1$
 $T_2.\text{write}(y,2)$
 $T_1.1$
 $T_2.\text{Ok}$
 $T_1.\text{write}(x,2)$
 $T_1.\text{Ok}$
 $T_1.\text{commit}$
 $T_1.\text{Ok}$
 $T_2.\text{commit}$
 $T_2.A_{T_2}$

History $H_0|T_2$:

$T_2.\text{read}(z)$
 $T_2.1$
 $T_2.\text{write}(y,2)$
 $T_2.\text{Ok}$
 $T_2.\text{commit}$
 $T_2.A_{T_2}$

Examples

History H_0 :

$T_1.read(y)$
 $T_2.read(z)$
 $T_2.1$
 $T_2.write(y,2)$
 $T_1.1$
 $T_2.Ok$
 $T_1.write(x,2)$
 $T_1.Ok$
 $T_1.commit$
 $T_1.Ok$
 $T_2.commit$
 $T_2.A_{T_2}$

History $H_0|T_2$:

$T_2.read(z)$
 $T_2.1$
 $T_2.write(y,2)$
 $T_2.Ok$
 $T_2.commit$
 $T_2.A_{T_2}$

History $H_0|\{T_1, T_2\}$:

$T_1.read(y)$
 $T_2.read(z)$
 $T_2.1$
 $T_2.write(y,2)$
 $T_1.1$
 $T_2.Ok$
 $T_1.write(x,2)$
 $T_1.Ok$
 $T_1.commit$
 $T_1.Ok$
 $T_2.commit$
 $T_2.A_{T_2}$

Examples

History H_0 :

$T_1.read(y)$
 $T_2.read(z)$
 $T_2.1$
 $T_2.write(y,2)$
 $T_1.1$
 $T_2.Ok$
 $T_1.write(x,2)$
 $T_1.Ok$
 $T_1.commit$
 $T_1.Ok$
 $T_2.commit$
 $T_2.A_{T_2}$

History $H_0|T_2$:

$T_2.read(z)$
 $T_2.1$
 $T_2.write(y,2)$
 $T_2.Ok$
 $T_2.commit$
 $T_2.A_{T_2}$

History $H_0|\{T_1, T_2\}$:

$T_1.read(y)$
 $T_2.read(z)$
 $T_2.1$
 $T_2.write(y,2)$
 $T_1.1$
 $T_2.Ok$
 $T_1.write(x,2)$
 $T_1.Ok$
 $T_1.commit$
 $T_1.Ok$
 $T_2.commit$
 $T_2.A_{T_2}$

History $H_0|y$:

$T_1.read(y)$
 ~~$T_2.read(z)$~~
 ~~$T_2.1$~~
 $T_2.write(y,2)$
 $T_1.1$
 $T_2.Ok$
 ~~$T_1.write(x,2)$~~
 ~~$T_1.Ok$~~
 $T_1.commit$
 ~~$T_1.Ok$~~
 $T_2.commit$
 ~~$T_2.A_{T_2}$~~

Examples

History H_0 :

$T_1.read(y)$
 $T_2.read(z)$
 $T_2.1$
 $T_2.write(y,2)$
 $T_1.1$
 $T_2.Ok$
 $T_1.write(x,2)$
 $T_1.Ok$
 $T_1.commit$
 $T_1.Ok$
 $T_2.commit$
 $T_2.A_{T_2}$

History $H_0|T_2$:

$T_2.read(z)$
 $T_2.1$
 $T_2.write(y,2)$
 $T_2.Ok$
 $T_2.commit$
 $T_2.A_{T_2}$

History $H_0|\{T_1, T_2\}$:

$T_1.read(y)$
 $T_2.read(z)$
 $T_2.1$
 $T_2.write(y,2)$
 $T_1.1$
 $T_2.Ok$
 $T_1.write(x,2)$
 $T_1.Ok$
 $T_1.commit$
 $T_1.Ok$
 $T_2.commit$
 $T_2.A_{T_2}$

History $H_0|y$:

$T_1.read(y)$
 $T_2.write(y,2)$
 $T_1.1$
 $T_2.Ok$

Well-formed Histories

History H is **well-formed** if the following applies to every transaction T :

- $H|T$ consists alternately of calls and returns
- $H|T$ does not contain any events after the event $T.A_T$
- If $H|T$ contains the event $T.commit$, then either nothing follows, or one instance of $T.A_T$ or one instance of $T.Ok$ follows
- If $H|T$ contains the event $T.abort$, then either nothing follows or one instance of $T.A_T$ follows

From now on, we will only consider well-formed histories!

Classifications of a transaction T in a history H :

- T is **committed** in H if $H|T$ ends with $T.\text{commit}$, $T.\text{Ok}$.
- T is **aborted** in H if $H|T$ ends with $T.A_T$.
- T is **complete** in H if T is committed or aborted in H .
- T is **running** in H if T is incomplete in H .

Restriction to committed transactions:

- $\text{comm}(H) := H|\{T_1, \dots, T_n\}$,
where $T_1, \dots, T_n$ are all transactions in H that are committed

Sequential History

A history is **sequential** if all events of a transaction are in a subsequence.

Examples:

History H_0 :

$T_1.read(y)$
 $T_2.read(z)$
 $T_2.1$
 $T_2.write(y,2)$
 $T_1.1$
 $T_2.Ok$
 $T_1.write(x,2)$
 $T_1.Ok$
 $T_1.commit$
 $T_1.Ok$
 $T_2.commit$
 $T_2.A_{T_2}$

not sequential

History H_1 :

$T_1.read(y)$
 $T_1.1$
 $T_1.write(x,2)$
 $T_1.Ok$
 $T_1.commit$
 $T_1.Ok$
 $T_2.read(z)$
 $T_2.1$
 $T_2.write(y,2)$
 $T_2.Ok$
 $T_2.commit$
 $T_2.A_{T_2}$

sequential

History H_2 :

$T_2.read(z)$
 $T_2.1$
 $T_2.write(y,2)$
 $T_2.Ok$
 $T_2.commit$
 $T_2.Ok$
 $T_1.read(y)$
 $T_1.2$
 $T_1.write(x,2)$
 $T_1.Ok$
 $T_1.commit$
 $T_1.Ok$

sequential

Definition

Two histories H, H' are **equivalent** (written as $H \sim H'$),
if for every transaction T : $H|T = H'|T$
(the histories per transaction are identical).

Example

<u>History H_1:</u>	<u>History H_2:</u>	<u>History H_3:</u>	<u>History H_4:</u>
$T_1.\text{read}(x)$	$T_1.\text{read}(x)$	$T_1.\text{read}(x)$	$T_2.\text{read}(x)$
$T_2.\text{read}(x)$	$T_2.\text{read}(x)$	$T_1.v$	$T_2.v$
$T_1.v$	$T_2.v$	$T_1.\text{write}(x, v')$	$T_2.\text{read}(y)$
$T_2.v$	$T_1.v$	$T_1.\text{Ok}$	$T_2.v''$
$T_2.\text{read}(y)$	$T_1.\text{write}(x, v')$	$T_1.\text{commit}$	$T_2.\text{commit}$
$T_1.\text{write}(x, v')$	$T_1.\text{Ok}$	$T_1.\text{Ok}$	$T_2.\text{Ok}$
$T_1.\text{Ok}$	$T_1.\text{commit}$	$T_2.\text{read}(x)$	$T_1.\text{read}(x)$
$T_1.\text{commit}$	$T_2.\text{read}(y)$	$T_2.v'$	$T_1.v$
$T_2.v''$	$T_1.\text{Ok}$	$T_2.\text{read}(y)$	$T_1.\text{write}(x, v')$
$T_1.\text{Ok}$	$T_2.v''$	$T_2.v''$	$T_1.\text{Ok}$
$T_2.\text{commit}$	$T_2.\text{commit}$	$T_2.\text{commit}$	$T_1.\text{commit}$
$T_2.\text{Ok}$	$T_2.\text{Ok}$	$T_2.\text{Ok}$	$T_1.\text{Ok}$

Example

History H_1 :
 $T_1.\text{read}(x)$

$T_1.v$

$T_1.\text{write}(x, v')$

$T_1.\text{Ok}$

$T_1.\text{commit}$

$T_1.\text{Ok}$

History H_2 :
 $T_1.\text{read}(x)$

$T_1.v$

$T_1.\text{write}(x, v')$

$T_1.\text{Ok}$

$T_1.\text{commit}$

$T_1.\text{Ok}$

History H_3 :
 $T_1.\text{read}(x)$

$T_1.v$

$T_1.\text{write}(x, v')$

$T_1.\text{Ok}$

$T_1.\text{commit}$

$T_1.\text{Ok}$

History H_4 :

$T_1.\text{read}(x)$

$T_1.v$

$T_1.\text{write}(x, v')$

$T_1.\text{Ok}$

$T_1.\text{commit}$

$T_1.\text{Ok}$

Example

History H_1 : $T_2.\text{read}(x)$ $T_2.v$ $T_2.\text{read}(y)$ $T_2.v''$ $T_2.\text{commit}$ $T_2.\text{Ok}$ History H_2 : $T_2.\text{read}(x)$ $T_2.v$ $T_2.\text{read}(y)$ $T_2.v''$ $T_2.\text{commit}$ $T_2.\text{Ok}$ History H_3 : $T_2.\text{read}(x)$ $T_2.v'$ $T_2.\text{read}(y)$ $T_2.v''$ $T_2.\text{commit}$ $T_2.\text{Ok}$ History H_4 : $T_2.\text{read}(x)$ $T_2.v$ $T_2.\text{read}(y)$ $T_2.v''$ $T_2.\text{commit}$ $T_2.\text{Ok}$

Example

History H_1 :	History H_2 :	History H_3 :	History H_4 :
$T_1.\text{read}(x)$	$T_1.\text{read}(x)$	$T_1.\text{read}(x)$	$T_2.\text{read}(x)$
$T_2.\text{read}(x)$	$T_2.\text{read}(x)$	$T_1.v$	$T_2.v$
$T_1.v$	$T_2.v$	$T_1.\text{write}(x, v')$	$T_2.\text{read}(y)$
$T_2.v$	$T_1.v$	$T_1.\text{Ok}$	$T_2.v''$
$T_2.\text{read}(y)$	$T_1.\text{write}(x, v')$	$T_1.\text{commit}$	$T_2.\text{commit}$
$T_1.\text{write}(x, v')$	$T_1.\text{Ok}$	$T_1.\text{Ok}$	$T_2.\text{Ok}$
$T_1.\text{Ok}$	$T_1.\text{commit}$	$T_2.\text{read}(x)$	$T_1.\text{read}(x)$
$T_1.\text{commit}$	$T_2.\text{read}(y)$	$T_2.v'$	$T_1.v$
$T_2.v''$	$T_1.\text{Ok}$	$T_2.\text{read}(y)$	$T_1.\text{write}(x, v')$
$T_1.\text{Ok}$	$T_2.v''$	$T_2.v''$	$T_1.\text{Ok}$
$T_2.\text{commit}$	$T_2.\text{commit}$	$T_2.\text{commit}$	$T_1.\text{commit}$
$T_2.\text{Ok}$	$T_2.\text{Ok}$	$T_2.\text{Ok}$	$T_1.\text{Ok}$

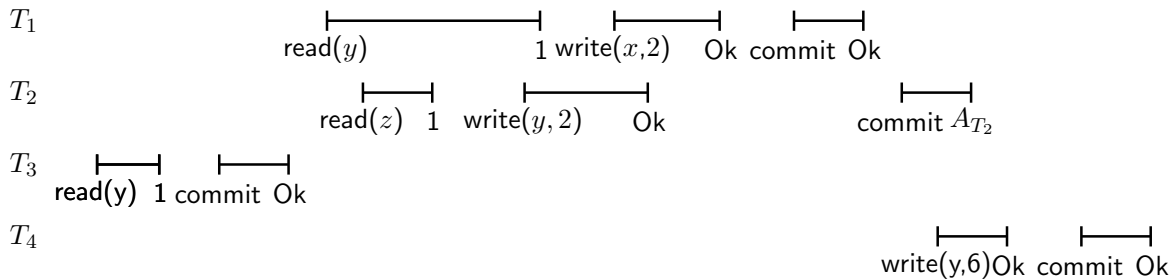
- H_1 and H_2 are equivalent
- H_1 and H_3 are not equivalent, H_1 and H_4 are equivalent

Definition

For a history H , the **real-time ordering** of transactions $\prec_H$ is the partial ordering defined by: If T_i is complete in H and the first event of T_j occurs after the last event of T_i in H , then $T_i \prec_H T_j$.

For sequential histories H whose transactions are all complete in H , the real-time order is always a total order.

Example of Real-Time Ordering



The following applies:

$$T_3 \prec_H T_1 \quad T_3 \prec_H T_2 \quad T_3 \prec_H T_4 \quad T_1 \prec_H T_4$$

Definition:

A **sequential** history H , where **all transactions are committed** except for **possibly the last** transaction, is **legal** if the following applies to all transactional variables:

$H|x$ matches the sequential description of atomic registers (the read and write operations and their returns behave as if they had been executed in a sequential program).

Examples

History H_5 :

$T_1.\text{read}(y)$

$T_1.3$

$T_1.\text{write}(x,4)$

$T_1.\text{Ok}$

$T_1.\text{read}(x)$

$T_1.4$

$T_1.\text{commit}$

$T_1.\text{Ok}$

$T_2.\text{read}(x)$

$T_2.2$

$T_2.\text{read}(y)$

$T_2.3$

History $H_5|x$:

$T_1.\text{write}(x,4)$

$T_1.\text{Ok}$

$T_1.\text{read}(x)$

$T_1.4$

$T_2.\text{read}(x)$

$T_2.2$

History $H_5|y$:

$T_1.\text{read}(y)$

$T_1.3$

$T_2.\text{read}(y)$

$T_2.3$

H_5 is not legal!

If aborted transactions are included in the sequential history, the previous definition of legal does not apply.

Therefore, the histories for each transaction are considered together with the previous successful ones.

Definition:

A sequential history H , whose transactions are all complete in H , is **legal** if for every transaction T_i in H , the history H_i is legal, where H_i is derived from H by transferring exactly those events $T_j.e$ from H to H_i with

- $j = i$, or
- T_j is committed in H and $T_j \prec T_i$ in H .

Note: H_i fulfills the requirements of the previously defined concept of legality: All transactions except possibly transaction T_i are committed in H_i .

Example

History H_6	History $H_{6,1}$	History $H_{6,2}$	History $H_{6,3}$
$T_1.write(x, 3)$	$T_1.write(x, 3)$	$T_3.read(x)$	$T_3.read(x)$
$T_1.Ok$	$T_1.Ok$	$T_3.4$	$T_3.4$
$T_1.read(x)$	$T_1.read(x)$	$T_3.write(y, 2)$	$T_3.write(y, 2)$
$T_1.3$	$T_1.3$	$T_3.Ok$	$T_3.Ok$
$T_1.abort$	$T_1.abort$	$T_3.commit$	$T_3.commit$
$T_1.A_{T_1}$	$T_1.A_{T_1}$	$T_3.Ok$	$T_3.Ok$
$T_3.read(x)$		$T_2.read(y)$	
$T_3.4$		$T_2.2$	
$T_3.write(y, 2)$		$T_2.read(x)$	
$T_3.Ok$		$T_2.4$	
$T_3.commit$			
$T_3.Ok$			
$T_2.read(y)$			
$T_2.2$			
$T_2.read(x)$			
$T_2.4$			

Since $H_{6,1}$, $H_{6,2}$ and $H_{6,3}$ are legal sequential histories of committed transactions, H_6 is a legal complete sequential history.

Minimal Requirements for Correct STM Implementations

- Transactions that are committed appear as if they were performed indivisibly in a single step.
- Aborted transactions appear as if they were never executed.

Further Requirements: Preservation of Real-Time Ordering

- The time at which the effects of a transaction appear is somewhere within the runtime of the transaction.
- This means that the transaction itself should not read outdated memory states.
- This effect can occur, for example, when old values of transactional variables are still stored due to the use of caches.
- The following should apply: If a transaction T_1 modifies and commits an object x , and then a transaction T_2 starts and reads x , then T_2 reads the value written by T_1 and not an older value.

Formally, the following is defined for histories:

A history H' **preserves the real-time order** of a history H if $\prec_H \subseteq \prec_{H'}$.

Further Requirements: Never Read Inconsistent States

- Are running transactions that are neither committed nor aborted allowed to read inconsistent memory states if it is ensured that they will be aborted later anyway?
- For database transactions, this is permitted and unproblematic
- However: STM transactions do not run in a completely isolated environment

Example:

- Let x and y be two (transactional) variables that must always satisfy the invariants $y = x^2$ and $x \geq 2$.
- The programmer satisfies the invariant by ensuring that all transactions satisfy it.
- Let $x = 4$ and $y = 16$
- Now consider the transaction $T_1: x := 2; y := 4$.
- If another transaction T_2 can observe the old value of x (4) and the new value of y (4), it could calculate $1/(y - x)$ and thus cause a runtime error.
- Aborting T_2 due to reading inconsistent values does not prevent this error.

Linearizability:

- Each transaction T acts like an atomic function call on the transactional variables, which is executed in one step.
- The successful transactions are then a sequence of such steps.

Sequentializability

Sequentializability requires that for every history H of the TM system, the following holds: There is a sequential and legal history S with $S \sim comm(H)$.

This means that the history of successful transactions is equivalent to the history of a sequential execution of the transactions.

Strict sequentializability

Strict sequentializability requires that for every history H of the TM system, the following applies: There is a sequential and legal history S with $S \sim comm(H)$ and $\prec_{comm(H)} \subseteq \prec_S$.

This means that in addition to sequentializability: Real-time order is maintained, i.e., the sequential execution found must correspond to the real-time order.

All terms so far only consider **successful transactions**.

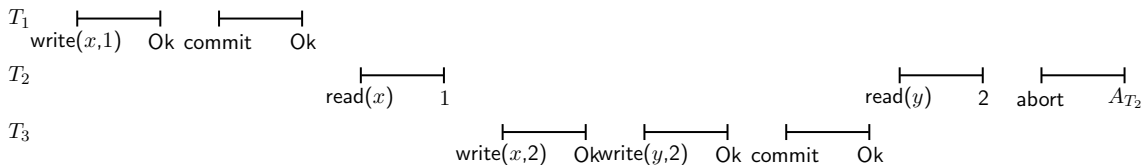
Recoverability

- If a transaction T_i writes to a transactional variable, then no other transaction T_j may read the same transactional variable before T_i has been successfully completed or aborted.

Problems:

- Sometimes enforces too much sequentialization of transaction execution.
- Does not prevent aborting transactions from seeing inconsistent states.

Example



Although recoverability applies, T_2 reads inconsistent values.

Correctness Criteria (5)

Opacity: Opacity requires that every history of the TM system is opaque:

History H is **opaque** if there exists a complete, sequential, legal history S that is equivalent to a history $H' \in complete(H)$ such that S respects the real-time order of H .

Here, $complete(H)$ is the set of histories H' such that all transactions occurring in H are complete in H' , where

- H' is well-formed,
- H' contains only additional commit and abort events compared to H , and only for transactions that are running in H .
- running transactions T from H for which $T.commit$ occurs in H are committed or aborted in H'
- running transactions T from H for which no $T.commit$ occurs in H are aborted in H' .

- The operations of each successful transaction are performed atomically in one step. This means in particular that if all steps of aborted or running transactions are removed, the (concurrent) execution of the successful transactions is equivalent to a sequential execution of these transactions.
In addition, sequential execution must adhere to the real-time order.
- The effects of aborted transactions are never visible to other transactions.
- Every transaction (whether successful or aborted) sees only consistent states, i.e., those that have been created by committed transactions and correspond to a state in sequential execution.

The TL2 Algorithm

- Proposed by Dice, Shalev, and Shavit, 2006
- TL = Transactional Locking
- Implements the transaction manager
- No `retry` and `orElse`
- No nested transactions
- Uses a global counter `gc` (fetch-and-increment object)
- Satisfies opacity.

The transactional variables $x_1, x_2, \dots$ are administered by:

- Array TVar, such that TVar[i] contains the content of x_i
- Array C, such that C[i] is a compare-and-swap object with content (ver,lock) where:
 - ver is the timestamp (of the global counter) of the last write access
 - lock is a Boolean value that indicates whether the memory location is considered locked

TL2: Local Data Structures of the Transaction

- Set readset: Addresses of the read memory locations
- Set writeset: Addresses of the written memory locations
- Array writelog[i]: Contains new value for x_i
Write operations are initially only performed locally!
- Timestamp readver: Timestamp of the first read access of the transaction
- Set lockset: Addresses of the locked memory locations

Shared data structures:

gc	(global counter), fetch-and-increment object, initial gc=0
TVar[1..]	Array of atomic registers, initial TVar[i] = 0
C[1..]	Array of compare-and-swap objects with pairs as content, initial C[i]=(0,false)

Local data structures:

readver	local counter value, initial readver = $\perp$
readset, writeset, lockset	sets of addresses, initial all = $\emptyset$
writelog[1..]	field of values

Operation $T_k.\text{read}(x_m)$

```
if readver =  $\perp$  then readver := read(gc); // set readver on first read
if  $m \in \text{writeset}$  then return writelog[m]; // if written locally, take this version
(ver1, locked1) := read(C[m]); // read CAS object
result := read(TVar[m]); // read content
(ver2, locked2) := read(C[m]); // read CAS object again
// Abort if read is not atomic, TVar is locked, or version is newer
if ver1  $\neq$  ver2 or locked2 or ver2 > readver then  $T_k.\text{abort}()$ ;
readset := readset  $\cup \{m\}$ ; // Add address to readset
return result;
```

TL2: Pseudo-Code (3)

Operation $T_k.\text{write}(x_m, val)$

```
writeset := writeset  $\cup$   $\{m\}$ ; // Add address to writeset  
writelog[m] := val; // write locally  
return Ok;
```

Operation $T_k.\text{abort}()$

```
// unlock all locked objects  
for all  $m \in \text{lockset}$  do  
    (ver,locked) := read(C[m]);  
    write(C[m],(ver,False));  
// restore initial state  
readset,writeset,lockset :=  $\emptyset, \emptyset, \emptyset$ ;  
readver :=  $\perp$ ;  
return  $A_{T_k}$ ;
```

TL2: Pseudo-Code (4)

Operation $T_k.\text{commit}()$

```
for all  $m \in \text{writeset}$  do // lock all TVars to be written and record them in the lockset
    (ver,locked) := read( $C[m]$ );
    if locked then  $T_k.\text{abort}()$ ;
    lock := compare-and-swap( $C[m]$ , (ver,False), (ver,True));
    if not lock then  $T_k.\text{abort}()$ ; // Abort if locking is not successful
    lockset := lockset  $\cup \{m\}$ ;
writever := 1+fetch-and-increment(gc); // Increment counter for new write version
if writever  $\neq$  readver+1 then // validate readset (not necessary for readver+1=writever)
    for all  $m \in \text{readset}$  do
        (ver,locked) := read( $C[m]$ );
        if ver > readver or (locked and  $m \notin \text{writeset}$ ) then  $T_k.\text{abort}()$ ;
for all  $m \in \text{writeset}$  do // Write new content and unlock
    write(TVar[m], writelog[m]);
    write( $C[m]$ , (writever,False))
readset, writeset, lockset :=  $\emptyset, \emptyset, \emptyset$ ;
readver :=  $\perp$ ;
return Ok;
```

- Checking the readset and locking the writeset ensures that the effect of the transaction acts like a function.
- TL2 uses few locks: Lock is only set when committing.
- Creating new memory locations: Simply by adding another set.

Read-only transactions

- Can also dispense with the readset
- They verify the validity of each read access only by means of the reader timestamp and the comparison of reader with ver_2 .
- Optimization: Treat each transaction as read-only at first; restart as a read/write transaction when writing for the first time.

TL2: Semantic Error in Conditional Non-Termination

Let $x = \text{True}$ at the beginning

Transaction 1

```
if x then  
  loop forever  
else return
```

Transaction 2

```
x := False;
```

If transaction 1 runs first, it will run in an endless loop, even though no endless loop is necessary after transaction 2 has been completed.

Possible remedy: Iterative examination of the readset for conflicts.

- Easier programming with TM
- Problem: Transactions can only safely process operations that are reversible.
- E.g., not reversible: Print "Hello," "Launch Missiles," etc.
- Further problems: Break code down into transactions:
 - Small transactions are better than large ones (due to potential for conflict)
 - You have to find out:
 - Which section of code must be executed atomically?
 - Data structures: E.g., a tree in a transactional variable vs. a single node in a transactional variable

Historical background:

- Haskell: Functional programming language, haskell.org
- Concurrent Haskell: Extension by threads
- STM in Haskell introduced by Harris, Marlow, Peyton Jones and Herlihy in 2005
- Implemented in the library `Control.Concurrent.STM`
- First paper that introduces `retry` and `orElse`
- Frameworks in Scala and TypeScript used the Haskell STM approach as a model

- Separation of operations on transaction variables and normal IO
- using the type system
- Like the IO monad, there is an STM monad
- Transactions have type STM a

- Separation of operations on transaction variables and normal IO
- using the type system
- Like the IO monad, there is an STM monad
- Transactions have type $\text{STM } a$
- $\text{atomically} :: \text{STM } a \rightarrow \text{IO } a$ converts an STM action into an IO operation

Semantics: $\text{atomically } t$ executes the transaction t atomically.

- Note: There is no inverse of type $\text{IO } a \rightarrow \text{STM } a$!

- `data TVar a = ...`
- `newTVar :: a -> STM (TVar a)`
Creates a new TVar with content
- `readTVar :: TVar a -> STM a`
Reads the current value of a TVar
- `writeTVar :: TVar a -> a -> STM ()`
Writes a new value into the TVar
- `newTVarIO :: TVar a -> a -> IO (TVar a)`
Creating a TVar in the IO monad

```
atomically (  
  do  
    ... accesses to TVar's ...  
  and  
    ... pure functional computations  
  
  ... but no IO!  
  
)
```

- `retry :: STM a`
- Aborts the transaction and restarts it
- **Implementation:**
 - Restart only when the values of TVars have changed
- This implicitly blocks!

Application: Using retry to wait for changes

Example:

- Processes change a counter
- There is a process that displays the counter on the screen
- Updates of the counter not constantly,
but only on changes of at least 1000

Application: Using retry to wait for changes (2)

Main program:

```
main = do
  -- TVar for the counter
  tv <- (newTVarIO 0)::IO (TVar Integer)
  -- create 10 worker threads
  mapM_ forkIO (replicate 10 (worker tv))
  -- start updating the display
  updateDisplay tv
```

Application: Using retry to wait for changes (3)

Randomly changing the counter:

```
worker tv =  
  do z <- randomRIO (-100,100)  
    threadDelay 10000  
    atomically $ do  
      old <- readTVar tv  
      writeTVar tv (z+old)  
worker tv
```

Application: Using retry to wait for changes (4)

Updating the display:

```
updateDisplay tv =  
  do b <- readTVarIO tv  
    printNext b -- display once  
    loop b      -- enter the loop  
where  
  loop b = do next <- atomically $  
    do c <- readTVar tv  
      if abs (b-c) < 1000  
      then retry  
      else return c  
    printNext next  
    loop next  
printNext i = putStr $ "\r" ++ show i ++ " "
```

Application: Using retry to wait for changes (4)

A more realistic example:

- Window manager
- Redraw windows when the position data changes
- Rendering waits with `retry` for position changes.

Examples

- We consider further programming examples
- Semaphores, philosophers, channels

Example: Binary semaphore

```
type Semaphore = TVar Bool

newSem :: Bool -> IO Semaphore
newSem k = newTVarIO k -- k True/False

wait :: Semaphore -> STM ()
wait sem = do b <- readTVar sem
             if b
               then writeTVar sem False
               else retry

signal :: Semaphore -> STM ()
signal sem = writeTVar sem True
```

Example: Unbounded Buffer

```
type Buffer a = TVar [a]

newBuffer :: IO (Buffer a)
newBuffer = newTVarIO []

put :: Buffer a -> a -> STM ()
put buffer item = do ls <- readTVar buffer
                    writeTVar buffer (ls ++ [item])

get :: Buffer a -> STM a
get buffer = do ls <- readTVar buffer
               case ls of
                 [] -> retry
                 (item:rest) -> do writeTVar buffer rest
                                   return item
```

Combining transactions

- $\gg=$, $\gg$, do build composed **sequences** from transactions

Combining transactions: Choice

- `orElse :: STM a -> STM a -> STM a`
- combines two transactions into one
- if the first transaction succeeds, then so does the combined one
- if the first transaction performs a `retry`, then the second transaction is performed.
- if both transactions perform a `retry`, then the entire transaction is restarted.

Dining Philosophers in STM

see: <http://computationalthoughts.blogspot.com/2008/03/some-examples-of-software-transactional.html>

```
simulation n = do
  forks <- sequence (replicate n (newSem True))
  outputBuffer <- newBuffer
  sequence_ [forkIO (philosoph i
                      outputBuffer
                      (forks!!i)
                      (forks!!((i+1)'mod'n)))
            | i <- [0..n-1]]
  output outputBuffer

output buffer = do
  str <- atomically $ get buffer
  putStrLn str
  output buffer
```

Dining Philosophers in STM (2)

```
philosoph :: Int->Buffer String->Semaphore->Semaphore->IO ()
philosoph n out fork1 fork2 = do
  atomically $ put out ("Philosopher " ++ show n ++ " is thinking.")
  atomically $ do wait fork1
                  wait fork2
  atomically $ put out ("Philosopher " ++ show n ++ " is eating.")
  atomically $ do signal fork1
                  signal fork2
philosoph n out fork1 fork2
```

- compose with TVars

```
type TChan a      = (TVar (TVarList a), TVar (TVarList a))
type TVarList a   = TVar (TList a)
data TList a      = TNil | TCons a (TVarList a)
```

Channels with STM Haskell: Creating

```
newTChan :: STM (TChan a)
newTChan = do
    hole <- newTVar TNil
    read  <- newTVar hole
    write <- newTVar hole
    return $ (read, write)
```

```
write :: TChan a -> a -> STM ()
write (read,write) val = do
  newEnd <- newTVar TNil
  oldEnd <- readTVar write
  writeTVar write newEnd
  writeTVar oldEnd (TCons val newEnd)
```

```
read :: TChan a -> STM a
read (read,write) =
  do
    listHead <- readTVar read
    tryHead <- readTVar listHead
    case tryHead of
      TNil -> retry
      TCons val rest -> do writeTVar read rest
                          return val
```

Channels with STM Haskell: Duplicating

```
dupChan :: TChan a -> STM (TChan a)
dupChan (read,write) =
  do
    hole    <- readTVar write
    new_read <- newTVar hole
    return (new_read,write)
```

Channels with STM Haskell: Test for emptiness

```
isEmpty :: TChan a -> STM Bool
isEmpty (read,write) =
  do
    listHead <- readTVar read
    tryHead <- readTVar listHead
    case tryHead of
      TNil      -> return True
      TCons _ _ -> return False
```

Recap: STM Haskell-API

Transactional variables

```
data TVar a = ...  
newTVar    ::  a -> STM (TVar a)  
readTVar   ::  TVar a -> STM a  
writeTVar  ::  TVar a -> a -> STM ()
```

Composing transactions

```
return      ::  a -> STM a  
(>>=)      ::  STM a -> (a -> STM b) -> STM b  
orElse      ::  STM a -> STM a -> STM a  
retry       ::  STM a
```

Executing transactions

```
atomically :: STM a -> IO a
```

Implementation of STM Haskell in GHC



- We describe the internal implementation in the Glasgow Haskell Compiler
- In particular: How are conflicts detected?
- When may a transaction commit?

- One transaction log is maintained per thread
- Transaction log: a table that contains:
 - Which TVars were written, read, created?
 - Read value of the TVar
 - New (to-be-written) value of the TVar
- A transaction log is **valid** if the read values match the current values of the TVars.
- Equality here: pointer equality.

Executing a transaction

- Create an empty transaction log
- Record read, write and create operations in the log
- At the end of the transaction: check the validity of the log
- If the log is valid, write the new values into the TVars
- If the log is invalid, restart (discard the log)

Note:

- Checking validity and writing happen atomically with respect to the other transactions.
- `retry` and `orElse` not yet considered

Example of Conflict Detection

saldo :: TVar Int

7

Example of Conflict Detection

saldo :: TVar Int

7

Thread 1

```
atomically (  
  do  
    local <- readTVar saldo  
    writeTVar saldo (local + 1)  
  )
```

Thread 2

```
atomically (  
  do  
    local <- readTVar saldo  
    writeTVar saldo (local - 3)  
  )
```

Example of Conflict Detection

saldo :: TVar Int

7

Thread 1

```
atomically (  
  do  
    local <- readTVar saldo  
    writeTVar saldo (local + 1)  
)
```

Thread 2

```
atomically (  
  do  
    local <- readTVar saldo  
    writeTVar saldo (local - 3)  
)
```

TVar	read	written
saldo		

Transaction log

Example of Conflict Detection

saldo :: TVar Int

7

Thread 1

```
atomically (  
  do  
    local <- readTVar saldo  
    writeTVar saldo (local + 1)  
  )
```

TVar	read	written
saldo		

Transaction log

Thread 2

```
atomically (  
  do  
    local <- readTVar saldo  
    writeTVar saldo (local - 3)  
  )
```

TVar	read	written
saldo		

Transaction log

Example of Conflict Detection

saldo :: TVar Int

7

Thread 1

```
atomically (  
  do  
    local <- readTVar saldo  
    writeTVar saldo (local + 1)  
  )
```

TVar	read	written
saldo		

Transaction log

Thread 2

```
atomically (  
  do  
    local <- readTVar saldo  
    writeTVar saldo (local - 3)  
  )
```

TVar	read	written
saldo	7	

Transaction log

Example of Conflict Detection

saldo :: TVar Int

7

Thread 1

```
atomically (  
  do  
    local <- readTVar saldo  
    writeTVar saldo (local + 1)  
)
```

TVar	read	written
saldo	7	

Transaction log

Thread 2

```
atomically (  
  do  
    local <- readTVar saldo  
    writeTVar saldo (local - 3)  
)
```

TVar	read	written
saldo	7	

Transaction log

Example of Conflict Detection

saldo :: TVar Int

7

Thread 1

```
atomically (  
  do  
    local <- readTVar saldo  
    writeTVar saldo (local + 1)  
)
```

TVar	read	written
saldo	7	8

Transaction log

Thread 2

```
atomically (  
  do  
    local <- readTVar saldo  
    writeTVar saldo (local - 3)  
)
```

TVar	read	written
saldo	7	

Transaction log

Example of Conflict Detection

saldo :: TVar Int

7

Thread 1

```
atomically (  
  do  
    local <- readTVar saldo  
    writeTVar saldo (local + 1)  
)
```

TVar	read	written
saldo	7	8

Transaction log

Thread 2

```
atomically (  
  do  
    local <- readTVar saldo  
    writeTVar saldo (local - 3)  
)
```

TVar	read	written
saldo	7	4

Transaction log

Example of Conflict Detection

saldo :: TVar Int

7

Thread 1

```
atomically (  
  do  
    local <- readTVar saldo  
    writeTVar saldo (local + 1)  
)
```

TVar	read	written
saldo	7	8

Transaction log

Thread 2

```
atomically (  
  do  
    local <- readTVar saldo  
    writeTVar saldo (local - 3)  
)
```

TVar	read	written
saldo	7	4

Transaction log

Example of Conflict Detection

saldo :: TVar Int

8

Thread 1

```
atomically (  
  do  
    local <- readTVar saldo  
    writeTVar saldo (local + 1)  
)
```

TVar	read	written
saldo	7	8

Transaction log

Thread 2

```
atomically (  
  do  
    local <- readTVar saldo  
    writeTVar saldo (local - 3)  
)
```

TVar	read	written
saldo	7	4

Transaction log

Example of Conflict Detection

saldo :: TVar Int

8

Thread 1

```
atomically (  
  do  
    local <- readTVar saldo  
    writeTVar saldo (local + 1)  
)
```

Thread 2

```
atomically (  
  do  
    local <- readTVar saldo  
    writeTVar saldo (local - 3)  
)
```

TVar	read	written
saldo	7	4

Transaction log

Example of Conflict Detection

saldo :: TVar Int

8

Thread 1

```
atomically (  
  do  
    local <- readTVar saldo  
    writeTVar saldo (local + 1)  
  )
```

Thread 2

```
atomically (  
  do  
    local <- readTVar saldo  
    writeTVar saldo (local - 3)  
  )
```

TVar	read	written
saldo	7	4

Transaction log

Abort & Restart

Example of Conflict Detection

saldo :: TVar Int

8

Thread 1

```
atomically (  
  do  
    local <- readTVar saldo  
    writeTVar saldo (local + 1)  
  )
```

Thread 2

```
atomically (  
  do  
    local <- readTVar saldo  
    writeTVar saldo (local - 3)  
  )
```

TVar	read	written
saldo		

Transaction log

Example of Conflict Detection

saldo :: TVar Int

8

Thread 1

```
atomically (  
  do  
    local <- readTVar saldo  
    writeTVar saldo (local + 1)  
  )
```

Thread 2

```
atomically (  
  do  
    local <- readTVar saldo  
    writeTVar saldo (local - 3)  
  )
```

TVar	read	written
saldo	8	

Transaction log

Example of Conflict Detection

saldo :: TVar Int

8

Thread 1

```
atomically (  
  do  
    local <- readTVar saldo  
    writeTVar saldo (local + 1)  
  )
```

Thread 2

```
atomically (  
  do  
    local <- readTVar saldo  
    writeTVar saldo (local - 3)  
  )
```

TVar	read	written
saldo	8	5

Transaction log

Example of Conflict Detection

saldo :: TVar Int

8

Thread 1

```
atomically (  
  do  
    local <- readTVar saldo  
    writeTVar saldo (local + 1)  
)
```

Thread 2

```
atomically (  
  do  
    local <- readTVar saldo  
    writeTVar saldo (local - 3)  
)
```

TVar	read	written
saldo	8	5

Transaction log

Example of Conflict Detection

saldo :: TVar Int

8

Thread 1

```
atomically (  
  do  
    local <- readTVar saldo  
    writeTVar saldo (local + 1)  
)
```

Thread 2

```
atomically (  
  do  
    local <- readTVar saldo  
    writeTVar saldo (local - 3)  
)
```

TVar	read	written
saldo	8	5

Transaction log

Example of Conflict Detection

saldo :: TVar Int

5

Thread 1

```
atomically (  
  do  
    local <- readTVar saldo  
    writeTVar saldo (local + 1)  
  )
```

Thread 2

```
atomically (  
  do  
    local <- readTVar saldo  
    writeTVar saldo (local - 3)  
  )
```

- Every TVar has an associated wait queue of threads
- When `retry` is executed, the corresponding thread enrolls itself in the wait lists of all read TVars and blocks.
- When another thread commits, it unblocks all threads in the wait queues of the TVars it writes to. These check whether the values have now changed compared to the read ones, and then restart, or wait again if the values are still the same.

- The transaction log is a stack of the previously described tables for the new values to be written; the read values are all kept (without a stack)
- orElse creates a new level on the stack
- Idea: If, in orElse T_1 T_2 , the transaction T_1 leads to retry, then the topmost level of the stack is deleted before T_2 is started.
- For checking the validity of the transaction log, all read TVars are checked.

Regular checking of the log

- GHC checks the transaction log regularly (not only at the end)
- As a result: earlier conflict detection
- Necessary for semantically correct handling of non-terminating transactions!

Example

```
i1 tv = atomically $
  do
    c <- readTVar tv
    if c then
      let loop i = do loop (i+1) in loop 0
    else return ()

i2 tv = atomically (writeTVar tv False)

main = do tv <- atomically (newTVar True)
         s <- newEmptyMVar
         forkIO (i1 tv >> putMVar s ())
         forkIO (i2 tv >> putMVar s ())
         takeMVar s >> takeMVar s
```

- **Isolation:** Strong (no access to transaction variables outside of transactions)
- **Granularity:** Word-based
- **Update:** Delayed (new values are written locally, only into memory upon commit)
- **Concurrency Control:** optimistic (means no locks on the memory locations)
- **Synchronization:** Blocking
- **Conflict Detection:** Late (only at commit)
- **Nested Transactions:** No (prevented by the type system)