STSM Scientific Report

Towards a model of Transactional Memory with Irrevocable Operations

Poznań, 30.07.2013

COST Action	IC1001—Transactional Memories: Foundations, Algorithms, Tools, and Applications (Euro-TM)
Reference code	COST-STSM-ECOST-STSM-IC1001-190513-030430
STSM Period	19.05.2013–30.06.2013
Participant	**Jan KOŃCZAK** Poznań University of Technology Institute of Computing Science Poland
Host	Rachid GUERRAOUI École Polytechnique Fédérale de Lausanne, Distributed Programming Laboratory Switzerland

Contents

1 Purpose of the STSM

Contrary to ACID transactions, transactions that appear in transactional memory and geo-distributed (or geo-replicated) systems cannot be precisely and comprehensively described using only the concepts, models and properties from database and shared memory systems. In recent years, Guerraoui and Kapałka defined a model of transactional memory which can be used to reason about properties and correctness of TM implementations. For this, they introduced a notion of opacity which replaces classical serializability of database transactions, and also discussed the progress (or liveness) properties of lock-based and obstruction-free TMs.

However, the model does not address some aspects of transactional systems, such as irrevocable operations and nested transactions. These aspects are notoriously difficult to design and reason about. Many existing TM implementations simply either forbid them which decreases the number of potential applications (e.g. Haskell TM), or they do not give any guarantees which jeopardizes correctness. The use of irrevocable operations (such as I/O or system calls) inside transactions is from pragmatic reasons unavoidable. On the other hand, nested transactions appear naturally whenever a transaction calls a library function that contains some synchronized code.

Thus, there is an urgent need to develop a model in which TM properties will be rethought taking into account irrevocable operations and also nested transactions. The irrevocable operations cause side effects that cannot be rolled back but may be compensated. This means that consistency guarantees should take into account these various additional scenarios. This raises several open research questions: How to model the side effects? What properties can a TM ensure if we assume side effects in addition to memory effects (or memory state)? When is a TM correct then? How to prove the correctness of a TM implementation with side effects and partial rollbacks (due to nesting)? What are the impossibility results (if any)? In the proposed project, we plan to focus on a model which will provide an abstraction to reason about these open problems. During the STSM we would like to begin this work focusing on irrevocable operations, and continue after STSM, possibly including in the model other aspects such as transaction nesting. The project will be done under supervision of Prof. Rachid Guerraoui at EPFL and Dr Paweł Wojciechowski at Poznań University of Technology.

2 Description of the work carried out during the STSM

2.1 Defining model of a TM with irrevocable operations

In order to reason about the irrevocability, first I had to precisely define it. Taking under account the existing work regarding the irrevocability in transactional memory, the proper way to introduce irrevocability is to add an *irrevocable* operation support. The semantics of the operation is simple—it may abort, but if it succeeds, the transaction cannot be forceably aborted any more. Each transaction that successfully executed the *irrevocable* operation is called an irrevocable transaction.

Definition 2.1. An *irrevocable* operation is an operation that can be called at most once by any transaction during its execution. The *irrevocable* operation can return either A (abort) or *ok* (success). If the return value is *ok*, the transaction can no longer be forceably aborted (i.e. no subsequent operation can return A instead of the expected value).

When the programmer wants to use operations with side-effect in the transaction, in order to guarantee that either each of the operation is executed exactly once or none is executed, he must call before the first operation with side-effect the *irrevocable* operation. In any real TM system this would most probably be added automatically before first operation that has a side-effect.

Some articles use a different model [2, 8], in which a transaction must a priori be labelled as irrevocable. The model described here is more popular, used at least in [1, 6, 7]. It is more flexible—it can emulate the other model (where the transactions must be tagged as irrevocable before they start) by requiring the first operation to be an *irrevcable* operation. Yet another approach to provide some support for irrevocability is delaying the execution of irrevocable code to the commit phase, this however forbids the programmer to read the values produced by the irrevocable part of code.

My work during the STSM focused on optimistic TM systems—TM systems that execute transactions disregarding if there are potential conflicts, and as soon as a conflict is detected, resolve it by aborting some transactions. The main characteristic of the optimistic approach is that a transaction can be forceably aborted. The other approach—pessimistic—detects the possibility of conflict before starting a transaction, and thus never forces an abort. This makes all operations performed by a pessimistic TM inherently irrevocable [4].

2.2 Reasoning about the correctness of a TM with irrevocable support

To reason abut the correctness of a TM, Guerraoui and Kapałka defined *opacity* [3]—a property that guarantees all transactions (regardless if finished or aborted) to be ordered and to see a consistent state of data. The definition (quoted from [3]) is as follows:

> A TM object M is opaque if, and only if, for every finite history H of M there exists a sequential TM history S equivalent to any completion of H , such that (1) S preserves the real-time order of H, and (2) every transaction T_i in S is legal in S.

The definition uses two key terms—real-time order and legality. For supporting irrevocable operation, the definition of opacity does not need any change. The meaning of condition (2) must however be altered. In a TM supporting irrevocable operations for a transaction to be legal in history H it must not only be legal in a TM that does not support irrevocability, but also take under account that if the transaction returned *ok* from the *irrevocable* operation, it must not be forceably aborted in H.

2.3 Impact of side-effects on correctness

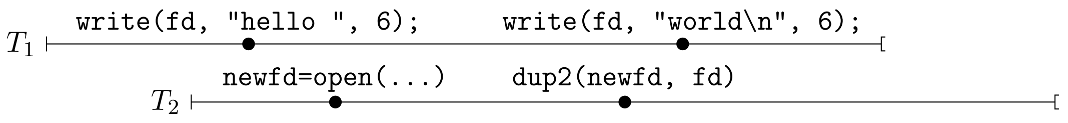
The conditions designed for a transactional memory to be correct take under account only the state of the TM system. The purpose of irrevocability is to allow the programmer to use operations that cause side effects, i.e., effects external to the TM, that cannot be rolled back or compensated. The operations with side effects may conflict with each other. However, for the TM, except from the need of calling the *irrevocable* operation, the operations with side effects are a subset of the local operations. The local operations are ignored as they do not operate on shared variables, thus may not cause any conflicts. The side effects, by definition, are not known to the TM system, neither the conflicts among them are. Running multiple irrevocable transaction in parallel can cause unexpected results due to conflicting side-effects.

To describe the problem, the following example is presented: read-only transactions T_1 and T_2 share only one variable, fd. The variable stores an Unix file descriptor. T_1 writes to the file, while T_2 changes file described by the descriptor.

```
transaction_T1() {
  write(fd, "hello ", 6);
  write(fd, "world\n", 6);
}
```

```
transaction_T2() {
  newfd = open("new.txt", O_RDWR, 0640);
  dup2(newfd, fd);
}
```

For the TM system, conflicts can happen only if two transaction access the same variable and at least one of the accesses is write. T_1 and T_2 do not conflict, thus the events can be executed in any order (preserving order inside the transactions of course), unless restricted by conflicts with other transactions. This makes possible the following history:



The programmer does not expect the world "hello" be in one file, and "world" in other, the TM system has however no knowledge of what `write` or `dup2` mean and cannot prevent this. The transactions here are no longer consistent, although the TM system cannot detect this.

One of the solutions is to allow at most one irrevocable transaction at a time. Other solutions would require the programmer to indicate all possible conflicts. This however would take down a serious advantage of the TM systems—the ease of use.

The problem described here is yet another issue to consider when replacing some synchronization mechanism by transactional memory.

2.4 The number of irrevocable transactions at a time

While the existing solutions for irrevocability in TM allow only one irrevocable transaction at any given point of time, no article provides the reader with a reason for it. In some papers this is referred to as a design choice, while other do not comment this assumption. Here I give the explanation why in general two irrevocable update transactions cannot make progress concurrently.

Theorem 2.2. *There is no TM that is opaque, wait-free and can support two parallel irrevocable transactions.*

Sketch of the proof: By contradiction, let's assume that there is a system that allows two transactions (T_i and T_k) to be live and irrevocable at the same time. Let's look at the example transactions depicted below:

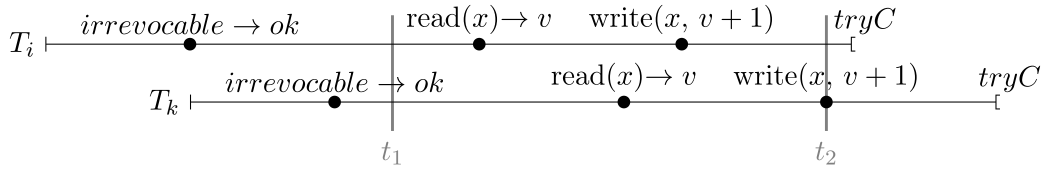


Figure 1: Example history for Theorem 2.2

The history H of the system at time t_1 is:
$\{inv_i(irrevocable), ret_i(irrevocable \to ok), inv_k(irrevocable), ret_k(irrevocable \to ok)\}$.

There is no shorter history that fulfils the assumptions (two live irrevocable transactions a time). The only other minimal histories fulfilling the assumptions would be the legal permutations of H, and the following reasoning holds for these histories as well.

The history of any two transactions that are irrevocable and live at the same time would be a superset of H (or its permutations), so showing that H is impossible, means that no other history is possible.

If the system allows H to happen, the situation depicted on Figure 1 is possible. As the TM is wait-free (according to the assumptions), no operation can wait for another, and since the transactions are irrevocable, neither of them is allowed to be forceably aborted.

But then the transaction can conflict at time $t_2 > t_1$, so either the transactions violate opacity (and atomicity)—this contradicts safety, or one of the transaction must be forceably aborted—this contradicts the transactions to be irrevocable.

The Theorem 2.2 assumes wait-freedom. Abandoning the wait-freedom, there exist transactions that can progress together as irrevocable transactions safely. For example, a write-only transaction can progress alongside an arbitrary irrevocable transaction and not violate correctness or irrevocability. However, there are limits for that.

Theorem 2.3. *There is no TM that is opaque, can support two (parallel) update irrevocable transactions T_1 and T_2, such that: a) T_1 reads any T-Var before T_2 commits and b) T_2 reads any T-Var before T_1 commits and c) both transactions eventually commit.*

Sketch of the proof: By contradiction: Assume that there exists such a TM. Take history H in which T_1 performs $read_1(x_1) \to v_1$ and T_2 performs $read_2(x_2) \to v_2$. Both events happen before any transaction commits (i.e. H contains no commit events). $read_2$ happens before T_1 commit (and vice versa) to fulfil the Theorem 2.3 assumptions. Now there exists an extension to the history that introduces a conflict between T_1 and T_2—e.g., $write_1(x_2, v'_2), write_2(x_1, v'_1)$. If both transactions were allowed to commit, one of them would violate opacity, however the theorem states that both must eventually commit. Reaching contradiction we conclude that such a TM cannot exist.

The Theorem 2.3 limits the parallelism of irrevocable transactions in any TM system. Update transactions that do not perform any reads are rare, and only such transactions can effectively progress concurrently.

The Theorems 2.2 and 2.3 show that the possible parallelism of update irrevocable transactions is very low (if any). In such case assuming that a TM can support at most one irrevocable update transaction a time is not limiting the performance or parallelism in any notable way.

The read-only transactions are a different story—using e.g., multiversioning, it is possible to guarantee that no conflicts arise. Thus a transaction is guaranteed to commit—so there is no limit on the number of irrevocable transactions a time. The conflict of side-effects cannot be however calculated, so the problem described in Section 2.3 remains valid.

2.5 Liveness of a TM with irrevocable support

A TM system, to support the irrevocable transactions, must at least:

a) return from some *irrevocable* operation the value *ok*,

b) eventually commit some irrevocable transaction.

A system that lacks any of these properties is not usable in any way. If a) does not hold, the system would be just an ordinary TM, as no transaction were able to become irrevocable. A TM missing b) would start some irrevocable transactions but would not let any of these commit, so the irrevocable transaction would never progress. This set of conditions describes the *minimal* liveness.

In a TM system that does not support irrevocability the liveness (or progressiveness) properties describe whether a transaction can make progress and commit. The irrevocability introduces two classes of transactions, and the liveness properties must be altered accordingly. As long as no irrevocable transaction is live, old definitions suffice. Additional properties must however be introduced to describe:

- if a transaction can transit to the irrevocable state,

- if an irrevocable transaction can finish,
- how the liveness of other transactions change if there is an irrevocable transaction.

From the new properties only the second is trivial—the fact, that for a TM system to be practical, every irrevocable transaction has to be able to commit. As there can be at most one update irrevocable transaction making progress, if some would not finish, no irrevocable transaction could start any more.

2.5.1 Transiting to the irrevocable state

The strongest liveness property regarding if a transaction can become irrevocable should state that a transaction must return *ok* from the *irrevocable* operation unless it would lead to compromising safety.

> If there is a non-empty group of transactions calling concurrently the *irrevcable* operation, then one of them (let's call it T_i) returns *ok* if *a)* no irrevocable transaction is live and *b)* T_i does not conflict with any committed or commit-pending transaction.

This property is achievable in a TM system (e.g., by the algorithm A_{CAS} presented later on), and there is no stronger property if one assumes that at most one irrevocable transaction can be live at the same time.

2.5.2 Liveness of revocable transactions while an irrevocable is live

As long as no irrevocable transaction is live, the liveness of revocable transactions can be classified using properties that stem from TM systems without irrevocability. However, when an irrevocable transaction is live, some classical properties cannot be applied, while other no longer describe what has been their intention. Also the behaviour of some existing TM systems supporting irrevocability cannot be described using properties for an ordinary TM. This imposes the need for new properties.

Many existing TM systems supporting irrevocability work so that when a irrevocable transaction is live, all other processing is halted [2, 8]. Other systems only allow read-only transactions while an irrevocable is live, forbidding concurrent update transactions [5]. It is possible however to build a TM system that allows non-conflicting update transactions to commit while an irrevocable is live. Some algorithms of this kind have been proposed in [6, 7].

To be able to tell how flexible the TM with irrevocable support is, one can use the following sentence:
If a irrevocable transaction is live:

- no (read only|update) revocable transactions can commit,
- some (read only|update) revocable transactions can commit,
- (all read only|not conflicting update) revocable transactions can commit.

A strong and still achievable in a wait-free system property, referring to progress of revocable transaction, can be formulated as follows:

> Any operation called by transaction T_k while an irrevocable operation T_i is live, must finish in a finite number of steps (even if T_i is not making progress). Moreover, any transaction T_k that will be forceably aborted while an irrevocable transaction T_i is live must:

- either conflict with some other revocable transaction
- or the write set of T_k intersects the read set of T_i at the moment of abort.

This property means that a transaction may be aborted only if a real conflict with an other transaction would exist.

2.6 Re-using existing properties

Some properties used to characterise progress in TM systems that do not support irrevocability can, after some extension, be used in systems that support irrevocability. Most progress conditions designed for systems without irrevocability are infeasible, as one of two totally not conflicting transactions may need to be aborted—if both want to become irrevocable. To workaround this, an artificial variable can be introduced—a variable x_{irr}, read from and written to by every *irrevocable* operation. This introduces conflict among all irrevocable transactions, thus eliminates the barrier preventing the use of some classical liveness properties.

With the x_{irr} variable, the *strong progressiveness* as defined by R. Guerraoui and M. Kapałka (Definition 12.2 in [3]) can be applied to any TM with irrevocable operations. Without the artificial variable *strong progressiveness* is not achievable, as while an irrevocable transaction T_i is live, no other transaction—including a not conflicting one as well—that calls *irrevocable* would not be able to commit, what is forbidden by *strong progressiveness.*

If x_{irr} were also be added at the beginning of each revocable transaction as $read(x_{irr})$, *strong progressiveness* would suit the model when no other transactions are allowed when an irrevocable transaction is live.

2.7 Cost of adding irrevocability

Once the properties characterising a TM system with irrevocable support have been defined, one should consider whether it is possible to create a system that can provide them, and if yes, estimate the cost of adding the irrevocability.

The Algorithm A_{CAS} is an opaque wait-free algorithm that provides the following guarantees:

- a transaction's *irrevocable* operation returns *ok* if a) there is no live irrevocable transaction and b) no concurrent transaction is becoming irrevocable and c) the transaction is not conflicting with a live transaction
- ensures that all not-conflicting transactions commit while an irrevocable is live.

```
Shared data
    /* Triple (owner, newValueL, oldValueL) or (owner, lockedValueL, IRR)  */
    $TVar[1,2,\ldots] \leftarrow [(0,0,0),(0,0,0),\ldots]$ : CAS objects;
    /* State of each transaction, one of: {live, aborted, irrevocable, committed}  */
    $State[0,1,2\ldots\ldots] \leftarrow$ [committed, live, live, ...] : CAS objects;
    $Mem[0,1,\ldots] \leftarrow [0,0,\ldots]$ : registers;    /* pointed by ... valueL, hold the values */
    $C \leftarrow 1$ : fetch-and-increment object;
    $irrTransaction \leftarrow \perp$ : CAS object;    /* ID of the irrevocable transaction */
    $irrAcquiring[1,2,\ldots] \leftarrow$ [false, false, ...];
Local data
    $k$ ;    /* Unique transaction identifier */
    $rset \leftarrow \varnothing$ : map[] ;    /* Read set. For $x_m$, key—$m$, value—last seen $TVar[m]$ */
    $wset \leftarrow \varnothing$ : map[] ;    /* Write set. (k-v as in $rset$) */
operation read($x_m$)
    if $irrTransaction == k$ then return readIrr($x_m$);
    if $m \in wset$ then return getValue($wset[m]$);
    if $m \in rset$ then return getValue($rset[m]$);
    $locator = Tvar[m]$;
    /* Validate - check if the readset is consistent  */
    foreach $m \to locator : rset$ do
        if not validate(m, locator) then return abort();
    if $locator.oldValueL ==$ IRR then
        $rset.add(m, locator)$;    /* Irrevocable transaction read $x_m$ */
        return $locator.newValue$;
    /* State can be live only if some transaction acquired $x_m$ for write  */
    if $state[locator.owner] ==$ live then return abort();
    $rset$.add($m, locator$);
    return getValue($locator$)
function getValue($locator$)
    if $state[locator.oldValueL] ==$ IRR then return $Mem[locator.newValueL]$;
    if $state[locator.owner] ==$ aborted then
        return $Mem[locator.oldValueL]$;
    else
        return $Mem[locator.newValueL]$;
function validate($m$, $locator$)
    $current \leftarrow TVar[m]$;
    if $current == locator$ then return true;
    /* If the irrevocable transaction read $x_m$, locator has changed but the
       value can still be correct  */
    if $current.oldValueL ==$ IRR and $current.newValueL ==$
      ($state[locator.owner] ==$ aborted?$locator.oldValueL : locator.newValueL$) then
        return true;
    return false;
```

Algorithm A_{CAS}(1/3)**:** Operations for revocable transactions

```
operation write($x_m$, $value$)
    if $irrTransaction == k$ then return writeIrr($x_m$, $value$);
    if $m \in wset$ then $Mem[wset[m].newValueL] \leftarrow value$;
    if not acquireWrite(m) then return abort();
    $Mem[wset[m].newValueL] \leftarrow value$;
function acquireWrite($m$)
    if $irrevocable[m] ==$ true then return false ;   /* Irr trans is acquiring $x_m$ */
    /* Has data been read, use locator from read set.  If not, from $TVar$ */
    $oldLocator \leftarrow (m \in rset?rset[m] : TVal[m])$;
    CAS($state[oldLocator.owner]$, live, aborted);
    if $state[oldLocator.owner] ==$ irrevocable then return false;
    $newLocator.owner \leftarrow k$;
    $newLocator.newValueL \leftarrow C$.incrementAndGet();
    $newLocator.oldValueL \leftarrow (state[oldLocator.owner] ==$ aborted?
                                   $oldLocator.oldValueL : oldLocator.newValueL)$;
    if not CAS($TVal[m]$, $oldLocator$, $newLocator$) then return false;
    $wset$.add($m, newLocator$);
    $rset$.remove($m$);
    return true;
operation commit()
    if $irrTransaction == k$ then return commitIrr();
    if CAS($state[k]$, live, committed) then
        return $C_k$;
    else
        return $A_k$;
operation abort()
    $state[k] \leftarrow aborted$;
    return $A_k$;
```

Algorithm A_{CAS}(2/3)**:** Operations for revocable transactions

```
operation irrevocable()
    if not $CAS(irrTransaction, \bot, k)$ then return abort();
    if not $CAS(state[k]$, live, irrevocable$)$ then {$irrTransaction \leftarrow \bot$; return abort();};
    local var $acquired \leftarrow \varnothing$ : map[];
    foreach $m \rightarrow locator : rset$ do
        $oldLocator \leftarrow$acquireIrrRead($m$);
        if $oldLocator == locator$ then
            acquired.add($m$,$locator$);
        else
            /* Cleaning up locks - needed for correctness */
            foreach $m \rightarrow locator : acquired$ do $TVar[m] \leftarrow locator$;
            $state[k] \leftarrow$ aborted;
            $irrTransaction \leftarrow \bot$;
            return $A_k$;
    return $ok$;
operation readIrr($x_m$)
    if $m \in wset$ then return getValue($wset[m]$);
    if $m \in rset$ then return getValue($rset[m]$);
    acquireIrrRead($m$);
    return getValue($rset[m]$);
operation writeIrr($x_m$, $value$)
    if $m \notin wset$ then
        if $m \notin rset$ then acquireIrrRead($m$);
        $newLocator \leftarrow Tvar[m]$;
        $newLocator.oldValueL \leftarrow newLocator.newValueL$;
        $newLocator.newValueL \leftarrow C$.incrementAndGet();
        $Tvar[m] \leftarrow newLocator$;
    $Mem[wset[m].newValueL] \leftarrow value$;
operation commitIrr()
    $state[k] \leftarrow$ committed;
    $irrTransaction \leftarrow \bot$;
    return $C_k$;
function acquireIrrRead($m$)
    $newLocator.owner \leftarrow k$;
    $newLocator.newValueL \leftarrow C$.incrementAndGet();
    $irrevocable[m] \leftarrow$ true;
    /* After $irrevocable[m]$ is true, only $p-1$ other processes can try to CAS
       the $TVar[m]$, so the loop will be repeated no more than $p$ times */
    repeat
        $oldLocator \leftarrow TVar[m]$;
        CAS($state[oldLocator.owner]$, live, aborted);
        $newLocator.oldValueL \leftarrow$ ($state[oldLocator.owner]$ == aborted?
                                $oldLocator.oldValueL$ : $oldLocator.newValueL$);
    until $CAS(TVar[m], oldLocator, newLocator)$;
    $irrevocable[m] \leftarrow$ false;
    return $oldLocator$;
```

Algorithm A_{CAS}(3/3)**:** Operations for irrevocable transactions

The algorithm A_{CAS} uses compare-and-swap operation, that provides an abstraction much stronger than needed to construct a TM system. As Guerraoui and Kapałka have proven in [3], to build a lock-based wait-free TM it is sufficient to use an abstraction of strong try-locks and registers.

Irrevocability support can be provided using only registers and strong try-locks. We present Algorithm $A_{try-locks}$ as a proof of concept. When using only abstractions as weak as strong try-locks and registers, the algorithms become more complex. Thus, to keep the algorithm short and easy to understand I decided to resign from supporting invisible reads in revocable transactions, what implicates weaker guarantees than provided by A_{CAS}.

The main problem to overcome when constructing a lock-based TM algorithm supporting irrevocable operations is to pass ownership for a shared T-variable x_m from a revocable transaction T_k to the irrevocable transaction T_i. In every algorithm, a transaction T_k at some point between issuing $write_k(x_m, v')$ and successfully finishing $tryC_k()$ must change the value of T-Variable to v' (assuming v' is the last written value by T_k to x_m). The scheduler, acting as an adversary, can yield the processor to other transactions just before the v' is copied to x_m. At any later time the adversary can wake up T_k and perform an unconditional write of v' to x_m, thus destroying any meantime changes. For an algorithm to be wait-free, it must not wait for T_k to perform the write, and must be able to let an irrevocable transaction T_i write to x_m. As the write done by T_k is inevitable, it must be turned to be harmless.

In Algorithm $A_{try-locks}(1/2)$ and $A_{try-locks}(2/2)$, I presented an algorithm in which passing (or rather hijacking) ownership is done the following way:

1. First T_i marks that it is going to use the variable x_m (in a register) and tries to tryLock the lock guarding x_m.

2. *If* tryLock succeeds, T_i has exclusive ownership over x_m $\square$. Otherwise there was one transaction holding the lock on x_m.

3. No subsequent transaction can access x_m until T_i commits, even if it successfully tryLock's x_m lock, as after successful tryLock the transaction checks if the irrevocable transaction marked x_m to be in use.

4. So there is a T_k holding lock on x_m, and it may at any time perform a write to x_m regardless if T_i is using x_m. In the algorithm, transactions use a working copy of all transactional variables.

5. T_k can write to the global copy only on commit. T_i tries to prevent commit of T_k. To enable this, T_k, as the first step of committing, must mark that it commits and must acquire a lock. If it fails to acquire the lock, it aborts.

6. *If* T_i wins the lock needed for T_k to commit, it also marks that the T_k is aborted. T_i can create its own working copy of x_m. $\square$

7. When T_i fails to win the lock and T_k is not marked as aborted, the T_k entered the commit phase and can no longer abort. T_k cannot modify its local copy any more (as it performs a commit). It will overwrite the global copy of x_m with its working copy to at some (unpredictable) moment of time.

8. T_i uses the working copy of x_m that belongs to T_k as its own working copy, so that the possibly pending copy will not overwrite the new value of x_m (T_k may perform the copy after T_i commits).

```
Shared data
    $TVar[1,\ldots]$ : registers$\leftarrow [0,0,\ldots]$ ;                    /* Transaction variables */
    $TVarWC[1,\ldots][1,\ldots]$ : registers ;                                /* Working copies of TVar */
    $irrevocableTransaction$ : strong try-lock$\leftarrow$ $unlocked$;
    $irrUsingVar[1,\ldots]$ : logical$\leftarrow$ [false, false, ...] ;        /* True if $T_{irr}$ uses $x_m$ */
    $comitting[1,\ldots]$ : logical$\leftarrow$ [false, false, ...];
    /* (↑) Set by any transaction before it commits */
    $comitted[1,\ldots]$ : logical$\leftarrow$ [false, false, ...];
    /* (↑) Set by any transaction after it commits */
    $aborted[1,\ldots]$ : logical$\leftarrow$ [false, false, ...];
    /* (↑) Set by irrevocable transaction as it aborts other transaction */
    $usingVariable[1,\ldots][1,\ldots]$ : logical$\leftarrow$ $all$ false;
    /* If $T_k$ uses $x_m$, $usingVar\ldots[k][m]$ is 1 (acquiring) or >1 ($version$) */
    $varLocks[1,\ldots]$ : strong try-lock[] $\leftarrow$ $unlocked$;          /* Protecting variables */
    $transLocks[1,\ldots]$ : strong try-lock[] $\leftarrow$ $unlocked$;        /* Protecting transactions */
    $version[1,\ldots]$ : int$\leftarrow [1,1,\ldots]$;                          /* Versions of the TVar */
    $irrComitting$ : logical $\leftarrow$ false ;                  /* True if an irr. trans. is committing */
    $hijackedTransactions$ : set $\leftarrow \emptyset$ ;                 /* Transactions that were hijacked */
Local data
    $locksHeld \leftarrow \emptyset$ ;                         /* Locks on variables held by the transaction */
    $hijacked \leftarrow \emptyset$;          /* Variables accessed by $T_{irr}$ that are not in $locksHeld$ */
    $id$ ;                                                                     /* ID of the transaction */
    $local[1,\ldots]$ : int ;                      /* For each variable, points to $TVarWC$ column */
    $amiIrrevocable$ : logical$\leftarrow$ false;
operation read($x_m$)
    if not $acquire(m)$ then return abort();
    $TVarWC[local[m]][m] \leftarrow TVar[m]$;
    return $TVarWC[local[m]][m]$;
operation write($x_m$, $v$)
    if not $acquire(m)$ then return abort();
    $TVarWC[local[m]][m] \leftarrow v$;
    return $ok$;
operation commit()
    $comitting[id] \leftarrow$ true;
    if $amiIrrevocable == true$ then $irrComitting \leftarrow$ true;
    if not $tryLock(transLocks[id])$ then return abort();
    foreach $m \in locksHeld \cup hijacked$ do
        $temp \leftarrow TVarWC[local[m]][m]$;
        $TVar[m] \leftarrow temp$;
        if $temp \neq TVarWC[local[m]][m]$ then
            if $irrComitting ==$ true then $TVar[m] \leftarrow TVarWC[local[m]][m]$;
        if $x \in locksHeld$ then unlock($varLocks[m]$);
        $usingVariable[id][m] \leftarrow 0$;
        if $amiIrrevocable$ then $irrUsingVar[m] \leftarrow$ false;
    $comitted[id] \leftarrow$ true;
    if $amiIrrevocable$ then release($irrevocableTransaction$);
    return $C_k$;
```

Algorithm $A_{try-locks}(1/2)$**:** Wait-free lock-based TM

```
operation abort()
    foreach $m \in lockHeld$ do
        unlock($varLocks[m]$);
        $usingVariable[id][x] \leftarrow 0$;
    return $A_k$;
operation irrevocable()
    if not tryLock($irrevocableTransaction$) then return abort();
    foreach $k : hijackedTransactions$ do
        if $comitted[k] ==$ true then $hijackedTransactions$.remove($k$);
    if $hijackedTransactions \neq \emptyset$ then
        release($irrevocableTransaction$);
        return abort();
    $irrComitting \leftarrow$ false;
    $amiIrrevocable \leftarrow$ true;
    return $ok$;
function acquire($m$)
    if $m \in locksHeld \cup hijacked$ then return true;
    if $amiIrrevocable ==$ true then return hijack($m$);
    if $irrUsingVar[m] ==$ true then return false;
    $usingVariable[id][m] \leftarrow 1$;
    if not tryLock($varLocks[m]$) then
        $usingVariable[id][m] \leftarrow 0$;
        return false;
    $locksHeld$.add($m$);
    if $irrUsingVar[m] ==$ true then return false;
    $version[m] \leftarrow version[m] + 1$;
    $usingVariable[id][m] \leftarrow version[m]$;
    $local[m] \leftarrow id$;
    $TVarWC[local[m]][m] \leftarrow TVar[m]$;
    return true;
function hijack($m$)
    $irrUsingVar[m] \leftarrow$ true;
    if tryLock($varLocks[m]$) then
        $locksHeld$.add($m$);
        $local[m] \leftarrow id$;
        $TVarWC[local[m]][m] \leftarrow TVar[m]$;
    else
        local var $version \leftarrow 0$;
        foreach $k : usingVariable[k][m] > 0$ do
            if tryLock($transLocks[k]$) then
                /* Transaction $T_k$ can no longer perform a commit */
                $aborted[k] =$ true;
            else if $comitting[k] ==$ true and $aborted[k] ==$ false then
                /* Transaction $T_k$ started a commit before tryLock */
                $hijackedTransactions$.add($k$);
                if $usingVariable[k][m] > version$ then
                    $version \leftarrow usingVariable[k][m]$;
                    $local[m] \leftarrow k$;
        if $version == 0$ then
            $local[m] \leftarrow id$;
            $TVarWC[local[m]][m] \leftarrow TVar[m]$;
        $hijacked$.add($m$);
    return true;
```

Algorithm $A_{try-locks}(2/2)$**:** Wait-free lock-based TM

9. If the copy done by T_k (that is a composition of load and store) will not be done atomically, it is being detected. If local and global copy differ after the move operation, T_k check if T_i started committing. If it is live—it will fix the global copy at its commit. Otherwise the copy is repeated (as no transaction can modify the local copy of x_m, it must succeed now).

10. As T_k may perform the second, unchecked copy at an unpredictable moment of time (after T_i commits), no new transaction can become irrevocable until T_k commits.

The algorithm $A_{try-locks}$ uses visible reads—thus it introduces read-read conflicts. Transforming the part of algorithm regarding revocable transactions by introducing invisible reads and a verify phase is possible; it has not been done in order to keep the algorithm simpler. The algorithm is wait-free and guarantees that revocable transactions that are not in conflict with the irrevocable transactions can commit. The algorithm however does not guarantee that a transaction that called *irrevcable* can become irrevocable even if it conflicts with none and there is no irrevocable operation—some transaction that has been stalled at commit can prevent it from becoming irrevocable.

The additional cost introduced by the algorithm is:

- a number of registers (of magnitude of the transaction and variable number)
- one strong try-lock object per each revocable transaction
- one strong try-lock object shared by all irrevocable transactions
- one additional tryLock call during each revocable transaction
- two additional tryLock calls during each irrevocable transaction

The number of tryLock calls for each transaction (without the overhead) is in the order of accessed transactional variables, so additional constant number of tryLock operation would be noticeable only if the majority of transactions were very short, which usually is not the case [1]. The memory requirements are not a big issue as well, so the overall cost of introducing irrevocability is not high.

2.8 Obstruction-freedom

Obstruction-freedom in the transactional memory systems is defined as follows (definition from [3]):

> We say that a transaction T_k in E, executed by a process p_i, encounters *step contention* in E, if there is a step of a process other than p_i in E after the first event of T_k and, if T_k is completed in E, before the commit or abort event of T_k.
>
> A TM shared object M is *obstruction-free* if in every implementation history E of M, for every transaction T_k in E, if T_k is forceably aborted, then T_k encounters step contention.

This cannot be guaranteed by any TM system with irrevocable operation support. If a live irrevocable transaction has been stalled, conflicting transactions must be prevented from committing to retain correctness. Even without step contention these transactions must be forceably aborted, violating the obstruction freedom.

2.9 Failures

The irrevocable transactions are meant always to commit. Speaking about a TM system, one must consider how the failures impact it. Avoiding any kind of failures in every irrevocable transaction is crucial, as when such transaction fails—either by crash or by becoming parasitic—it gravely impacts the liveness of TM system. Whereas with the possibility of detecting a failed transaction it is possible to restore liveness, without a perfect failure detector it is not possible to guarantee any progress after a failure. A variable accessed by an irrevocable transaction cannot be accessed until the owner commits, so if a failure prevents the commit, no transaction accessing the variable can progress.

3 Description of the main results obtained

Creating a model of TM systems with support of irrevocability includes not only defining its semantics, but also properties able to describe how well a particular TM satisfies the idea.

My work is based on the Guerraoui and Kapałka model, thus as the correctness criterion I chose opacity and adopted it to the modified model. For the TMs with irrevocability a separate discussion about correctness is additionally needed—e.g., discussion on how to solve the possible side-effect conflicts. The automatically undetectable conflicts limit in practise the number of concurrent irrevocable transactions to a single a time, unless the programmer has an option to mark possible parallelism among transactions. The gain in concurrency from such solution would still limited by the results concerning number of irrevocable transactions at a time—effectively no more than one update transaction can progress.

Having a correct, irrevocability capable TM, one needs to argue about its progress guarantees—to be able to tell if one TM system is more robust than another one. The semantic of irrevocable operations introduces the new division of the progress properties—as the transactions, until now indistinguishable, now are grouped into two classes: the revocable and the irrevocable transactions. Two additional classes of properties have been identified. One tells us how the revocable transactions behave when some irrevocable transaction is live—such properties are a must, as the current approaches range from running the irrevocable transaction in an exclusive mode to full parallelism among the irrevocable and revocable transactions. The second class of properties describing a TM system supporting irrevocability tell us what conditions suffice for a transaction to become irrevocable. This allows to further classify the TM system by describing the differences among possible approaches.

Except for proposing new properties for the TM systems with irrevocability, I have designed two algorithms showing that strong properties can be achieved—even by a wait-free algorithm, without a significant overhead compared to the algorithms offering no irrevocability support. Whereas some existing algorithms can be proven to have these properties as well, no wait-free algorithms of this kind were proposed so far.

4 Future collaboration with the host institution

The STSM started a collaboration between the host's and participant's institutes. The future work plans focus on exploiting and continuing the results achieved during the STSM.

5 Foreseen publications/articles resulting from the STSM

Both participant sides are currently discussing the opportunity of publishing a joint paper extending the results of this STSM.

6 Confirmation by the host institution of the successful execution of the STSM

Acknowledgements The project was funded by NCN grant 2012/06/M/ST6/00463. The Short Term Scientific Mission at EPFL was funded by COST Action IC1001.

References

[1] Lee Baugh and Craig Zilles. An analysis of I/O and syscalls in critical sections and their implications for transactional memory. In *IEEE International Symposium on Performance Analysis of Systems and Software 2008, ISPASS 2008*, pages 54–62. IEEE, 2008.

[2] Colin Blundell, E Christopher Lewis, and Milo M. K. Martin. Unrestricted transactional memory: Supporting I/O and system calls within transactions. Technical Report CIS-06-09, Department of Computer and Information Science, University of Pennsylvania, 2006.

[3] Rachid Guerraoui and Michał Kapałka. *Principles of Transactional Memory.* Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2010.

[4] Konrad Siek and Paweł T. Wojciechowski. Brief announcement: Towards a fully-articulated pessimistic distributed transactional memory. In *Proceedings of SPAA '13: the 25th ACM Symposium on Parallelism in Algorithms and Architectures (Montreal, Canada)*, July 2013.

[5] Michael Spear, Maged Michael, and Michael Scott. Inevitability mechanisms for software transactional memory. In *3rd ACM SIGPLAN Workshop on Transactional Computing*, 2008.

[6] Michael Spear, Michael Silverman, Luke Dalessandro, Maged M Michael, and Michael L Scott. Implementing and exploiting inevitability in software transactional memory. In *37th International Conference on Parallel Processing, 2008. ICPP'08*, pages 59–66. IEEE, 2008.

[7] Adam Welc, Bratin Saha, and Ali-Reza Adl-Tabatabai. Irrevocable transactions and their applications. In *Proceedings of the twentieth annual Symposium on Parallelism in Algorithms and Architectures*, pages 285–296. ACM, 2008.

[8] Ferad Zyulkyarov, Vladimir Gajinov, Osman S Unsal, Adrián Cristal, Eduard Ayguadé, Tim Harris, and Mateo Valero. Atomic quake: using transactional memory in an interactive multiplayer game server. In *ACM Sigplan Notices*, volume 44, pages 25–34. ACM, 2009.