

# NZTM: Nonblocking Zero-indirection Transactional Memory (Appendix)

Fuad Tabba  
The University of Auckland  
fuad@cs.auckland.ac.nz

Mark Moir  
Sun Microsystems  
mark.moir@sun.com

James R. Goodman  
The University of Auckland  
goodman@cs.auckland.ac.nz

Andrew W. Hay  
The University of Auckland  
andrewh@cs.auckland.ac.nz

Cong Wang  
The University of Wisconsin  
wang@cs.wisc.edu

## Appendix

This is an appendix containing more information on NZTM: Non-blocking Zero-indirection Transactional Memory [7], that did not make it into the paper due to space constraints. Namely, this appendix describes the read sharing algorithm we use in NZTM, and presents the PROMELA code used with SPIN to perform model-checking on NZSTM.

## 1. READ SHARING

### Read Sharing in NZSTM

As mentioned earlier, the issue of read sharing is largely independent of the NZSTM algorithm; different implementations of visible-reads [6], invisible-reads [2], or a mixture of the two [5] could work with NZSTM. In this section, we briefly describe the read sharing algorithm used in our experiments. This read sharing method is not necessarily the best to use with NZSTM; it is presented here as one possible implementation. This algorithm is inspired by the visible reads algorithm used in RSTM [6].

NZSTM uses visible reads, where competing reader and writer transactions can observe and abort each other. We use visible reads mainly because it makes it easier to integrate hardware and software transactions, as hardware transactions that need to acquire an object exclusively can easily identify potential conflicts with software transactions.

For our read sharing implementation, we have restructured the NZObject as in Figure 1. NZObject contains in place  $r$  slots for readers, and a saturation counter that keeps track of whether these slots have been occupied. When all slots are occupied, the next reader to come allocates a readers list consisting of  $P$  cache lines, where  $P$  is the total number of processors in the system. The reader then adds a pointer to its transaction to the cache line whose offset matches its own id number.

The algorithm for opening an object for read is as follows: Transaction  $T_r$  wants to open an object for reading;  $T_r$  checks for active writers that might currently own the object. If such a writer ex-

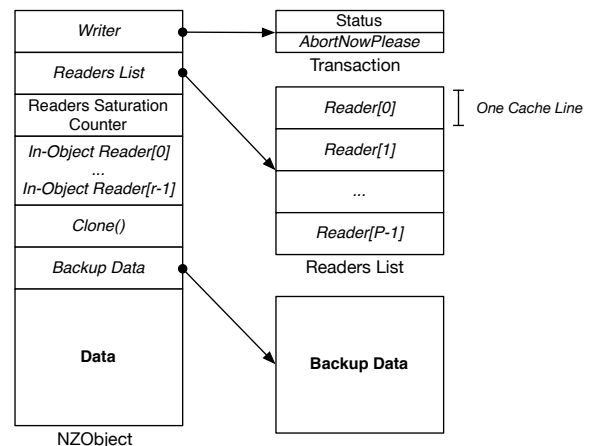


Figure 1: An NZObject using our proposed visible readers scheme. Each readers slot is one cache line big for performance.

ists and depending on the contention management policy,  $T_r$  either waits for the writer to finish, asks it to abort, or inflates the object.

Assuming there are no active writers, or that all potential conflicts have been resolved, the reader  $T_r$  checks the Readers Saturation Counter to see whether all in place reader slots have been occupied; this is indicated by the value of the counter being one more than the total number of available slots. If the counter is *not* saturated,  $T_r$  briefly acquires the object exclusively, by atomically swapping the Writer field to point to its own transaction, adds itself to the first slot not occupied by an active reader, and increments the saturation counter. Before  $T_r$  releases the object, it restores the backup data if the previous writer transaction aborted and Backup Data points to valid data (rather than NULL).

If the counter *is* saturated,  $T_r$  creates a new readers list, if one does not exist, by allocating memory for it and atomically setting the Readers List field to point to the newly created list. If a list does exist,  $T_r$  adds a pointer to its transaction to the appropriate cache line on that list, then  $T_r$  checks for any conflicts with active writers. If no active writers exist,  $T_r$  has succeeded in acquiring the object for read and can now access the data. If there are active writers,  $T_r$  refers to the contention management policy, and either waits, aborts the writer, or inflates the object.

Before  $T_r$  is done acquiring the object, it validates by checking its own AbortNowPlease flag, and aborts itself (relinquishing ownership of the object first if necessary) if the flag is set. This is to ensure that the set of objects the transaction has opened for read

are consistent: if  $T_r$  has been aborted due to a conflict with another transaction on another object while attempting to open this object for read, the reads may not be consistent, in which case it is not safe to return to user code [4].

Because many object-based systems execute in managed runtime environments that can catch and mask exceptions, this validation may be unnecessary in some cases. But given the complication involved in ensuring that all possible bad behavior that could arise from inconsistent reads (including infinite loops that do not raise any exceptions), and the low cost of validating in our system, we prefer to avoid relying on such mechanisms for now.

Validating at other times, while not strictly necessary, may be desirable for performance reasons. For example, validating before asking another transaction to abort and/or before waiting for another transaction that we have requested to abort may avoid unnecessary aborts and waiting.

Now if a transaction  $T_w$  is opening the object for writing, the procedure is not very different from what it would do assuming there were no read sharing. First  $T_w$  checks and resolves conflicts with any writer transactions. Then it acquires the object exclusively by using the `Writer` field in an analogous way to the `Owner` field in the exclusive version [?]NZTM-SPAA]. However, having acquired the object exclusively,  $T_w$  cannot proceed until it ensures that no active readers are currently accessing the object. If the saturation counter indicates that the in place slots are not saturated, the writer only needs to resolve conflicts with those readers that are using the in place slots. However, if the slots are saturated, then the writer needs to resolve conflicts with the in place readers and the whole readers list.

It is worth noting, that should a writer  $T_w$  decide to abort a reader transaction, it only asks it to abort by setting its `AbortNowPlease` flag and need *not* wait for the reader to acknowledge the abort. The writer can immediately access the object having ensured that all readers are either not active, or being asked to abort. The reason is, in this algorithm, every time a reader reads a value from an object, it validates by checking its own transaction status field to ensure it has not been asked to abort before using the read data. This ensures that the reader will only use consistent data, and makes the system nonblocking without the need to inflate for unresponsive readers, reducing even further the chances of an object being inflated.

Once a writer has resolved all potential conflicts with existing readers, and there are no more active readers accessing the object, the writer resets the saturation counter to 0. This ensures that future writer transactions do not need to traverse the readers list unless the counter saturates again. This is the only method for resetting the saturation counter in software transactions, as readers do not locally track objects they have acquired for reading.

Displacing data for non-responsive writers is the same as before. We have decided, for simplicity, that once an object is inflated, it can only be accessed exclusively until it is deflated — no read sharing is allowed. This is to make the algorithm simpler, as we believe this scenario is rare.

This treatment of read sharing favors readers over writers, which might be a good policy assuming readers significantly outnumber writers. Moreover, the overhead incurred on writers, if the number of readers is below the saturation threshold, is low, since writers only need to access the cache lines that contain the object and its metadata to resolve all conflicts.

## Read Sharing between NZTM and NZSTM transactions

In NZTM, the HyTM system that uses NZSTM for software transactions mentioned earlier, transactions can be attempted using

HTM and if (repeatedly) unsuccessful, are eventually retried using NZSTM software transactions.

When a transaction running using HTM opens an object, it checks for conflicts with software transactions, and explicitly aborts itself if any are discovered; it can then retry either in hardware or in software, according to decisions made by the contention manager. If no conflicts are discovered, the transaction can safely proceed because a subsequent conflict that arises with a software transaction will modify data that the hardware transaction has read, thereby aborting the hardware transaction.

Since NZSTM uses visible reads, it is easy for hardware transactions to detect the presence of potentially conflicting software transactions. A hardware transaction opening an object for reading only needs to check that the `Writer` field does not point to an active transaction, in which case it has to abort and try again. If it points to a committed or an aborted transaction, then it can proceed, restoring any available backup data in case of an aborted transaction.

A hardware transaction opening an object for writing performs the same check on the `Writer` mentioned above. If successful, it also needs to ensure that there are no conflicts with any software readers. If the `Readers Saturation Counter` is zero, then the procedure is done. If the counter is greater than zero, the hardware transaction must check for the existence of active transactions in the in place reader's slots, and the reader's list if the counter is saturated. Once it has ensured that none of the software readers are active, it resets the `Readers Saturation Counter` to zero, eliminating the need for subsequent hardware transactions to perform similar checks. Finally, the hardware transaction sets the `Writer` field to `NULL` if it is not already.

## 2. PROMELA MODEL

This appendix contains the source code of the PROMELA model used for model checking NZSTM. A quick PROMELA primer can be found at <http://spinroot.com/spin/Man/Quick.html>; alternatively, refer to Holzmann's work [3] for a more complete reference.

An advantage to having the PROMELA source code, since it is a high-level model description, is that it is easier to follow the details of the NZSTM algorithm than the C implementation.

The model presented here covers all combinations of the blocking version of NZSTM, the nonblocking NZSTM, and the read sharing algorithm presented in Appendix 1. Some of the ideas and techniques used in writing this model were either taken from, or inspired by, Ananian's PROMELA model [1].

```

/**
 * @file nztm.pml
 * Promela model for NZSTM
 *
 * @author Fuad Tabba (fuad at cs.auckland.ac.nz)
 *
 * Some ideas taken from/inspired by C.S. Ananian's Promela model:-
 *
 * C.S. Ananian, Architectural and Compiler Support for Strongly Atomic
 * Transactional Memory, 2007.
 * http://flex-compiler.csail.mit.edu/Harpoon/swx.pml
 *
 * Goals:
 * - Only use atomic statments (atomic, d_step) in scenarios where the
 *   operation can be made to look atomic using nothing more than CAS
 * - Only one process can modify an object at a time
 * - Multiple processes can read an object at a time
 * - Consistency must be preserved
 * - All threads must reach the end of the code (no deadlocks)
 * - Livelocks within the algorithm itself are not allowed. Meaning each
 *   transaction must eventually either commit or abort. Livelocks in the
 *   larger sense are possible though (i.e. everyone aborting), and that is
 *   handled using good contention management.
 *
 * Current Issues and Bugs:-
 * - none known
 *
 * Minor Issues:-
 * - reduce the state-space size
 * - some lines are slightly over 80 characters in width
 */

/*
 * =====
 * Global Constants
 * =====
 */

/* If defined then visible reads are used, otherwise it's exclusive reads */
#define READ_VISIBLE

/* If defined then the blocking model is used */
// #define BLOCKING

/**
 * Number of processes running in the system.
 */
#define PROCESSES 3

/**
 * Maximum number of transactions the system can create
 *
 * Must have one extra transaction since a txn identifier of 0 is like
 * a NULL pointer. In practice, 0 points to a committed transaction that
 * all newly created objects point to.
 */
#define TOTAL_TXN 10

/**
 * Number of objects to preallocate.
 */
#define TOTAL_OBJECTS 3

/**
 * Maximum number of transactions per thread
 */
#define TXN_PER_PROC ((TOTAL_TXN - 1)/PROCESSES)

/*
 * Stuff to help with debugging and validating

```

```

*/
/**
 * A number that represents an invalid state. i.e. uninitialized memory
 *
 * To ensure it's never used, must be bigger than all arrays. That way, any
 * errors would be detected by spin's bound checking (indexing error).
 */
#define INVALID 77

/**
 * Represents NULL pointers. Just to make it clearer that the value we're
 * assigning NULL to represents a pointer rather than a proper value.
 */
#define NULL 0

/**
 * The value the data in the objects is initialized to.
 */
#define INITV 13

/*
 * =====
 * Structure Definitions
 * =====
 */

/**
 * This data structure represents the pointer to the oldData.
 */
typedef BackupData
{
    /**
     * ID Number for this version of the backup.
     *
     * This information is stored implicitly in NZTM since the pointer itself
     * acts as an ID number. To ensure uniqueness, and without making the system
     * state space explode, always set its value to the current txid
     *
     * If it's set to NULL it means there's no backup copy.
     */
    byte id = NULL;

    /**
     * The backup copy (if any)
     */
    byte backup = INVALID;
};

/**
 * Used to keep track of the data after the object has been inflated.
 *
 * This is the mechanism that enabled NZTM to be nonblocking.
 */
typedef Locator
{
    /**
     * Points to the transaction that owns this locator
     */
    byte owner = INVALID;

    /**
     * Points to the nonresponsive transaction that caused the object to be
     * inflated.
     * A NULL (0) value indicates that the object was inflated because of a
     * nonresponsive reader.
     */
    byte abortedTxn = INVALID;

    /**
     * Points to the actual value of object, if transaction is aborted; and a

```

```

    * backup copy of the value when the transaction is active
    */
    byte old = INVALID;

/**
 * Points to the actual of object if transaction is committed;
 * Tentative value, if transaction is active.
 */
    byte new = INVALID;
};

/**
 * The main data structure used in NZIM.
 *
 * This maintains the data and the metadata required for proper access
 */
typedef NZObject
{
    /**
     * A "pointer" to the writer (or exclusive) transaction (if the object is
     * not inflated), OR a pointer to the locator that contains the object (if
     * the object is inflated)
     *
     * A value of zero indicates that no one is pointing to it (assuming it's
     * not inflated)
     * Must be changed atomically.
     */
    byte owner = NULL;

    /**
     * Indicates whether this object has been inflated, thereby the data is
     * reachable through the locator.
     *
     * If this value is true, then the owner field points to the locator.
     *
     * In C, this value is actually represented by the least significant bit of
     * the transaction field. Therefore it can be manipulated atomically with it
     */
    bool isInflated = false;

    /**
     * The actual data in place
     */
    byte data = INITV;

    /**
     * A "pointer" to the backup data
     */
    BackupData old;

    /**
     * Visible list of readers reading this object.
     *
     * In NZIM it's a different structure, but since it's allocated at the same
     * time as the object we'll consider it as part of it.
     */
    byte readers[PROCESSES] = NULL;

    /**
     * Preallocated Locators.
     *
     * A locator is preallocated for every process. It's done this way since
     * dynamic memory allocation is not supported in the spin models. Moreover,
     * the maximum number of locators possible is 1 per process per transaction.
     */
    Locator locators[PROCESSES];
};

/**
 * The different states a transaction can be in.
 *

```

```

* ACTIVE: The transaction is running and hasn't been asked to abort. If it
* tries to commit it would succeed, and its status would be COMMITTED.
* COMMITTED: The transaction has finished successfully, it's changes are
* permanent.
* ABORTED: The transaction has finished but failed. Its changes are rolled
* back.
* ABORT_NOW: The transaction is still running but it has been asked to abort.
* If it tries to commit it would fail, and its status would be ABORTED.
*
* NOTE: In the paper we mention that when a transaction has been asked to
* abort its least significant bit is set. In Promela, this is modeled as
* a new state.
*/
mtype = {ACTIVE, COMMITTED, ABORTED, ABORT_NOW};

/**
 * Definition of a transaction.
 *
 * There must be one unique location per transaction, do not recycle.
 */
typedef Transaction
{
    mtype status = COMMITTED;
};

/*
 * =====
 * Function (and macro) definitions
 * =====
 */

/*
 * Transactional functions
 */

/**
 * Determines whether a transaction is active
 */
#define TXACTIVE(x) (transactions[x].status == ACTIVE)

/**
 * Determines whether a transaction finished successfully
 */
#define TXCOMMITTED(x) (transactions[x].status == COMMITTED)

/**
 * Determines whether a transaction is failed to commit (and knows that)
 */
#define TXABORTED(x) (transactions[x].status == ABORTED)

/**
 * Determines whether a transaction has been asked to abort
 */
#define TXABORTING(x) (transactions[x].status == ABORT_NOW)

/**
 * Transaction definitions: defined local variables that are needed by all
 * processes, to be defined per process.
 *
 * owner: Points to my transaction
 * enemyid: Points to the current enemy transaction
 * oldData: scratch variable used to compare values of backup data
 * lid: keep track of the current locator's id number
 * isInflated: true if the last object opened is inflated
 * success: resembles a return value of whether the function was successful
 */
#define TXN_DEFS() \
    byte txid = NULL; \
    byte enemyid = NULL; \
    BackupData oldData; \
    byte lid = INVALID; \
    bool isInflated = false; \

```

```
byte dummy;
```

```
/**  
 * Begins a new transaction.  
 *  
 * Each process can allocate transactions from predefined slots in a particular  
 * order. This is a modeling restriction to reduce the state-space size.  
 *  
 * txid is the variable that maintains the value of the pointer to the  
 * transaction. Must always be the same variable for every process. Initially  
 * must be 0.  
 */
```

```
inline beginTransaction()  
{  
    /* Atomic since setting pointers in C appear that way */  
    d_step {  
        if  
        :: (txid == NULL) -> txid = _pid * TXN_PER_PROC + 1;  
        :: else -> txid++;  
        fi;  
  
        /* Ensure that we're in the correct range */  
        assert(txid <= _pid * TXN_PER_PROC + TXN_PER_PROC);  
  
        /* Redundant: Stress that txid cannot be 0, since it's reserved */  
        assert(txid != 0);  
  
        /* Ensure that we have enough to handle all processes */  
        assert(_pid < PROCESSES);  
  
        /* Ensure we're allocating from a fresh transaction*/  
        assert(TXCOMMITTED(txid));  
  
        transactions[txid].status = ACTIVE;  
    }  
}
```

```
/**  
 * Attempts to commit a transaction.  
 */
```

```
inline commitTransaction()  
{  
    d_step {  
        assert(TXACTIVE(txid) || TXABORTING(txid));  
  
        if  
        :: TXACTIVE(txid) -> transactions[txid].status = COMMITTED;  
        :: else -> transactions[txid].status = ABORTED;  
        fi;  
    }  
}
```

```
/**  
 * Validates my current transaction, ensures that I'm still active.  
 *  
 * Aborts if I've been asked to abort.  
 */
```

```
inline validateTransaction()  
{  
    atomic {  
        /*  
        * Doesn't need to be atomic. Done that way to ensure that the assertion  
        * takes place at the correct instance. Other than the assertion, being  
        * atomic does not affect behaviour.  
        */  
        if  
        :: TXABORTING(txid) -> goto aborted;  
        :: else -> assert(TXACTIVE(txid));  
        fi;  
    }  
}
```

```

}

/**
 * Aborts an enemy transaction atomically.
 *
 * Atomically changes the state of the transaction from ACTIVE to ABORT_NOW.
 * Do not use to abort self.
 */
inline abortEnemy(enemyid)
{
    d_step {
        if
        :: TXACTIVE(enemyid) -> transactions[enemyid].status = ABORT_NOW;
        :: else;
        fi;

        assert(enemyid != txid);
        assert(!TXACTIVE(enemyid));
    }
}

/**
 * Blocks until the enemy acknowledges an abortion or that I have been aborted
 * myself.
 *
 * This is the only blocking portion in the *blocking version* of NZSTM
 */
inline blockAborted(enemyid)
{
    (!TXABORTING(enemyid) || !TXACTIVE(txid));
}

/**
 * Handles a conflict with another transaction.
 *
 * @param eid The id of the transaction there's a conflict with
 */
inline resolveConflict(eid)
{
    /*
     * Contention managers will eventually abort any enemy.
     * The way spin operates, this will encompass all possibilities, from waiting
     * for the enemy to finish by itself, to actually aborting it.
     */
    abortEnemy(eid);
}

/**
 * Loops through every reader in the readers list, and performs an action if the
 * reader isn't me on ALL the readers.
 *
 * Done in this awkward manner to reduce the state-space size
 */
#define LOOP_READERS(action) \
    if \
    :: (PROCESSES > 4 && _pid != 4) -> \
        action(objects[oid].readers[4]); \
    :: else; \
    fi; \
    if \
    :: (PROCESSES > 3 && _pid != 3) -> \
        action(objects[oid].readers[3]); \
    :: else; \
    fi; \
    if \
    :: (PROCESSES > 2 && _pid != 2) -> \
        action(objects[oid].readers[2]); \

```

```

:: else;
fi;

if
:: (PROCESSES > 1 && _pid != 1) ->
    action(objects[oid].readers[1]);
:: else;
fi;

if
:: (PROCESSES > 0 && _pid != 0) ->
    action(objects[oid].readers[0]);
:: else;
fi;

```

```

/**
 * Handles a conflict with the readers list (visible reads).
 *
 * @param oid The 'pointer' to the object with the readers list
 */

```

```

inline resolveReadersConflicts(oid)
{
    /*
     * Resolve conflicts with all readers
     */
    LOOP_READERS(resolveConflict);
}

```

```

/**
 * Models waiting for the whole readers list or being impatient and
 * deciding to inflate it.
 */

```

```

inline waitOrInflateReaders(oid)
{
#ifdef BLOCKING
    LOOP_READERS(blockAborted);
#else /* nonblocking */
    if
    :: (PROCESSES > 4 && _pid != 4 && TXABORTING(objects[oid].readers[4])) ->
        inflateObjectReader(oid);

    :: (PROCESSES > 3 && _pid != 3 && TXABORTING(objects[oid].readers[3])) ->
        inflateObjectReader(oid);

    :: (PROCESSES > 2 && _pid != 2 && TXABORTING(objects[oid].readers[2])) ->
        inflateObjectReader(oid);

    :: (PROCESSES > 1 && _pid != 1 && TXABORTING(objects[oid].readers[1])) ->
        inflateObjectReader(oid);

    :: (PROCESSES > 0 && _pid != 0 && TXABORTING(objects[oid].readers[0])) ->
        inflateObjectReader(oid);

    :: else; /* No inflation */
    fi;
#endif /* BLOCKING */
}

```

```

/**
 * Breaks out of an outer loop if there are nonresponsive readers.
 *
 * Used when trying to deflate an object.
 */

```

```

inline breakNonresponsiveReaders(oid)
{
    if
    :: (PROCESSES > 4 && _pid != 4 && TXABORTING(objects[oid].readers[4])) ->
        break;

    :: (PROCESSES > 3 && _pid != 3 && TXABORTING(objects[oid].readers[3])) ->

```

```

        break;
    :: (PROCESSES > 2 && _pid != 2 && TXABORTING(objects[oid].readers[2])) ->
        break;
    :: (PROCESSES > 1 && _pid != 1 && TXABORTING(objects[oid].readers[1])) ->
        break;
    :: (PROCESSES > 0 && _pid != 0 && TXABORTING(objects[oid].readers[0])) ->
        break;
    :: else;
    fi;
}

/*
 * BackupData Functions
 */

/**
 * Tests whether there's a backup copy of the data
 */
#define IS_OLD(oid) (objects[oid].old.id)

/*
 * Inflated locator functions
 */

/**
 * Derefernces the locator associated with this particular id
 *
 * @param lid The locator id that we want dereferenced
 */
#define LOCATOR(oid, lid) objects[oid].locators[lid]

/**
 * Sets lid to the id of the locator if the object is inflated. If the object
 * isn't inflated, sets the locator id to an invalid value.
 *
 * @param oid The id of the possibly inflated object
 * @param lid Where to store the id of the locator
 */
inline findLocator(oid, lid)
{
    /* Atomic since the two fields are overloaded into one field in NZTM */
    d_step {
        if
        :: (objects[oid].isInflated) -> lid = objects[oid].owner;
        :: else -> lid = INVALID;
        fi;
    }
}

/**
 * Inflated an object owned by a nonresponsive transaction.
 *
 * Only to be used by a writer to a nonresponsive reader that wants to inflate.
 *
 * @param oid The 'pointer' to the NZObject I want to inflate
 */
inline inflateObject(oid)
{
    validateTransaction();

    /* Find the possibly nonresponsive transaction */
    d_step {
        enemyid = objects[oid].owner;
        isInflated = objects[oid].isInflated;

        /* This is used for hung writers, so the enemy cannot be me */

```

```

    assert(isInflated || enemyid != txid);
}

/* Remember the value of the old data pointer */
d_step {
    oldData.backup = objects[oid].old.backup;
    oldData.id = objects[oid].old.id;
};

/* If the old data pointer doesn't exist, copy the in place data */
if
:: (oldData.id == NULL) -> oldData.backup = objects[oid].data;
:: else;
fi;

/* Create and initialize the locator */
d_step {
    LOCATOR(oid, _pid).old = oldData.backup;
    LOCATOR(oid, _pid).new = oldData.backup;
    LOCATOR(oid, _pid).owner = txid;
    LOCATOR(oid, _pid).abortedTxn = enemyid;
}

/*
 * Check that all the assumption I've made still hold:-
 * - There is a nonresponsive transaction
 * - The object is not inflated
 * - The value of the old pointer is the same as it was before
 * - I am still active
 */
if
:: (!TXABORTING(enemyid) || isInflated ||
    oldData.id != objects[oid].old.id || !TXACTIVE(txid)) ->
    enemyid = INVALID; /* (return fail) Read note at end of function */

:: else ->
    /* Attempt to install the newData locator to inflate the object */
    d_step {
        if
        :: (objects[oid].owner == enemyid && !objects[oid].isInflated) ->
            objects[oid].isInflated = true;
            objects[oid].owner = _pid;
            isInflated = true;

        :: else;
        fi;
    }

    /* Check if I was successful in inflating the object */
    if
    :: (objects[oid].owner == _pid && objects[oid].isInflated) ->
#ifdef READ_VISIBLE
        /* Before proceeding, resolve all conflicts with readers */
        resolveReadersConflicts(oid);
#endif /* READ_VISIBLE */
        break; /* Breaks out of the open loop (i.e. return success) */

    :: else -> enemyid = INVALID; /* (i.e. return fail) */
    fi;

    /*
     * NOTE: I am overloading the enemyid field to also indicate a failed
     * inflation. This is done to keep the state space small. In C, it
     * would return either TRUE or FALSE to indicate success or failure.
     */
    fi;
} /* inflateObject(oid) */

/**
 * Inflated an object being visibly read by a nonresponsive transaction.
 * Assumes that I have already acquired the object exclusively.
 */

```

```

* I can only leave this if I got aborted.
* Only used with Visible Reads.
*
* @param oid The 'pointer' to the NZObject I want to inflate.
*/
inline inflateObjectReader(oid)
{
    validateTransaction();

    oldData.backup = objects[oid].data; /* A direct copy of the data*/

    /* Create and initialize the locator */
    d_step {
        /* Sanity Checks */
        assert(!IS_OLD(oid));
        assert(objects[oid].owner == txid || objects[oid].isInflated);

        LOCATOR(oid, _pid).old = oldData.backup;
        LOCATOR(oid, _pid).new = oldData.backup;
        LOCATOR(oid, _pid).owner = txid;
        LOCATOR(oid, _pid).abortedTxn = NULL; /* Hung on a reader */
    }

    /* Attempt to install the newData locator to inflate the object */
    atomic {
        if
            :: (objects[oid].owner == txid && !objects[oid].isInflated) ->
                objects[oid].isInflated = true;
                objects[oid].owner = _pid;
                isInflated = true;

            :: else -> assert(TXABORTING(txid));
        fi;
    }

    /* Check if I was successful in inflating the object */
    if
        :: (objects[oid].owner == _pid && objects[oid].isInflated) ->
            /* Before proceeding, resolve all conflicts with readers */
            resolveReadersConflicts(oid);
            break; /* Breaks out of the open loop */

        :: else -> /* I must have been aborted */
            assert(TXABORTING(txid));
            goto aborted;
    fi;
} /* inflateObjectReader(oid) */

/**
 * Opens an inflated object exclusively (used for both read and write)
 *
 * @param oid The 'pointer' to the NZObject I want to open
 */
inline inflatedOpenExclusive(oid)
{
#ifdef READ_VISIBLE
    validateTransaction();

    /*
     * Before proceeding, resolve all conflicts with readers. This is important
     * since the transaction that started the inflation process might have not
     * gotten around to doing this.
     */
    resolveReadersConflicts(oid);
#endif /* READ_VISIBLE */

do
    :: TXABORTING(txid) -> goto aborted; /* been asked to abort */

    :: else ->
        findLocator(oid, lid);

```

```

/* Check if the object is inflated */
if
:: (lid == INVALID) -> break;
:: else -> enemyid = LOCATOR(oid, lid).owner;
fi;

/* See if it's already open by me */
if
:: (enemyid == txid) -> break;
:: else;
fi;

resolveConflict(enemyid);
validateTransaction();

/* Create a new locator */
d_step {
    LOCATOR(oid, _pid).owner = txid;
    LOCATOR(oid, _pid).abortedTxn = LOCATOR(oid, lid).abortedTxn;

    if
    :: TXCOMMITTED(objects[oid].locators[lid].owner) ->
        LOCATOR(oid, _pid).old = LOCATOR(oid, lid).new;
    :: else ->
        LOCATOR(oid, _pid).old = LOCATOR(oid, lid).old;
    fi;

    LOCATOR(oid, _pid).new = LOCATOR(oid, _pid).old;
};

/* Try to install the new locator */
d_step {
    if
    :: (objects[oid].owner == lid && objects[oid].isInflated) ->
        objects[oid].owner = _pid;
        isInflated = true;

    :: else;
    fi;
}

/* If I acquired the object, try to deflate it (step 1 below) */
if
:: (objects[oid].owner == _pid && objects[oid].isInflated) ->
    deflateObject(oid);
    break;

:: else;
fi;
od;
} /* inflatedOpenExclusive(oid) */

```

```

/**
 * Attempts to deflate an object that has been inflated
 *
 * 1. Acquire the object the DSTM way.
 *
 * 2. Read and remember the value of the NZObject's oldData pointer.
 *
 * 3. Check if the transaction we're waiting on to abort has finally aborted, if
 * not then just proceed with the normal DSTM acquisition stuff. Make sure to
 * check for readers as well.
 *
 * (If Visible Reads) also check that all the transactions in the read list are
 * responsive, if not then just proceed with the normal DSTM acquisition stuff.
 *
 * 4. Check that the current transaction is still Active (meaning that it still
 * has the object acquired). If not, then proceed with the normal I'm aborted
 * stuff.
 *
 * 5. CAS the NZObject's oldData pointer to point to the real current data. If

```

```

* it fails abort.
*
* 6. CAS the NZObject's transaction pointer to point to my transaction.
* If the CAS was successful, then copy the old data back to the data.
*/
inline deflateObject(oid)
{
    /* I would have already taken place */

    do
    :: true -> /* Pseudo-infinite Loop */

        enemyid = LOCATOR(oid, _pid).abortedTxn;

        /* 2 */

        /* Atomic since in C it's one pointer field */
        d_step {
            oldData.id = objects[oid].old.id;
            oldData.backup = objects[oid].old.backup;
        }

        /* 3 */
#ifdef READ_VISIBLE
        if
        :: TXABORTING(enemyid) -> break;
        :: else -> breakNonresponsiveReaders(oid);
        fi;
#else /* Exclusive Reads*/
        if
        :: TXABORTING(enemyid) -> break;
        :: else;
        fi;
#endif /* READ_VISIBLE */

        /* 4 */
        validateTransaction();

        /* 5 */
        atomic {
            assert(enemyid == NULL || TXABORTED(enemyid));

            if
            :: (objects[oid].old.id == oldData.id) ->
                objects[oid].old.id = txid;
                objects[oid].old.backup = LOCATOR(oid, _pid).new;

            :: else ->
                /*
                 * A nonresponsive transaction that's responsive again might
                 * wake up and decide to take a backup copy. Doesn't
                 * necessarily mean I was aborted.
                 */
                break;
            fi;
        }

        /* 6 */
        atomic {
            if
            :: (objects[oid].owner == _pid && objects[oid].isInflated) ->
                objects[oid].owner = txid;
                objects[oid].isInflated = false;
                isInflated = false;

            :: else ->
                assert(TXABORTING(txid));
                goto aborted;
            fi;
        }

        objects[oid].data = objects[oid].old.backup;
        break;

```

```

    od;
} /* deflateObject(oid) */

/*
 * NZObject functions
 */

/* Defined opening macros for different kind of reads */

#ifdef READ_VISIBLE
#define OPEN_WRITE(oid) openWriteVisible(oid)
#define OPEN_READ(oid) openReadVisible(oid)
#else /* Exclusive Reads */
#define OPEN_WRITE(oid) openWriteExclusive(oid)
#define OPEN_READ(oid) openReadExclusive(oid)
#endif /* READ_VISIBLE */

/**
 * Creates a backup copy of the data and points the old pointer to that copy.
 */
inline cloneData(oid)
{
    /* Done atomically since setting a pointer in C appears to be atomic */

    d_step {
        objects[oid].old.id = txid;
        objects[oid].old.backup = objects[oid].data;
    }
}

/**
 * Restores the data pointed to by the old pointer. (as it was taken for backup)
 */
inline restoreBackupData(oid)
{
    /* Should not be atomic since it doesn't appear that way in C */

    objects[oid].data = objects[oid].old.backup;
    objects[oid].old.id = NULL;
}

/**
 * Atomically changes the transaction pointer to point to my transaction, iff
 * the object is acquired by who I think it is.
 */
inline casTransaction(oid, eid)
{
    d_step {
        if
            :: (objects[oid].owner == eid && !objects[oid].isInflated) ->
                objects[oid].owner = txid;
                isInflated = false;

        :: else;
        fi
    }
}

/**
 * Acquires the object exclusively (as if using CAS)
 *
 * Clears unnecessary old data pointers as well...
 */
inline acquireExclusive(oid, eid)
{
    /*
     * Nonblocking
     * Problem: I acquire the object, which belonged to a committed txn,

```

```

* and old points to stale data. Before I clear the old pointer, I
* get aborted. The txn that aborted me now thinks that the old pointer
* actually points to the real data rather than the state one.
*
* Possible solution: before acquiring the object, clear the old data
* pointer using CAS.
*
* Doesn't acquire the object if clearing the old pointer fails.
*/

/* Atomic since in C it's one pointer field */
d_step {
    oldData.id = objects[oid].old.id;
    oldData.backup = objects[oid].old.backup;
}

if
:: (oldData.id && objects[oid].owner == enemyid && TXCOMMITTED(enemyid)
    && !objects[oid].isInflated) ->
    atomic {
        if
        :: (objects[oid].old.id == oldData.id) ->
            objects[oid].old.id = NULL;
            objects[oid].old.backup = INVALID;

        :: else;
        fi;
    }

    if
    :: (!objects[oid].old.id) -> casTransaction(oid, eid);
    :: else;
    fi;

:: else -> casTransaction(oid, eid);
fi;
}

/**
 * Releases the object from my transaction (atomically, if I have it)
 */
inline releaseObject(oid)
{
    d_step {
        if
        :: (objects[oid].owner == txid && !objects[oid].isInflated) ->
            objects[oid].owner = NULL;
        :: else;
        fi;
    }
}

/**
 * Models waiting for an enemy then inflating the object if it's not responsive.
 *
 * @param eid The enemy transaction I'm waiting to abort.
 * @param oid The 'pointer' to the object that might inflate
 */
inline waitOrInflate(eid, oid)
{
#ifdef BLOCKING
    /*
     * Wait until the enemy has finished (or I have been aborted)
     */
    blockAborted(eid);
#else /* Nonblocking */
    /*
     * If the enemy is not responsive then inflate. This models both waiting for
     * the object to abort and being impatient since spin tries all possible
     * interleavings of instructions.
     */
#endif
}

```

```

    if
    :: TXABORTING(eid) -> inflateObject(oid);
    :: else;
    fi;
#endif /* BLOCKING */
}

/**
 * Opens the object exclusively for writing. Goes to Label aborted if it fails.
 *
 * Doesn't check for readers in the readers list
 *
 * @param oid The 'pointer' to the object I want to open
 */
inline openWriteExclusive(oid)
{
    if
    :: (objects[oid].owner == txid && !objects[oid].isInflated) ->
        validateTransaction();
        isInflated = false;

    /*
     * Check where there's a backup copy of the data or not,
     * which would be the case if I had the object open for read.
     */
    if
    :: (!IS_OLD(oid)) -> cloneData(oid); /* was open for read - upgraded */
    :: else; /* Already open for write */
    fi;

    :: else ->
    do
    :: TXABORTING(txid) -> goto aborted; /* been asked to abort */

    :: else ->
        d_step {
            enemyid = objects[oid].owner;
            isInflated = objects[oid].isInflated;
        }

    if
#endif BLOCKING /* nonblocking */
    :: isInflated ->
        inflatedOpenExclusive(oid);

    /* See if inflation or deflation were successful */
    if
    :: (objects[oid].owner == _pid && objects[oid].isInflated) ->
        break;

    :: (objects[oid].owner == txid && !objects[oid].isInflated) ->
        break;

    :: else;
    fi;
#endif /* BLOCKING */
    :: else -> /* Not inflated */
        resolveConflict(enemyid);
        waitOrInflate(enemyid, oid);
        validateTransaction();

    /* Now try to acquire the object */
    acquireExclusive(oid, enemyid);

    /* Check if I successfully acquired the object */
    if
    :: (objects[oid].owner == txid && !objects[oid].isInflated) ->
        /* Cleanup if necessary */
        if
        :: (TXABORTED(enemyid) && IS_OLD(oid)) ->
            restoreBackupData(oid);

```

```

        :: else;
        fi;

        /* Take a backup copy of the data */
        cloneData(oid);

        /* Object acquired (return TRUE) */
        break;

        :: else; /* Failed to acquire object... */
        fi;
    }
}

od;
fi;
} /* openWriteExclusive */

/**
 * Opens the object exclusively for reading. Goes to Label aborted if it fails.
 *
 * Doesn't backup any of the old data
 *
 * @param oid The 'pointer' to the object I want to open
 */
inline openReadExclusive(oid)
{
    if
    :: (objects[oid].owner == txid && !objects[oid].isInflated) ->
        validateTransaction();
        isInflated = false;

    :: else ->
        do
        :: TXABORTING(txid) -> goto aborted;

        :: else ->
            d_step {
                enemyid = objects[oid].owner;
                isInflated = objects[oid].isInflated;
            }

            if
#ifndef BLOCKING /* nonblocking */
                :: isInflated ->
                    inflatedOpenExclusive(oid);

                /* See if inflation or deflation were successful */
                if
                :: (objects[oid].owner == _pid && objects[oid].isInflated) ->
                    break;

                :: (objects[oid].owner == txid && !objects[oid].isInflated) ->
                    break;

                :: else;
                fi;
#endif /* BLOCKING */
                :: else -> /* Not inflated */
                    resolveConflict(enemyid);
                    waitOrInflate(enemyid, oid);
                    validateTransaction();

                    /* Now try to acquire the object */
                    acquireExclusive(oid, enemyid);

                    /* Check if I acquired the object */
                    if
                    :: (objects[oid].owner == txid && !objects[oid].isInflated) ->
                        /* Cleanup if necessary */
                        if
                        :: (TXABORTED(enemyid) && IS_OLD(oid)) ->
                            restoreBackupData(oid);

```

```

                :: else;
                fi;

                /* Object acquired (return TRUE) */
                break;

                :: else; /* Failed to acquire object... */
                fi;
            fi;
        od;
    fi;
} /* openReadExclusive */

/**
 * Opens the object exclusively for writing. Goes to Label aborted if it fails.
 *
 * Checks for visible reads.
 *
 * @param oid The 'pointer' to the object I want to open
 */
inline openWriteVisible(oid)
{
    if
    /* See if I already have the object open for writing */
    :: (objects[oid].owner == txid && !objects[oid].isInflated) ->
        validateTransaction();
        isInflated = false;

    :: else ->
        do
            :: TXABORTING(txid) -> goto aborted; /* been asked to abort */

        :: else ->
            d_step {
                enemyid = objects[oid].owner;
                isInflated = objects[oid].isInflated;
            }

        if
#ifdef BLOCKING /* nonblocking */
        :: isInflated ->
            inflatedOpenExclusive(oid);

            /* See if inflation or deflation were successful */
            if
            :: (objects[oid].owner == _pid && objects[oid].isInflated) ->
                break;

            :: (objects[oid].owner == txid && !objects[oid].isInflated) ->
                break;

            :: else;
            fi;
#endif /* BLOCKING */
        :: else -> /* Not inflated */
            resolveConflict(enemyid);
            waitOrInflate(enemyid, oid);
            validateTransaction();

            /* Now try to acquire the object */
            acquireExclusive(oid, enemyid);

            /* Check if I successfully acquired the object */
            if
            :: (objects[oid].owner == txid && !objects[oid].isInflated) ->

                /* Cleanup if necessary (safe before checking reads) */
                if
                :: (TXABORTED(enemyid) && IS_OLD(oid)) ->
                    restoreBackupData(oid);

                :: else;

```

```

        fi;

        resolveReadersConflicts(oid);
        waitOrInflateReaders(oid);
        validateTransaction();

        /* Take a backup copy of the data */
        cloneData(oid);

        /* Object acquired */
        break;

        :: else; /* Failed to acquire object, try again... */
        fi;
    fi;
od;
fi;
} /* openWriteVisible */

/**
 * Opens the object for read sharing (visibly). Goes to Label aborted if it fails.
 *
 * Uses visible reads with predefined slots per read.
 *
 * @param oid The 'pointer' to the object I want to open
 */
inline openReadVisible(oid)
{
    if
    /* Check if the object is already open for write or read */
    :: ((objects[oid].owner == txid || objects[oid].readers[_pid] == txid) &&
        !objects[oid].isInflated) ->
        validateTransaction();
        isInflated = false;

    :: else ->
        /*
         * Add this transaction to the readers list.
         * Must be done before checking writers to ensure that readers
         * are visible
         */
        objects[oid].readers[_pid] = txid;

    do
    :: TXABORTING(txid) -> goto aborted; /* been asked to abort */

    :: else ->
        d_step {
            enemyid = objects[oid].owner;
            isInflated = objects[oid].isInflated;
        }

    if
#ifdef BLOCKING /* nonblocking */
    :: isInflated ->
        inflatedOpenExclusive(oid);

        /* See if inflation or deflation were successful */
        if
        :: (objects[oid].owner == _pid && objects[oid].isInflated) ->
            break;

        :: (objects[oid].owner == txid && !objects[oid].isInflated) ->
            break;

        :: else;
        fi;
#endif
    :: else -> /* Not inflated */
        resolveConflict(enemyid);
        waitOrInflate(enemyid, oid);
        validateTransaction();
}

```

```

        /* Cleanup if necessary */

        if
        :: enemyid == INVALID; /* Inflation failed, try the loop again */

        /* If I need to cleanup, acquire, clean then release to do so */
        :: (enemyid != INVALID && TXABORTED(enemyid) && IS_OLD(oid)) ->
            casTransaction(oid, enemyid);

            if
            :: (objects[oid].owner == txid && !objects[oid].isInflated) ->
                restoreBackupData(oid);
                releaseObject(oid);
                break; /* Object acquired for read */

            :: else; /* Failed to acquire, redo loop */
            fi;

        :: else -> break; /* Object acquired for read */
        fi;

    fi;
od;
fi;
} /* openReadVisible */

```

```

/**
 * Finds the current data value of the object and returns it in value.
 *
 * Assumes the object is open for read (or write)! Doesn't validate that.
 */
inline getValue(oid, value)
{
#ifdef BLOCKING
    value = objects[oid].data;
#else /* nonblocking */
    if
    :: isInflated ->
        findLocator(oid, lid);

        if
        :: (lid == _pid); /* Ensure that it's my locator */
        :: else ->
            assert(TXABORTING(txid));
            goto aborted;
        fi;

        value = LOCATOR(oid, _pid).new;

    :: else ->
        value = objects[oid].data;
    fi;

    validateTransaction();
#endif /* BLOCKING */
}

```

```

/**
 * Sets the current data value of the object to value.
 *
 * Assumes the object is open for write! Doesn't validate that.
 */
inline setValue(oid, value)
{
#ifdef BLOCKING
    objects[oid].data = value;
#else /* nonblocking */
    if
    :: isInflated ->
        findLocator(oid, lid);

```

```

    if
    :: (lid == _pid); /* Ensure that it's my locator */
    :: else ->
        assert(TXABORTING(txid));
        goto aborted;
    fi;

    LOCATOR(oid, _pid).new = value;

    :: else ->
        objects[oid].data = value;
    fi;
#endif /* BLOCKING */
}

/*
=====
* Sanity Checks and Assertions
=====
*/

/**
* Performs sanity checks related to the object being open for write.
*
* Run before the data is modified.
*/
inline assertOpenWrite(oid)
{
#ifdef BLOCKING
    d_step {
        assert(TXACTIVE(txid) || TXABORTING(txid));
        assert(objects[oid].owner == txid);
        assert(objects[oid].isInflated == false);
        assert(objects[oid].old.id == txid);
        assert(objects[oid].data == objects[oid].old.backup);
    }
#else /* nonblocking */
    d_step {
        assert(TXACTIVE(txid) || TXABORTING(txid));

        if
        /* Not inflated and I know it */
        :: (!isInflated && !objects[oid].isInflated) ->
            assert(objects[oid].owner == txid);
            assert(objects[oid].old.id == txid);
            assert(objects[oid].data == objects[oid].old.backup);

        /* I think it's not inflated but it is inflated => I was aborted */
        :: (!isInflated && objects[oid].isInflated) ->
            assert(TXABORTING(txid));
            assert(objects[oid].old.id == txid);
            assert(objects[oid].data == objects[oid].old.backup);

        /* I think it's inflated but it's not => I was aborted */
        :: (isInflated && !objects[oid].isInflated) ->
            assert(TXABORTING(txid));
            assert(LOCATOR(oid, _pid).owner == txid);
            assert(LOCATOR(oid, _pid).old == LOCATOR(oid, _pid).new);

        /* I think it's inflated and it is */
        :: (isInflated && objects[oid].isInflated) ->
            assert(LOCATOR(oid, _pid).owner == txid);
            assert(LOCATOR(oid, _pid).old == LOCATOR(oid, _pid).new);
        fi;
    }
#endif /* BLOCKING */
}

/**
* Performs sanity checks related to the object being open for read.
*/

```

```

inline assertOpenRead(oid)
{
#ifdef READ_VISIBLE
#ifdef BLOCKING
    /* visible blocking */
    d_step {
        assert(TXACTIVE(txid) || TXABORTING(txid));

        if
        /* Actually open for write */
        :: (objects[oid].owner == txid) ->
            assert(objects[oid].isInflated == false);
            assert(objects[oid].old.id == txid);
            assert(objects[oid].data == objects[oid].old.backup);

        :: else ->
            assert(objects[oid].readers[_pid] == txid);
        fi;
    }
#else /* nonblocking */
    /* visible nonblocking */
    d_step {
        assert(TXACTIVE(txid) || TXABORTING(txid));

        if
        /* Not inflated and I know it */
        :: (!isInflated && !objects[oid].isInflated) ->
            if
            /* Actually open for write */
            :: (objects[oid].owner == txid) ->
                assert(objects[oid].old.id == txid);
                assert(objects[oid].data == objects[oid].old.backup);

            :: else ->
                assert(objects[oid].readers[_pid] == txid);
            fi;

        /* I think it's not inflated but it is inflated */
        :: (!isInflated && objects[oid].isInflated) ->
            skip; /* Many different things could have happened */

        /* I think it's inflated but it's not => I was aborted */
        :: (isInflated && !objects[oid].isInflated) ->
            assert(TXABORTING(txid));
            assert(LOCATOR(oid, _pid).owner == txid);
            assert(LOCATOR(oid, _pid).old == LOCATOR(oid, _pid).new);

        /* I think it's inflated and it is */
        :: (isInflated && objects[oid].isInflated) ->
            assert(LOCATOR(oid, _pid).owner == txid);
            assert(LOCATOR(oid, _pid).old == LOCATOR(oid, _pid).new);
        fi;
    }
#endif /* BLOCKING */
#else /* exclusive reads */
#ifdef BLOCKING
    /* exclusive blocking */
    d_step {
        assert(TXACTIVE(txid) || TXABORTING(txid));
        assert(objects[oid].owner == txid);
        assert(objects[oid].isInflated == false);
    }
#else /* nonblocking */
    /* exclusive nonblocking */
    d_step {
        assert(TXACTIVE(txid) || TXABORTING(txid));

        if
        /* Not inflated and I know it */
        :: (!isInflated && !objects[oid].isInflated) ->
            assert(objects[oid].owner == txid);

```

```

    /* I think it's not inflated but it is inflated => I was aborted */
    :: (!isInflated && objects[oid].isInflated) ->
        assert(TXABORTING(txid));

    /* I think it's inflated but it's not => I was aborted */
    :: (isInflated && !objects[oid].isInflated) ->
        assert(TXABORTING(txid));
        assert(LOCATOR(oid, _pid).owner == txid);
        assert(LOCATOR(oid, _pid).old == LOCATOR(oid, _pid).new);

    /* I think it's inflated and it is */
    :: (isInflated && objects[oid].isInflated) ->
        assert(LOCATOR(oid, _pid).owner == txid);
        assert(LOCATOR(oid, _pid).old == LOCATOR(oid, _pid).new);
    fi;
}
#endif /* BLOCKING */

#endif /* READ_VISIBLE */
}

/*
 * =====
 * Preallocating Memory
 * =====
 */

/**
 * Preallocated transactions.
 *
 * (remember that txn #0 is an unusable committed txn)
 */
Transaction transactions[TOTAL_TXN];

/**
 * Preallocated objects for NZSTM
 */
NZObject objects[TOTAL_OBJECTS];

/**
 * Expected Values
 *
 * Used for some tests to store what the expected values of objects are
 */
byte expected[TOTAL_OBJECTS] = INITV;

/*
 * =====
 * Tests
 * =====
 */

/*
 * I the C code I would be using functions/macros similar to getValue/setValue
 * as defined above. However, to keep the state space as small as possible I
 * have to resort to the ugly code below.
 *
 * An example of proper getValue and setValue usage can be found below though
 * in addAround
 */

/**
 * Increments and decrements the same object while performing sanity checks.
 */
inline incDecObject(oid)
{
    OPEN_WRITE(oid);
    assertOpenWrite(oid);
}

#ifdef BLOCKING
    objects[oid].data++;

```

```

    assert(objects[oid].data == 1 + INITV);

    objects[oid].data--;
#else /* nonblocking */
    if
    :: isInflated ->
        findLocator(oid, lid);

        if
        :: (lid == _pid); /* Ensure that it's my locator */
        :: else ->
            assert(TXABORTING(txid));
            goto aborted;
        fi;

        LOCATOR(oid, _pid).new++;
        assert(LOCATOR(oid, _pid).new == 1 + INITV);
        LOCATOR(oid, _pid).new--;

    :: else ->
        objects[oid].data++;
        assert(objects[oid].data == 1 + INITV);
        objects[oid].data--;
    fi;
#endif /* BLOCKING */
}

```

```

/**
 * Reads the data of an object that is expected to be 0.
 */

```

```

inline readZeroObject(oid)
{
    /*
     * assert(objects[oid].data == INITV);
     */

    OPEN_READ(oid);
    assertOpenRead(oid);

```

```

#ifdef BLOCKING
    assert(objects[oid].data == INITV);
#else /* nonblocking */
    if
    :: isInflated ->
        findLocator(oid, lid);

        if
        :: (lid == _pid); /* Ensure that it's my locator */
        :: else ->
            atomic {
                assert(TXABORTING(txid));
                goto aborted;
            }
        fi;

        assert(LOCATOR(oid, _pid).new == INITV);

    :: else ->
        assert(objects[oid].data == INITV);
    fi;
#endif /* BLOCKING */
}

```

```

/**
 * Takes from o1 and deposits into o2 the amount in val
 */

```

```

inline transferAmount(o1, o2, val)
{
    /*
     * objects[o1].data -= val;
     */

```

```

    * objects[o2].data += val;
    * assert(objects[o1].data + objects[o2].data == 2 * INITV);
    */

#ifdef BLOCKING
    OPEN_WRITE(o1);
    OPEN_WRITE(o2);

    objects[o1].data = objects[o1].data - val;
    objects[o2].data = objects[o2].data + val;

    assert(objects[o1].data + objects[o2].data == 2 * INITV);
#else /* nonblocking */
    dummy = 0;

    OPEN_WRITE(o1);
    assertOpenWrite(o1);

    if
    :: isInflated ->
        findLocator(o1, lid);

        if
        :: (lid == _pid); /* Ensure that it's my locator */
        :: else ->
            assert(TXABORTING(txid));
            goto aborted;
        fi;

        LOCATOR(o1, _pid).new = LOCATOR(o1, _pid).new - val;
        dummy = dummy + LOCATOR(o1, _pid).new;

    :: else ->
        objects[o1].data = objects[o1].data - val;
        dummy = dummy + objects[o1].data;
    fi;

    OPEN_WRITE(o2);
    assertOpenWrite(o2);

    if
    :: isInflated ->
        findLocator(o2, lid);

        if
        :: (lid == _pid); /* Ensure that it's my locator */
        :: else ->
            assert(TXABORTING(txid));
            goto aborted;
        fi;

        LOCATOR(o2, _pid).new = LOCATOR(o2, _pid).new + val;
        dummy = dummy + LOCATOR(o2, _pid).new;

    :: else ->
        objects[o2].data = objects[o2].data + val;
        dummy = dummy + objects[o2].data;
    fi;

    assert(dummy == 2 * INITV);
#endif /* BLOCKING */
}

/**
 * Checks that o1 and o2 balance, the sum of their values is twice of their
 * initial value
 */
inline checkBalance(o1, o2)
{
    /*
     * assert(objects[o1].data + objects[o2].data == 2 * INITV);
     */
}

```

```

#ifdef BLOCKING
    OPEN_READ(o1);
    OPEN_READ(o2);
    assert(objects[o1].data + objects[o2].data == 2 * INITV);
#else /* nonblocking */
    dummy = 0;

    OPEN_READ(o1);
    assertOpenRead(o1);

    if
    :: isInflated ->
        findLocator(o1, lid);

        if
        :: (lid == _pid); /* Ensure that it's my locator */
        :: else ->
            assert(TXABORTING(txid));
            goto aborted;
        fi;

        dummy = dummy + LOCATOR(o1, _pid).new;

    :: else ->
        dummy = dummy + objects[o1].data;
    fi;

    OPEN_READ(o2);
    assertOpenRead(o2);

    if
    :: isInflated ->
        findLocator(o2, lid);

        if
        :: (lid == _pid); /* Ensure that it's my locator */
        :: else ->
            assert(TXABORTING(txid));
            goto aborted;
        fi;

        dummy = dummy + LOCATOR(o2, _pid).new;

    :: else ->
        dummy = dummy + objects[o2].data;
    fi;

    assert(dummy == 2 * INITV);
#endif /* BLOCKING */
}

/**
 * Performs the add around test which reads the first value, then the second
 * value, adding their sum to the third. Then adding the third back to the
 * first.
 */
inline addAround(o1, o2, o3)
{
    /*
     * objects[o3].data += objects[o1].data + objects[o2].data;
     * objects[o1].data += objects[o3].data;
     * Asserting all the while that the variables match what's expected, OR
     * that I have been aborted.
     */

    byte a, b, c;

    OPEN_READ(o1);
    getValue(o1, a);
    assert(a == expected[o1] || TXABORTING(txid));

```

```

OPEN_READ(o2);
getValue(o2, b);
assert(b == expected[o2] || TXABORTING(txid));

OPEN_WRITE(o3);
getValue(o3, c);
assert(c == expected[o3] || TXABORTING(txid));
c = a + b + c;
setValue(o3, c);

OPEN_WRITE(o1);
a = a + c;
setValue(o1, a);
}

```

```

/*
 * =====
 * Running Processes
 * =====
 */

```

```

#ifdef INCDEC_TEST

```

```

/* *
 * The main body of the program representing the increment/decrement test.
 */

```

```

active [PROCESSES] proctype IncDecTest()
{

```

```

    TXN_DEFS();

```

```

    beginTransaction();

```

```

    /* Do a write test and a read test nondeterministically */

```

```

if
  :: true ->
    incDecObject(0);

```

```

  :: true ->
    readZeroObject(1);
fi;

```

```

if
  :: true ->
    readZeroObject(0);

```

```

  :: true ->
    incDecObject(1);
fi;

```

```

/* All transaction must either commit or abort */
progress:

```

```

aborted:
    commitTransaction();
}

```

```

#endif /* INCDEC_TEST */

```

```

#ifdef BALANCE_TEST

```

```

/* *
 * The main body of the program representing the balance test.
 */

```

```

/*
 * NOTE:
 * - Comment out processes (from the end) to reduce the number of running ones
 * - Ensure that PROCESSES is greater or equal to the number of processes spawned
 */

```

```

active proctype BalanceTest0()

```

```

{
    TXN_DEFS();

```

```

    beginTransaction();

```

```

    transferAmount(0, 1, 2);

progress0:
aborted:
    commitTransaction ();
}

active proctype BalanceTest1 ()
{
    TXN_DEFS ();

    beginTransaction ();

    checkBalance(0, 1);

progress1:
aborted:
    commitTransaction ();
}

active proctype BalanceTest2 ()
{
    TXN_DEFS ();

    beginTransaction ();

    transferAmount(1, 0, 3);

progress2:
aborted:
    commitTransaction ();
}

active proctype BalanceTest3 ()
{
    TXN_DEFS ();

    beginTransaction ();

    checkBalance(1, 0);

progress3:
aborted:
    commitTransaction ();
}

#endif /* BALANCE_TEST */

#ifdef ADD_AROUND_TEST

/**
 * A process that attempts to perform the following transaction, while updating
 * the expected state for future processes.
 *
 * Read x
 * Read y
 * z += x + y
 * x += z
 */
proctype addrProc(byte x, y, z)
{
    TXN_DEFS ();

    beginTransaction ();

    addAround(x, y, z);

progress:
aborted:
    /* If I commit, atomically change the system's expected state */
    atomic {
        commitTransaction ();
    }
}

```

```

        if
        :: TXCOMMITTED( txid ) ->
            expected[z] = expected[x] + expected[y] + expected[z];
            expected[x] = expected[x] + expected[z];
        :: else;
        fi;
    }
}

/**
 * Set the initial values of the objects and start the processes
 */
init {
    d_step {
        objects[0].data = 3;
        expected[0] = objects[0].data;

        objects[1].data = 5;
        expected[1] = objects[1].data;

        objects[2].data = 7;
        expected[2] = objects[2].data;
    }

    /*
    * NOTE:
    * - Run as many processes you'd like to check
    * - Ensure that PROCESSES is greater or equal to the number of processes spawned
    */
    atomic {
        run addrProc(0, 1, 2);
        run addrProc(1, 2, 0);
        run addrProc(2, 0, 1);
//        run addrProc(2, 1, 0);
    }
}

#endif /* ADD_AROUND_TEST */

```

## References

- [1] C. S. Ananian. *Architectural and compiler support for strongly atomic transactional memory*. PhD thesis, Massachusetts Institute of Technology, 2007.
- [2] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for supporting dynamic-sized data structures. In *PODC*, pages 92–101, 2003.
- [3] G. J. Holzmann. *The SPIN Model Checker*. Addison-Wesley Professional, 2003.
- [4] J. R. Larus and R. Rajwar. *Transactional Memory*. Synthesis Lectures on Computer Architecture. Morgan and Claypool Publishers, 2007.
- [5] Y. Lev and M. Moir. Fast read sharing mechanism for software transactional memory, 2004.  
<http://research.sun.com/scalable/pubs/PODC04-Poster.pdf>.
- [6] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. Scherer III, and M. L. Scott. Lowering the overhead of nonblocking software transactional memory, 2006.
- [7] F. Tabbà, M. Moir, J. R. Goodman, A. W. Hay, and C. Wang. NZTM: Nonblocking zero-indirection transactional memory. *SPAA '09: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, 2009.