A Compositional Method for Verifying Software Transactional Memory Implementations

April 10, 2008

Technical Report
MSR-TR-2008-56

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

This page intentionally left blank.

A Compositional Method for Verifying Software Transactional Memory Implementations

Serdar Tasiran
Koç University and Microsoft Research
stasiran@ku.edu.tr

Abstract

We present a compositional method for verifying software transactional memory (STM) implementations and its application to the Bartok STM. The method consists of two parts. The first is a generic, manual proof of serializability at the algorithm level for lazy-invalidate, write-in-place STM's. The proof relies on three properties of program executions that the STM must ensure. The second part consists of proving that the Bartok STM implementation guarantees these properties and thus refines the algorithm-level description. We present a novel technique for expressing the properties required of the STM implementation as assertions in sequential programs that model certain interference scenarios. This is a key benefit, as it allows these properties to be checked using sequential program verification tools. Using our method, the Spec# language and the Boogie verification tool, we were able to detect an omission in the published pseudo-code for the STM implementation and "challenge bugs" extracted from earlier versions of the STM. We were also able to prove correct the most recent version of the implementation.

1. Introduction

As multi-processor computing platforms become more commonplace, the need for higher-level, compositional programming primitives for writing concurrent programs is becoming more pronounced. In this context, transactional memory (TM) has received much recent interest[7]. A TM allows the programmer to designate a code block as a transaction and ensures that the executions of transactions appear atomic and serialized to the programmer while, in reality, they are executed highly concurrently.

As they find more widespread use, TM's are likely to become integral parts of execution platforms. The semantic specification of TM then becomes part of the programming model. Therefore, verifying whether a TM implementation satisfies its semantic specification is an important task. This paper presents a technique for carrying out this task for software transactional memory (STM) implementations.

STM implementations must provide high performance. This forces them to maximize concurrent processing using intricate mechanisms for the detection and management of conflicts between transactions. Industrial implementations of STM's may consist of several thousands of lines of highly-concurrent code. A program may have an arbitrary number of concurrent threads, nested transactions, and each transaction may consist of an arbitrarily long sequence of program instructions. These make the proof of correctness for a TM implementation a complex task.

In order to put our work in context, we view STM descriptions at two levels: algorithm-level descriptions and actual implementations. Algorithm-level descriptions fall into one of a few categories, e.g. eager vs. lazy invalidate STM's, and buffered vs. write-in-place STM's. Previous work on verifying STM's has focused on verifying that these algorithm- or semantic-level descriptions ensure atomicity and serializability. Our work has this component as well, but places the emphasis on verifying that actual implementations (say in C or C#) are correct refinements of the algorithm-level descriptions. We focus on this task because we believe this part of STM development is more error-prone and the verification effort has to be carried out separately for each STM implementation.

STM implementations are structured in a way [4] that reflects their correctness argument – a fact we exploit when verifying a lazy-invalidate, write-in-place STM. We separate the proof of serializability of the algorithm-level description, which is generic and performed manually, from the proof that the implementation is a correct refinement of the algorithm-level description. In the algorithm-level proof, we model a program composed with an abstract STM, and prove that executions produced are serializable if the abstract STM satisfies certain non-interference properties. The implementation-level proof consists of checking whether these properties are satisfied by the STM implementation, which implies correct refinement.

A novel contribution of our work is the statement of non-interference properties as assertions in simple sequential programs. These programs refer to a single object and a transaction, and mimic potential interference from other transactions. This provides two significant benefits. First, being able to use tools for sequential programs for verifying STM's is valuable since these tools are a lot more mature and powerful. Second, rather than arbitrarily long and many transactions acting on arbitrarily many objects, the sequential programs refer to one object and one transactional access in a (model of the) concurrent environment. This allowed us to use the Spec# language and the Boogie sequential program verifier to easily handle all proof tasks.

We applied our technique to the Bartok STM implementation [5], part of the Bartok runtime and the Singularity operating system developed at Microsoft Research. Using our technique, we were able to detect a bug in published pseudocode for the Bartok STM and challenge bugs from earlier versions of Bartok. Upon correcting the bugs, we were able to successfully complete the verification task using less than ten minutes of CPU time.

Section 2 presents our model for programs that use write-in place, lazy-invalidate transactions and the manual proof of correctness predicated on non-interference properties of the STM implementation. Section 4 presents the Bartok STM, our model for it, and the checking of the non-interference properties on the Bartok STM implementation.

2. Modeling Programs with Transactions

We model programs using software transactions using a language we call OTFJ. OTFJ is based on Transactional Featherweight Java (TFJ) [6] and is a simple language that has all the essential features of imperative object-oriented programs with multiple threads and transactions that are relevant to our work including nesting of transactions. Non-transactional accesses are modeled by single-

$$
\begin{array}{lcl}
\mathtt{P} & ::= & \mathtt{0} \mid \ \mathtt{P} \mid \mathtt{P} \ \mid \mathtt{t}[\mathtt{e}] \\
\mathtt{L} & ::= & \mathtt{class\ C}\ \{\overline{\mathtt{f}};\ \overline{\mathtt{M}}\} \\
\mathtt{M} & ::= & \mathtt{m}(\overline{\mathtt{x}})\{\mathtt{e};\} \\
\mathtt{s} & ::= & \mathtt{v} \mid\mid \mathtt{s.f} \mid \mathtt{s.m}(\overline{\mathtt{s}}) \mid \mathtt{s.f} := \mathtt{s} \mid \mathtt{new\ C} \mid \\
 & & \mathtt{lbl : onacid;\ s;\ commit} \mid \mathtt{null} \mid \mathtt{s;s} \\
\mathtt{e} & ::= & \mathtt{s} \mid \mathtt{s;s} \mid \mathtt{spawn\ s} \\
\mathtt{v} & ::= & \mathtt{r} \mid \mathtt{v.f} \mid \mathtt{v.m}(\overline{\mathtt{v}}) \mid \mathtt{v.f} := \mathtt{v}
\end{array}
$$

Figure 1. OTFJ Syntax

expression transactions in order to provide strong atomicity [7]. The syntax of OTFJ is given in Fig.1.

OTFJ differs from TFJ in that it also allows executions in which transactions conflict, and allows transactions to abort and roll back explicitly. OTFJ syntactically scopes transactions between a pair of `onacid` and `commit` statements and disallows thread creation within transactions.

`P` represents a process term, `s` an expression that is free of thread `spawn`'s and `e` an expression that allows spawning of threads. Threads spawned consist of a sequence of transactions. The symbol "`;`" is used to represent sequential composition. TFJ has no primitive types – the only values in TFJ are references and all references except `null` are created by the `new C()` construct. A process term can be the empty process `0`, or it is the parallel composition of two processes, or a thread `t` running an expression `e`, represented by `t[e]`.

In OTFJ, transactions are allowed to explicitly abort and roll back. They are then automatically re-tried until they succeed. In TFJ, conflicting transactions are simply not allowed to commit – the thread executing the transaction terminates and this execution is not considered part of the program's behavior. OTFJ more closely models the behavior of a realistic STM implementation where conflicts are experienced and need to be recovered from.

Semantics preliminaries: Each thread has a distinct thread label $\mathtt{t} \in \mathit{ThLbl}$. The execution an `onacid` action by `t` starts a new *transaction* whose context ends with the execution of a `commit`. Each transaction is given a unique *name* $\mathtt{l} \in \mathit{TrNames}$. Each transaction has a unique label $\overline{\mathtt{l}} \in \mathit{TransLbl}$, and, if within a transaction with label $\overline{\mathtt{l}}$ a new transaction with name `l` is started, the new transaction has label $\overline{\mathtt{l}}.\mathtt{l} = \mathtt{l}_0\mathtt{l}_1...\mathtt{l}_k\mathtt{l}$. $\overline{\mathtt{l}} \lhd \overline{\mathtt{l}}'$ denotes that the label $\overline{\mathtt{l}}$ is a prefix of $\overline{\mathtt{l}}$

The semantics of an OTFJ program `P` is given by the set of executions and traces that `P` can generate, denoted by $\mathit{Execs}(\mathtt{P})$ and $\mathit{Traces}(\mathtt{P})$, respectively. Executions are sequences of actions, and traces are sequences of triples of the form $\mathtt{P}_i\ \Gamma_i\ \sigma_i$ where $\mathtt{P}_i$ denotes a program, Γ_i denotes a program state and σ_i denotes an STM state.

Actions: An action is a triple of the form $\alpha = (\mathtt{t}, \overline{\mathtt{l}}, \kappa)$ where `t` is the thread performing the action, α is perfomed in the context a transaction labelled by $\overline{\mathtt{l}}$ and κ is the *kind* of the action. We define $\mathit{ownerTx}(\alpha) = \overline{\mathtt{l}}$ to be the (label of) the transaction as part of which α is executed.

The semantics of actions (Fig.s 2-4) model a sequentially-consistent interleaving of actions by OTFJ threads along with an associated STM implementation that determines which transactions are allowed to commit and how actions are undone when a transaction is being rolled back. Local actions (Fig.2) consist of field read and assignment (write) accesses, method invocation and object creation. OTFJ has call-by-value semantics: Before a method is invoked, all parameters are evaluated. Global actions (Fig.3) include local actions (PLAIN), spawning a new thread, starting, committing and rolling back transactions, and the completion of a thread that has no more computation to perform. The semantics of some action kinds refer to predicates (Fig 4) which describe how

READ FIELD

$$
\frac{\begin{array}{c}\tau = \mathit{O4R}(\mathtt{C}(\overline{\mathtt{u}})) \\ \kappa = \mathit{rd}(\mathtt{r}, \mathtt{u}_i) \qquad \langle \mathcal{S}', \mathcal{E}', \mathtt{C}(\overline{\mathtt{u}})\rangle = \mathit{read}(\mathcal{S}, \mathtt{r}, \mathcal{E}) \\ \sigma' = \mathit{Opn4Rd}(\sigma, \mathtt{C}(\overline{\mathtt{u}})) \qquad \mathit{fields}(\mathtt{C}) = \overline{\mathtt{f}}\end{array}}{\mathcal{S}, \sigma, \mathcal{E}\ \mathtt{r.f}_i \xrightarrow{\tau} \mathcal{S}, \sigma', \mathcal{E}\ \mathtt{r.f}_i \xrightarrow{\kappa} \mathcal{S}', \sigma', \mathcal{E}'\ \mathtt{u}_i}
$$

SUCCESSFUL FIELD ASSIGNMENT

$$
\frac{\begin{array}{c}\tau = \mathit{O4U}(\mathtt{C}(\overline{\mathtt{r}})) \\ \kappa = \mathit{wr}(\mathtt{v.f}_i, \mathtt{r}, \mathtt{r}') \qquad \langle \mathcal{S}', \mathcal{E}', \mathtt{C}(\overline{\mathtt{r}})\rangle = \mathit{read}(\mathcal{S}, \mathtt{v}, \mathcal{E}) \\ \sigma' = \mathit{Opn4Wrt}(\sigma, \mathtt{C}(\overline{\mathtt{r}})) \qquad \mathit{OK2Wrt}(\sigma, \mathtt{C}(\overline{\mathtt{r}}) \\ \langle \mathcal{E}'', \mathcal{S}''\rangle = \mathit{write}(\mathtt{v} \mapsto \mathtt{C}(\overline{\mathtt{r}})\downarrow_i^{\mathtt{r}'}, \mathcal{E}', \mathcal{S}')\end{array}}{\begin{array}{c}\mathcal{S}, \sigma, \mathcal{E}\ \mathtt{v.f}_i := \mathtt{r}' \xrightarrow{\tau} \mathcal{S}, \sigma', \mathcal{E}\ \mathtt{v.f}_i := \mathtt{r}' \\ \xrightarrow{\kappa} \mathcal{S}'', \sigma', \mathcal{E}''\mathtt{r}'\end{array}}
$$

FAILED FIELD ASSIGNMENT

$$
\frac{\begin{array}{c}\tau = \mathit{O4U}(\mathtt{C}(\overline{\mathtt{r}})) \qquad \langle \mathcal{S}', \mathcal{E}', \mathtt{C}(\overline{\mathtt{r}})\rangle = \mathit{read}(\mathcal{S}, \mathtt{v}, \mathcal{E}) \\ \sigma' = \mathit{Opn4Wrt}(\sigma, \mathtt{C}(\overline{\mathtt{r}})) \qquad \neg \mathit{OK2Wrt}(\sigma, \mathtt{C}(\overline{\mathtt{r}})\end{array}}{\mathcal{S}, \sigma, \mathcal{E}\ \mathtt{v.f}_i := \mathtt{r}' \xrightarrow{\tau} \mathcal{S}, \sigma', \mathcal{E}\ \mathtt{r}'}
$$

METHOD INVOCATION

$$
\frac{\begin{array}{c}\tau = \mathit{O4R}(\mathtt{C}(\overline{\mathtt{r}})) \\ \kappa = \mathit{rd}(\mathtt{v}, \mathtt{C}(\overline{\mathtt{r}})) \qquad \langle \mathcal{S}, \mathcal{E}', \mathtt{C}(\overline{\mathtt{r}})\rangle = \mathit{read}(\mathcal{S}, \mathtt{r}, \mathcal{E}) \\ \sigma' = \mathit{Opn4Rd}(\sigma, \mathtt{C}(\overline{\mathtt{r}})) \qquad \mathit{mbody}(\mathtt{m}, \mathtt{C}) = (\overline{\mathtt{x}}, \mathtt{e})\end{array}}{\begin{array}{c}\mathcal{S}, \sigma, \mathcal{E}\ \mathtt{r.m}(\overline{\mathtt{r}}) \xrightarrow{\tau} \mathcal{S}, \sigma', \mathcal{E}\ \mathtt{r.m}(\overline{\mathtt{r}}) \\ \xrightarrow{\kappa} \mathcal{S}', \sigma', \mathcal{E}'[\overline{\mathtt{r}}/\overline{\mathtt{x}}, \mathtt{r}/\mathtt{this}]\mathtt{e}\end{array}}
$$

OBJECT CREATION

$$
\frac{\begin{array}{c}\kappa = \mathit{xt}(\mathtt{r}, \mathtt{C}(\overline{\mathtt{null}})) \\ \mathtt{r\ fresh} \qquad (\mathcal{S}', \mathcal{E}') = \mathit{extend}(\mathcal{S}, \mathtt{r} \mapsto \mathtt{C}(\overline{\mathtt{null}}), \mathcal{E})\end{array}}{\mathcal{S}, \sigma, \mathcal{E}\ \mathtt{new\ C}() \xrightarrow{\kappa} \mathcal{S}', \sigma', \mathcal{E}'\mathtt{r}}
$$

Figure 2. Semantics for local actions.

the program and STM state is updated in response to these actions. The STM implementation state and predicates that refer to it are left unspecified. $\mathit{inTrans}(\mathtt{t}, \Gamma, \overline{\mathtt{l}})$ is a predicate that evaluates to true iff at program state Γ, thread `t` is executing a transaction with label $\overline{\mathtt{l}}$.

Program and STM states: A program state Γ is described as a tuple $\Gamma = (\mathcal{S}, \overline{(\mathtt{t}, \mathcal{E})})$ where $\mathcal{S}$ is the central, shared store and $\overline{(\mathtt{t}, \mathcal{E})}$ is a sequence of pairs of threads `t` and corresponding thread environments $\mathcal{E}$. The shared store is a partial map that specifies the object bindings for all references, i.e., for each reference `r`, $\mathtt{r} \mapsto \mathtt{C}(\overline{\mathtt{u}})$ or $\mathtt{r} \mapsto \mathtt{null}$. We write $\mathcal{S}(\mathtt{r}) = \mathtt{C}(\overline{\mathtt{u}})$ if in the current store, the reference `r` refers to an object $\mathtt{C}(\overline{\mathtt{u}})$. $\mathcal{S}[\mathtt{r} \mapsto \mathtt{C}(\overline{\mathtt{u}})]$ represents a store that is the same as $\mathcal{S}$ except the reference `r` now refers to an object $\mathtt{C}(\overline{\mathtt{u}})$. OTFJ is dynamically typed. A *thread environment* $\mathcal{E}$ is a sequence of transaction environments $\mathcal{E} = \langle(\mathtt{l}_0, \rho_0), (\mathtt{l}_1, \rho_1), ..., (\mathtt{l}_n, \rho_n)\rangle$ where thread `t` is in the process of executing the innermost transaction with label $\overline{\mathtt{l}} = \mathtt{l}_0\mathtt{l}_1...\mathtt{l}_n$. For each transaction with name $\mathtt{l}_i$, the *transaction log* ρ_i contains the sequence of read and write accesses performed by the transaction with name $\mathtt{l}_i$. In the rules in Figure 2, $\mathcal{E}\ \mathtt{e} \xrightarrow{\alpha} \mathcal{E}'; \mathtt{e}'$ represents the program taking the action α as a result of which the expression `e` evaluates to $\mathtt{e}'$ and the transaction environment changes from $\mathcal{E}$ to $\mathcal{E}'$.

Executions and traces: Formally, an execution ξ of an OTFJ program `P` is a sequence of reductions satisfying specifications given in Fig. 3. An execution ξ is represented by a partial map $\xi : \mathcal{N} \mapsto \mathit{ThLbl} \times \mathit{TransLbl} \times \mathit{Kind}$. $\xi(i)$ is the action $\alpha =$

PLAIN

$$\frac{\mathtt{P} = \mathtt{P}'' \mid \mathtt{t}[\mathtt{e}] \qquad \mathcal{S}, \sigma, \mathcal{E}\ \mathtt{e} \xrightarrow{\tau} \mathcal{S}, \sigma', \mathcal{E}\ \mathtt{e} \xrightarrow{\kappa} \mathcal{S}', \sigma', \mathcal{E}'\mathtt{e}' \qquad \mathtt{P}' = \mathtt{P}'' \mid \mathtt{t}[\mathtt{e}'] \qquad \Gamma = (\mathcal{S}, \overline{(\mathtt{t}, \mathcal{E})}.\mathtt{t}, \mathcal{E}) \qquad \Gamma' = (\mathcal{S}', \overline{(\mathtt{t}, \mathcal{E})}.\mathtt{t}, \mathcal{E}') \qquad inTrans(\mathtt{t}, \Gamma, \overline{\mathtt{l}})}{\mathtt{P}\ \Gamma\ \sigma \overset{(\mathtt{t},\overline{\mathtt{l}},\tau)}{\Longrightarrow} \mathtt{P}\ \Gamma\ \sigma' \overset{(\mathtt{t},\overline{\mathtt{l}},\kappa)}{\Longrightarrow} \mathtt{P}'\ \Gamma'\ \sigma'}$$

SPAWN

$$\frac{\mathtt{P} = \mathtt{P}'' \mid \mathtt{t}[E[\mathtt{spawn\ s}]] \qquad \mathtt{P}' = \mathtt{P}'' \mid \mathtt{t}[E[\mathtt{null}]] \mid \mathtt{t}'[\mathtt{s}] \qquad \mathtt{t}'\ \mathtt{fresh} \qquad \Gamma' = spawn(\mathtt{t}, \mathtt{t}', \Gamma)}{\mathtt{P}\ \Gamma\ \sigma \overset{(\mathtt{t},\mathtt{null},spawn(\mathtt{t}'))}{\Longrightarrow} \mathtt{P}'\ \Gamma'\ \sigma}$$

TRANSACTION START

$$\frac{\mathtt{P} = \mathtt{P}'' \mid \mathtt{t}[E[\mathtt{lbl : onacid; s}]] \qquad \mathtt{P}' = \mathtt{P}'' \mid \mathtt{t}[E[\mathtt{lbl : s}]] \qquad \mathtt{l}\ \mathtt{fresh} \qquad \sigma' = BgnTx(\mathtt{t}, \sigma) \qquad inTrans(\mathtt{t}, \overline{\mathtt{l}}, \Gamma) \qquad \Gamma' = start(\mathtt{l}, \mathtt{s}, \mathtt{t}, \Gamma)}{\mathtt{P}\ \Gamma\ \sigma \overset{(\mathtt{t},\overline{\mathtt{l}},newTr(\mathtt{t}))}{\Longrightarrow} \mathtt{P}'\ \Gamma'\ \sigma'}$$

TRANSACTION COMMIT

$$\frac{v\mathcal{E} = \mathcal{E}'.(\mathtt{l}, \mathtt{s}) : \rho \qquad inTrans(\mathtt{t}, \overline{\mathtt{l}}.\mathtt{l}, \Gamma) \qquad ValidTx(\overline{\mathtt{l}}) \qquad \mathtt{P} = \mathtt{P}'' \mid \mathtt{t}[E[\mathtt{lbl : r; commit;}]] \qquad \mathtt{P}' = \mathtt{P}'' \mid \mathtt{t}[E[\mathtt{r}]] \qquad OK2Cmt(\mathtt{t}, \sigma) \qquad \sigma' = CmtTx(\mathtt{t}, \sigma) \qquad \Gamma' = commit(\mathtt{t}, \mathcal{E}, \Gamma)}{\mathtt{P}\ \Gamma\ \sigma \overset{(\mathtt{t},\overline{\mathtt{l}}.\mathtt{l},cmt)}{\Longrightarrow} \mathtt{P}'\ \Gamma'\ \sigma'}$$

TRANSACTION ROLLBACK

$$\frac{\tau = undoLLE(\mathtt{l}) \qquad \Gamma = \Gamma''.\mathtt{t}, \mathcal{E} \qquad \mathcal{E} = \mathcal{E}'.(\mathtt{l}, \mathtt{s}) : \rho \qquad inTrans(\mathtt{t}, \overline{\mathtt{l}}.\mathtt{l}, \Gamma) \qquad \mathtt{P} = \mathtt{P}'' \mid \mathtt{t}\ [\ E\ [\ \mathtt{lbl : r; commit}\ ]\ ] \qquad \mathtt{P}' = \mathtt{P}'' \mid \mathtt{t}\ [\ E\ [\ \mathtt{lbl : s\ commit}\ ]\ ] \qquad \sigma_{i+1} = undoLast(\sigma_i, \mathtt{l}) \qquad \langle\Gamma_0, \Gamma_1, ..., \Gamma_k\rangle, \langle\alpha_0, \alpha_1, ..., \alpha_k\rangle = undo((\Gamma, \mathtt{t}, \rho, \mathtt{l}), \langle\rangle, \langle\rangle)}{\mathtt{P}\ \Gamma\ \sigma \overset{\tau}{\Longrightarrow} \mathtt{P}\ \Gamma\ \sigma_0 \overset{\alpha_0}{\Longrightarrow} \mathtt{P}\ \Gamma_0\ \sigma_0 \overset{\tau}{\Longrightarrow} \mathtt{P}\ \Gamma_0\ \sigma_1 \overset{\alpha_1}{\Longrightarrow} \mathtt{P}\ \Gamma_1\ \sigma_1 ... \overset{\tau}{\Longrightarrow} \mathtt{P}\ \Gamma_{k-1}\ \sigma_k \overset{\alpha_k}{\Longrightarrow} \mathtt{P}\ \Gamma_k\ \sigma_k \overset{(\mathtt{t},\overline{\mathtt{l}}.\mathtt{l},undo)}{\Longrightarrow} \mathtt{P}'\ \Gamma_k\ \sigma_k}$$

THREAD DEATH

$$\frac{\mathtt{P} = \mathtt{P}' \mid \mathtt{t}[\mathtt{r}] \qquad \Gamma = \mathtt{t}, \mathcal{E}.\Gamma'}{\mathtt{P}\ \Gamma\ \sigma \overset{(\mathtt{t},\mathtt{null},kill)}{\Longrightarrow} \mathtt{P}'\ \Gamma'\ \sigma'}$$

Figure 3. Semantics for global actions.

$(\mathtt{t}_i, \overline{\mathtt{l}}, \kappa_i)$. We informally write an execution along with the trace it corresponds to as follows $\xi = \mathtt{P}_0\ \Gamma_0\ \sigma_0 \overset{\alpha_0}{\Longrightarrow} \mathtt{P}_1\ \Gamma_1\ \sigma_1 \overset{\alpha_1}{\Longrightarrow} ...\mathtt{P}_{n-1}\ \Gamma_{n-1}\ \sigma_{n-1} \overset{\alpha_{n-1}}{\Longrightarrow} \mathtt{P}_n\ \Gamma_n\ \sigma_n$. The program state trace that the execution ξ of program P gives rise to is denoted by $ProgTrace(\xi, \mathtt{P}) = \langle\Gamma_0, \Gamma_1, ..., \Gamma_n\rangle$. The semantics of an OTFJ program P is the set of $(\xi, ProgTrace(\xi, \mathtt{P}))$ pairs that P gives rise to according to the rules described above.

Discussion: The OTFJ semantics makes explicit using the predicates and STM state transformer functions in Fig. 4 the interaction between the program and the STM implementation. Most of the internal mechanics of the STM is not modeled in the OTFJ semantics, e.g., the STM state σ is left unspecified, but the commit action is conditional on the *OK2Cmt* predicate on the STM state, and the STM state changes in reaction to read, write accesses, and commit and undo actions. This way of modeling the program-STM interaction facilitates the modular correctness proof described in this paper.

READ

$$\frac{\mathcal{E} = \mathcal{E}'.(\mathtt{l}, \mathtt{s}) : \rho \qquad \mathcal{S}(\mathtt{r}) = \mathtt{C}(\overline{\mathtt{r}}) \qquad \mathcal{E}'' = \mathcal{E}'.(\mathtt{l}, \mathtt{s}) : (\rho.\mathtt{r} \overset{rd}{\mapsto} \mathtt{C}(\overline{\mathtt{r}}))}{read(\mathcal{S}, \mathtt{r}, \mathcal{E}) = \langle\mathcal{S}', \mathcal{E}'', \mathtt{C}(\overline{\mathtt{u}})\rangle}$$

NON-DET. FAILED READ

$$\frac{\mathcal{E} = \mathcal{E}'.(\mathtt{l}, \mathtt{s}) : \rho \qquad \mathtt{r}'\ \mathtt{fresh} \qquad \mathcal{E}'' = \mathcal{E}'.(\mathtt{l}, \mathtt{s}) : (\rho.\mathtt{r} \overset{rd}{\mapsto} \mathtt{r}')}{read(\mathcal{S}, \mathtt{r}, \mathcal{E}) = \langle\mathcal{S}', \mathcal{E}'', \rangle}$$

WRITE

$$\frac{\mathcal{S}(\mathtt{v}) = \mathtt{C}(\overline{\mathtt{r}}') \qquad \mathcal{S}' = \mathcal{S}[\mathtt{v} \mapsto \mathtt{C}(\overline{\mathtt{r}})] \qquad \mathcal{E} = \mathcal{E}'.(\mathtt{l}, \mathtt{s}).\rho \qquad \mathcal{E}'' = \mathcal{E}'.(\mathtt{l}, \mathtt{s}).\rho.\mathtt{v} \overset{wr}{\mapsto} \mathtt{C}(\overline{\mathtt{r}}').\mathtt{v} \overset{wr}{\mapsto} \mathtt{C}(\overline{\mathtt{r}})}{write(\mathtt{v} \mapsto \mathtt{C}(\overline{\mathtt{r}}), \mathcal{E}, \mathcal{S}) = \langle\mathcal{E}'', \mathcal{S}'\rangle}$$

EXTEND

$$\frac{\mathcal{S}' = \mathcal{S}[\mathtt{v} \mapsto \mathtt{C}(\overline{\mathtt{null}})] \qquad \mathcal{E} = \mathcal{E}''.(\mathtt{l}, \mathtt{s}).\rho \qquad \mathcal{E}' = \mathcal{E}''.(\mathtt{l}, \mathtt{s}).\rho.\mathtt{v} \overset{wr}{\mapsto} \bot.\mathtt{v} \overset{wr}{\mapsto} \mathtt{C}(\overline{\mathtt{null}})}{extend(\mathcal{S}, \mathtt{r} \mapsto \mathtt{C}(\overline{\mathtt{null}}), \mathcal{E}) = (\mathcal{S}', \mathcal{E}')}$$

START

$$\frac{\Gamma = \mathtt{t}, \mathcal{E}.\Gamma' \qquad \Gamma'' = \mathtt{t}, (\mathcal{E}.(\mathtt{l}, \mathtt{s}) : \langle\rangle).\Gamma'}{\Gamma'' = start(\mathtt{l}, \mathtt{s}, \mathtt{t}, \Gamma)}$$

SPAWN

$$\frac{\Gamma = \mathtt{t}, \mathcal{E}.\Gamma' \qquad \Gamma'' = \mathtt{t}', \langle\rangle\ .\ \Gamma'}{spawn(\mathtt{t}, \mathtt{t}', \Gamma) = \Gamma''}$$

UNDO EMPTY LOG

$$\frac{\Gamma = \mathtt{t}, \mathcal{E}.\Gamma' \qquad \mathcal{E} = \mathcal{E}'.(\mathtt{l}, \mathtt{s}) : \langle\rangle \qquad \Gamma'' = \mathtt{t}, \mathcal{E}'.\Gamma'}{\langle\Gamma_0, \Gamma_1, ..., \Gamma_p, \Gamma''\rangle, \langle\alpha_0, \alpha_1, ..., \alpha_p, \epsilon\rangle = undo((\Gamma, \mathtt{t}, \mathtt{l}, \langle\rangle), \langle\Gamma_0, \Gamma_1, ..., \Gamma_p\rangle, \langle\alpha_0, \alpha_1, ..., \alpha_p\rangle)}$$

UNDO LAST LOG READ ENTRY

$$\frac{\Gamma = \mathtt{t}, \mathcal{E}.\Gamma' \qquad \mathcal{E} = \mathcal{E}'.(\mathtt{l}, \mathtt{s}) : \rho.\mathtt{r} \overset{rd}{\mapsto} \mathtt{C}(\overline{\mathtt{r}}) \qquad \mathcal{E}'' = \mathcal{E}'.(\mathtt{l}, \mathtt{s}) : \rho \qquad \Gamma'' = \mathtt{t}, \mathcal{E}''.\Gamma' \qquad \tilde{\Gamma} = undo(\Gamma'', \mathtt{t}, \rho, \mathtt{l})}{\langle\Gamma_0, \Gamma_1, ..., \Gamma_p, \tilde{\Gamma}\rangle, \langle\alpha_0, \alpha_1, ..., \alpha_p, \epsilon\rangle = undo((\Gamma, \mathtt{t}, \rho.\mathtt{r} \overset{rd}{\mapsto} \mathtt{C}(\overline{\mathtt{r}}), \mathtt{l}), \langle\Gamma_0, \Gamma_1, ..., \Gamma_p\rangle, \langle\alpha_0, \alpha_1, ..., \alpha_p\rangle)}$$

UNDO LAST LOG WRITE ENTRY

$$\frac{\Gamma = \mathtt{t}, \mathcal{E}.\Gamma' \qquad \mathcal{E} = \mathcal{E}'.(\mathtt{l}, \mathtt{s}) : \rho.\mathtt{r} \overset{wr}{\mapsto} \mathtt{C}(\overline{\mathtt{r}}).\mathtt{r} \overset{wr}{\mapsto} \mathtt{C}'(\overline{\mathtt{r}'}) \qquad \mathcal{E}'' = \mathcal{E}'.(\mathtt{l}, \mathtt{s}) : \rho \qquad \Gamma' = (\mathcal{S}, \overline{(\mathtt{t}, \mathcal{E})}) \qquad \mathcal{S}' = \mathcal{S}[\mathtt{r} \mapsto \mathtt{C}(\overline{\mathtt{r}})] \qquad \Gamma'' = (\mathcal{S}', \overline{(\mathtt{t}, \mathcal{E})}) \qquad \Gamma''' = \mathtt{t}.\mathcal{E}''.\Gamma'' \qquad \tilde{\Gamma} = undo(\Gamma''', \mathtt{t}, \rho, \mathtt{l})}{\langle\Gamma_0, ..., \Gamma_p, \tilde{\Gamma}\rangle, \langle\alpha_0, ..., \alpha_p, (\mathtt{t}, \mathtt{l}, wr(\mathtt{r}, \mathtt{C}'(\overline{\mathtt{r}'}), \mathtt{C}(\overline{\mathtt{r}}).\mathtt{r}))\rangle = undo((\mathtt{t}, \mathtt{l}, \rho.\mathtt{r} \overset{wr}{\mapsto} \mathtt{C}(\overline{\mathtt{r}}).\mathtt{r} \overset{wr}{\mapsto} \mathtt{C}'(\overline{\mathtt{r}}), \mathtt{l}), \langle\Gamma_0, ..., \Gamma_p\rangle, \langle\alpha_0, ..., \alpha_p\rangle)}$$

COMMIT TOP-LEVEL TRANSACTION

$$\frac{\Gamma = \mathtt{t}, \mathcal{E}.\Gamma' \qquad \mathcal{E} = (\mathtt{l}, \mathtt{s}) : \rho \qquad \mathcal{E}' = \langle\rangle \qquad \tilde{\Gamma} = \mathtt{t}, \mathcal{E}'.\Gamma'}{commit(\mathtt{t}, \mathcal{E}, \Gamma) = \tilde{\Gamma}}$$

COMMIT NESTED TRANSACTION

$$\frac{\Gamma = \mathtt{t}, \mathcal{E}.\Gamma' \qquad \mathcal{E} = \mathcal{E}'.(\mathtt{l}', \mathtt{s}') : \rho', (\mathtt{l}, \mathtt{s}) : \rho \qquad \mathcal{E}'' = \mathcal{E}'.(\mathtt{l}', \mathtt{s}') : \rho'.\rho \qquad \Gamma'' = \mathtt{t}, \mathcal{E}''.\Gamma' \qquad \tilde{\Gamma} = commit(\mathtt{t}, \mathcal{E}'', \Gamma'')}{commit(\mathtt{t}, \mathcal{E}, \Gamma) = \tilde{\Gamma}}$$

Figure 4. Semantics for predicates.

3. Algorithm-Level Correctness Proof

3.1 Defining Correctness

We call a transaction with label $\overline{\mathtt{l}}$ *completed in* ξ if $\overline{\mathtt{l}}$ occurs in ξ and ξ contains either a commit or undo (roll-back) action for $\overline{\mathtt{l}}$. We call commit and undo actions *completion actions*. $CompLbls(\xi)$ denotes the set of labels of transactions completed in ξ. An execution $\xi = (\mathtt{t}_0, \overline{\mathtt{l}_0}, \alpha_0), (\mathtt{t}_1, \overline{\mathtt{l}_1}, \alpha_1), ..., (\mathtt{t}_n, \overline{\mathtt{l}_n}, \alpha_n)$, is *serial* iff for all i, j, and k such that $0 \leq i < j \leq k \leq n$ and $\overline{\mathtt{l}_i} = \overline{\mathtt{l}_k} \in CompLbls(\xi)$ it is the case that $\overline{\mathtt{l}_k} \lhd \overline{\mathtt{l}_j}$. In words, actions comprising completed or rolled-back transactions are either contiguous or interleaved with actions of transactions nested inside them. An execution is said to be *conflict free* if it does not contain an undo action.

Given an execution ξ and a thread identifier $\mathtt{t}$, let $\xi|_{\mathtt{t}}$ denote the subsequence of actions of ξ which are performed by $\mathtt{t}$, and let $\xi\lfloor_{\mathtt{t}}$ be the subsequence of $\xi|_{\mathtt{t}}$ obtained by removing actions by transactions that are eventually undone within ξ. Two executions ξ and $\tilde{\xi}$ of a program $\mathtt{P}$ are said to be *equivalent* if they (i) start at the same state, (ii) have the same set of thread identifiers, (iii) their corresponding program state traces $ProgTrace(\xi)$ and $ProgTrace(\tilde{\xi})$ have the same end state and (iv) if for each $\mathtt{t} \in Tid$, $\xi|_{\mathtt{t}} = \tilde{\xi}\Big|_{\mathtt{t}}$. These two executions are said to be *equivalent modulo undo's* if for each $\mathtt{t} \in Tid$, $\xi\lfloor_{\mathtt{t}} = \tilde{\xi}\lfloor_{\mathtt{t}}$. The composition of a program $\mathtt{P}$ with a given STM implementation is said to be *serializable* iff every execution ξ that the program-STM implementation pair can produce is equivalent to a serial execution $\tilde{\xi}$. The program-STM composition is *purely serializable* iff every execution ξ produced by it is equivalent modulo undo's to a serial, conflict-free execution $\hat{\xi}$ of $\mathtt{P}$.

We define the *write set* and the *read set* of a transaction log ρ as follows. $WrSet(\rho)$ consists of references such that $\mathtt{r} \in WrSet(\rho)$ iff there exists some $\mathtt{C}(\overline{u})$ for which $(\mathtt{r} \overset{wr}{\mapsto} \mathtt{C}(\overline{u})) \in \rho$. Similarly, let $RdSet(\rho)$ is defined as follows: $\mathtt{r} \in RdSet(\rho)$ iff there exists some $\mathtt{C}(\overline{u})$ for which $(\mathtt{r} \overset{rd}{\mapsto} \mathtt{C}(\overline{u})) \in \rho$. We overload the definition of write set such that for a committing innermost transaction with label $\overline{\mathtt{l}}.\mathtt{l}$, letting ρ_{cmt} be the transaction log for $\overline{\mathtt{l}}.\mathtt{l}$ when the commit action is executed, $WrSet(\overline{\mathtt{l}}.\mathtt{l}) = WrSet(\rho_{cmt})$ and $RdSet(\overline{\mathtt{l}}.\mathtt{l}) = RdSet(\rho_{cmt})$. Enclosing transactions have the nested transactions logs appended to them at commit time, i.e. $WrSet(\overline{\mathtt{l}'}) \subseteq WrSet(\overline{\mathtt{l}})$ iff $\overline{\mathtt{l}'} \lhd \overline{\mathtt{l}}$.

The Exclusive Writes (EW) Property: Consider an execution $\xi = \mathtt{P}_0\ \Gamma_0 \overset{\mathtt{t}_0}{\Longrightarrow} \alpha_0\mathtt{P}_1\ \Gamma_1 \overset{\mathtt{t}_1}{\Longrightarrow} \alpha_1 ... \mathtt{P}_{n-1}\ \Gamma_{n-1} \overset{\mathtt{t}_{n-1}}{\Longrightarrow} \alpha_{n-1}\mathtt{P}_n\ \Gamma_n$ where $\alpha_{n-1} = cmt$, or $\alpha_{n-1} = undo$ for a transaction with label $\overline{\mathtt{l}}.\mathtt{l}$, and $\Gamma_{n-1} = (\mathcal{S}_{n-1}, \mathtt{t}, \overline{\mathcal{E}}.\mathtt{t}, \mathcal{E})$, and the log of the transaction $\overline{\mathtt{l}}.\mathtt{l}$ is such that $\mathcal{E} = \mathcal{E}'.(\mathtt{l}, \mathtt{s}).\rho$. For each $\mathtt{v}$ in $WrSet(\rho)$, let $FirstWrite(\xi, \mathtt{v}, \mathtt{l})$ be the smallest i for which $\alpha_i = wr(\mathtt{v}, \mathtt{x}, \mathtt{y})$, $\mathtt{t}_i = \mathtt{t}$ and $inTrans(\mathtt{t}, \overline{\mathtt{l}}.\mathtt{l}, \Gamma_i)$.

An execution $\tilde{\xi}$ is said to have the *exclusive writes (EW)* property iff for every such prefix ξ of $\tilde{\xi}$ and every $\mathtt{v} \in WrSet(\rho)$, whenever there is a write action $\alpha_j = wr(\mathtt{v}, \mathtt{x}', \mathtt{y}')$ where $FirstWrite(\xi, \mathtt{v}, \mathtt{l}) \leq j \leq n$, the thread performing α_j satisfies $inTrans(\mathtt{t}_j, \overline{\mathtt{l}_1}, \Gamma_j)$ for a transaction label $\overline{\mathtt{l}_1}$ such that $\overline{\mathtt{l}}.\mathtt{l} \lhd \overline{\mathtt{l}_1}$. In words, if there is another write action between a write action in a transaction and the corresponding commit or undo action, then the write action needs to belong to either this transaction or another transaction nested within this one.

The Valid Reads (VR) Property: Consider an execution $\xi = \mathtt{P}_0\ \Gamma_0\ \sigma_0 \overset{\alpha_0}{\Longrightarrow} \mathtt{P}_1\ \Gamma_1\ \sigma_1 \overset{\alpha_1}{\Longrightarrow} ... \mathtt{P}_{n-1}\ \Gamma_{n-1}\ \sigma_{n-1} \overset{\alpha_{n-1}}{\Longrightarrow} \mathtt{P}_n\ \Gamma_n\ \sigma_n$ where $\alpha_{n-1} = cmt$ and $\Gamma_{n-1} = (\mathcal{S}_{n-1}, \mathtt{t}, \overline{\mathcal{E}}.\mathtt{t}, \mathcal{E})$, for all i, α_i is performed by $\mathtt{t}_i$, and the log ρ of the committing transaction $\overline{\mathtt{l}}.\mathtt{l}$ is such that $\mathcal{E} = \mathcal{E}'.(\mathtt{l}, \mathtt{s}).\rho$. For each $\mathtt{r}$ in $RdSet(\rho)$, let $FirstRead(\xi, \mathtt{r}, \mathtt{l})$ be the smallest i for which $\alpha_i = rd(\mathtt{r}, \mathtt{x})$, $\mathtt{t}_i = \mathtt{t}$, and $inTrans(\mathtt{t}, \overline{\mathtt{l}}.\mathtt{l}, \Gamma_{i-1})$. An execution $\tilde{\xi}$ is said to have the *valid reads (VR)* property iff for every prefix ξ of $\tilde{\xi}$ of the form above and every $\mathtt{r} \in RdSet(\rho)$, the following condition holds: There is no other transaction $\overline{\mathtt{l}'}$ performed by another thread $\mathtt{t} \neq \mathtt{t}_{n-1}$ such that α_q is the completion action of $\overline{\mathtt{l}'}$ ($q = \infty$ if $\overline{\mathtt{l}'}$ is not completed in ξ) $\mathtt{r} \in WrSet(\overline{\mathtt{l}'})$, $p = FirstWrite(\xi, \mathtt{r}, \overline{\mathtt{l}'})$ and the intervals $[p, q]$ and $[FirstRead(\xi, \mathtt{r}, \overline{\mathtt{l}}, n-1]$ intersect.

The Correct Undo's (CU) Property: Consider an execution $\tilde{\xi} = \mathtt{P}_0\ \Gamma_0 \overset{\mathtt{t}_0}{\Longrightarrow} \alpha_0\mathtt{P}_1\ \Gamma_1 \overset{\mathtt{t}_1}{\Longrightarrow} \alpha_1 ... \mathtt{P}_{n-1}\ \Gamma_{n-1} \overset{\mathtt{t}_{n-1}}{\Longrightarrow} \alpha_{n-1}\mathtt{P}_n\ \Gamma_n$ where for some $k < n$, $\alpha_{k-1} = undo$ for a transaction label $\overline{\mathtt{l}}.\mathtt{l}$. Let $FirstWrite$ be the first index for which $inTrans(\mathtt{t}_i, \overline{\mathtt{l}}.\mathtt{l}, \Gamma_i)$. Let $\Gamma_n = (\mathcal{S}_n, (\mathtt{t}', \mathcal{E}'))$. An execution ξ is said to have the *correct undo's (CU) property* iff for each prefix $\tilde{\xi}$ of ξ of the form above, for each reference $\mathtt{v} \in WrSet(\overline{\mathtt{l}}.\mathtt{l})$ it is the case that $\Gamma_{i-1}(\mathtt{v}) = \Gamma_n(\mathtt{v})$. In words, the object referred to by $\mathtt{v}$ must be the same right before the transaction with label $\overline{\mathtt{l}}.\mathtt{l}$ modifies it and right after the rollback of $\overline{\mathtt{l}}.\mathtt{l}$ is completed.

Serialization order: For a given execution ξ, we define a *serialization order* on all completed transactions $TransLbl$. This is a total order that corresponds to the order of commit or undo actions in ξ. More formally, for a transaction label $\overline{\mathtt{l}}$, let $SerIndex(\xi, \overline{\mathtt{l}}) = i$ such that α_i is either the commit or the undo action for transaction $\overline{\mathtt{l}}$. Then $\overline{\mathtt{l}} \preceq \xi\overline{\mathtt{l}'}$ iff $SerIndex(\xi, \overline{\mathtt{l}}) \leq SerIndex(\xi, \overline{\mathtt{l}'})$.

LEMMA 1. *Consider an execution ξ with the VR and EW properties. $\xi = \mathtt{P}_0\ \Gamma_0 \overset{\mathtt{t}_0}{\Longrightarrow} \alpha_0\mathtt{P}_1\ \Gamma_1 \overset{\mathtt{t}_1}{\Longrightarrow} \alpha_1 ... \mathtt{P}_{n-1}\ \Gamma_{n-1} \overset{\mathtt{t}_{n-1}}{\Longrightarrow} \alpha_{n-1}\mathtt{P}_n\Gamma_n$ where $\xi \in Execs(\mathtt{P})$ and $\mathtt{t}_i \neq \mathtt{t}_{i+1}$ and, if both α_i and α_{i+1} are performed by transactions, the completion action of the transaction performing α_{i+1} comes* **before** *that of the transaction performing α_{i+1}.*

Let $\xi' = \alpha_0, \alpha_1, ..., \alpha_{i-1}, \alpha_{i+1}, \alpha_i, \alpha_{i+2}, \alpha_{i+3} ... \alpha_n$, i.e. ξ' is the same as ξ except that actions α_i and α_{i+1} are swapped. Then, ξ' is a legal execution of $\mathtt{P}$ and ξ and ξ' are equivalent.

THEOREM 1. *If all executions of a program $\mathtt{P}$ have the VR and EW properties, then every execution ξ of $\mathtt{P}$ is serializable and is equivalent to a serial execution ξ^{sr} which preserves the serialization order of ξ.*

THEOREM 2. *If every execution ξ of a program $\mathtt{P}$ has the VR, EW, and correct undo's properties, then every execution ξ of $\mathtt{P}$ is* purely serializable. *ξ is* equivalent modulo undo's *to a serial, conflict-free execution $\hat{\xi}$ of $\mathtt{P}$.*

The proofs of Theorems 1 and 2 are given in the Appendix.

4. Verifying the Bartok STM Implementation

The Bartok STM is a write-in-place, lazy-invalidate STM implementation. The EW, VR and CU properties capture the reasoning by which this style of STM guarantees pure serializability. EW, VR and CU are properties of executions, whereas an STM maintains some auxiliary execution history data with each object based on which it aims to ensure the EW, VR and CU properties. The key challenge in verifying an STM implementation is ensuring that this conflict-detection mechanism based on object-local and thread-local data manages inter-transaction interference properly.

In Section 4.1, we give an overview of the Bartok STM implementation, and, in Section 4.2, we describe how we formally relate executions produced by the Bartok runtime to executions produced by OTFJ programs. Then, in Sections 4.4-4.6, we prove lemmas on Bartok executions. These lemmas together constitute a sufficient condition for a Bartok execution mapped to an OTFJ execution satisfying the EW, VR and CU properties, i.e. purely serializable.

4.1 Modeling the Bartok STM

The description in this section is aimed at illustrating the verification-related aspects of the Bartok STM. For a more accurate description , see [5]. In the Bartok STM, each thread has an associated data structure called the transaction manager. Transaction managers are responsible for managing the interaction between threads. Each object is augmented with extra fields we refer to as the "STM metadata". STM metadata is invisible to the programmer. Transaction managers do not modify programmer-visible data except when a transaction is rolled back.

The object metadata is represented in the `STM_Word` and the `STM_Snapshot`. These two store the state of the object which consists of (i) whether the object is currently open for write ("owned") by a transaction, (ii) the owner transaction, and (iii) the version number of the object. The version number is an (unbounded) integer counter incremented by transactions updating an object. Reads and updates to a `STM_Snapshot-STM_Word` pair are guaranteed to be atomic.

The beginning of a transaction is marked by a call to the "`DTMStart`" function. Before fields of an object are read, the "`DTMOpenForRead`" (See pseudocode in the Appendix.) function must be called with the appropriate arguments. Similarly, before object fields are written to, the "`DTMOpenForUpdate`" function must be called. The STM implementation is notified of object fields updated during a transaction using the `DTMLogFieldStore` function which appends to an undo log an entry with the old and updated value of the field.

To complete a transaction, the "`DTMCommit`" function is called. To record and manage conflicts, the transaction managers keep logs of read and write accesses performed by the transaction. When "`DTMCommit`" is called, the transaction manager first calls "`ValidateRead`" for each object in the read log to ensure that no other concurrent transaction has opened the object for write during the timespan from `OpenForRead` to `ValidateRead`. If all `ValidateRead` calls return successfully, then the transaction manager executes `CloseSTMWord` for all objects in the write log of the transaction. Each object becomes available for conflict-free read and write access after `CloseSTMWord` successfully completes and increments the version number of the object. For a successfully-committing transaction, the first successful `ValidateRead` execution constitutes the commit point of the transaction [5]. For transactions that detect a conflict in one of the `ValidateRead` calls or experienced a conflict during one of the `OpenForUpdate` calls, the transaction is rolled back. The rolling back of the log is modeled by repeated calls to the `UndoLastLogEntry` function.

4.2 Modeling Bartok STM Executions

Modeling Bartok STM Executions: The OTFJ semantics has built into it an idealized, high level of granularity STM that supports the following atomic actions: *Opn4Rd*, *Opn4Wrt*, cmt, and $undoLLE$. In OTFJ semantics, these actions correspond to atomic transitions in STM state and in Fig. 3, these actions are denoted by τ and distinguished from actions that transform the program state and the program.

To model Bartok STM implementations, we provide a lower-level semantics, TFJ-Bk, for OTFJ programs composed with the Bartok STM implementation. TFJ-Bk is different from OTFJ semantics in two regards. First, each of the actions marked by a τ in the OTFJ semantics is replaced by a finite *sequence* $\overline{\varepsilon} = \varepsilon_0, \varepsilon_1, \varepsilon_2, ..., \varepsilon_k$ of actions by the same thread executing the action. Actions marked by ε_i model the actions in the execution of the functions `OpenForRead`, `OpenForUpdate`, `DTMCommit` and `UndoLastLogEntry`.

```
CheckNOWS()
1      Havoc(obj); // state and metadata
2  OpenForUpdate(Tx_good, obj);
3  assume( ExclusiveOwner(Tx_good, obj) );
4  assume( Tx_bad != Tx_good);
5      OpenForUpdate(Tx_bad, obj);
6  assert(ExclusiveOwner(Tx_good, obj));
7  assert(Tx_bad.invalid);


CheckUndoLogEntry()
1  obj_1 = obj.clone();
2  Update_k(logEntry_k)
3  Undo_k(logEntry_k)
4  assert( obj.Equals(obj_1) );
CheckVRS()
0      interferedWith = false;
1      InterfereWith(Tx, obj);
2  OpenForRead(Tx,obj);
3      InterfereWith(Tx, obj);
4  if (*) OpenForUpdate(Tx,obj);
5      InterfereWith(Tx, obj);
6  ValidateRead(Tx,obj);
7  assert(interferedWith <==> Tx.invalid);


InterfereWith (Transaction Tx, Object obj)
1  while (*) {
2    Tx_bad = non-deterministically
3             chosen transaction
4    assume(Tx_bad != Tx);
5    if (*) OpenForUpdate(Tx_bad, obj);
6    if (*) CloseSTMWord(Tx_bad, obj);
7    if (line 5 or 6 succeed)
8       interferedWith = true;
```

Figure 5. The assertion checks for proving the NOWS and VRS properties.

4.3 Reducing the Set of TFJ-Bk Executions

We first reduce the set of TFJ-Bk executions that we need to be concerned with using partial order reductions. We say two actions in a TFJ-Bk execution are *dependent* if they are performed by the same thread or they both refer to the same variable and at least one of them modifies this variable. Actions that are not dependent are called *independent*. An execution ξ_1^{Bk} is *partial-order equivalent* to ξ_2^{Bk} if ξ_2^{Bk} is obtained from ξ_1^{Bk} by a sequence of swaps of consecutive independent actions.

In order to be able to apply partial-order reductions to more executions, in the same manner as Qadeer et al. [3], we abstract the Bartok STM implementation in two ways: First, the compare-and-swap (CAS) operation is replaced by an ND-CAS operation. An ND-CAS operation can only succeed when a CAS operation can, but can fail non-deterministically even when the CAS would have succeeded. A failing ND-CAS is independent of all actions by other threads. Second, read field actions in transactions that lead to failed `ValidateRead`'s can read arbitrary non-deterministic values. Observe that OTFJ has this non-determinism built into its semantics as well. These abstractions are valid since they only add to the set of possible behaviors and we focus exclusively on safety properties. They are conceptual devices for simplifying the analysis in order to make more actions independent of each other – the actual STM is not modified.

LEMMA 2. *Each Bartok STM execution is partial-order equivalent to one where all actions corresponding to any particular execution of one of the following functions appear consecutively:* `DTMStart`, `DTMOpenForRead`, `DTMOpenForUpdate`, `CloseSTMWord`, `ValidateRead`. *We call such executions coarse atomic.*

The proof of the lemma is left to the Appendix.

The lemma allows us to consider thread interleavings at the level of the functions listed. If we can prove that all coarse-atomic Bartok STM executions satisfy the conditions we are after, then we have proved the same for all Bartok STM executions. In the rest of this paper, we restrict our attention to coarse atomic Bartok STM executions.

4.4 Verifying the "Exclusive Writes" Property

Consider a Bartok STM execution ξ^{Bk} in which transaction `tx` performs a write access to a field of object `o`, and in which the i-th action of ξ^{Bk} is the representative action of an `OpenForUpdate` of object `o` by transaction `tx`, and the j-th action of ξ^{Bk} is either the representative action of a `CloseSTMWord` executed on `o` by `tx`. We refer to the interval $[i, j]$ as the *write span of object* `o` *in* `tx`, denoted by $WriteSpan(\xi^{Bk}, \texttt{tx}, \texttt{o})$.

We prove that the Bartok STM guarantees the following *non-overlapping write spans (NOWS)* property: Let $WriteSpan(\xi, \texttt{tx}, \texttt{o}) = [i, j]$. Then ξ does not have as its j-th action for any $i < j < k$ the representative action of a successful `OpenForUpdate` of `o` by another transaction $\texttt{tx}'$ executed by another thread, i.e., write spans of an object in two different transactions do not overlap.

Expressing the NOWS property as a sequential Spec# assertion: We modeled the Bartok STM in the Spec# language and used the Boogie verification tool for sequential Spec# programs. One novel contribution of this paper is the statement of non-interference requirements in the form of assertion checks in sequential Spec# programs. This approach is illustrated in the code for `CheckNOWS` in Fig. 5. Here, `Tx_good` successfully opens object `obj` for update. `ExclusiveOwner(Tx_good, obj)` is a propositional formula which states that `Tx_good` is exclusive owner of `obj` according to the `STM_Word` for `obj`. Lemma 2 allows us to consider only coarse atomic thread interleavings in which an `OpenForUpdate` must run in its entirety before control switches to another thread. `Tx_bad` models the first potential offending transaction by another thread that attempts to perform an `OpenForUpdate` within the write span of `obj` in `Tx_good`. It is sufficient to consider this scenario because only transactions that already have an object open for update attempt to modify the STM metadata using `CloseSTMWord`. The interaction of the threads running `Tx_good` and `Tx_bad` is mimicked `CheckNOWS`, where actions implementing the function calls in lines 2 and 5 manipulate program and STM state in exactly the same way as two distinct C# threads running these functions. Using Boogie, this assertion check was verified in under a minute.

4.5 Verifying the "Valid Reads" Property

Consider a Bartok STM execution ξ^{Bk} where `tx` reads a field of object `o`, and in which the i-th action is the representative action of an `OpenForRead` of object `o` by transaction `tx`, and the j-th action is the representative action of a *successful* `ValidateRead` executed on `o` by `tx`. The interval $[i, j]$ is the *read span* of `o` by `tx`, denoted by $ReadSpan(\xi, \texttt{tx}, \texttt{o})$. Let `tx` and $\texttt{tx}'$ be two transactions performed by two different threads in an execution ξ^{Bk} and let `tx` perform a read access to `o` that is successfully validated and let $\texttt{tx}'$ perform a write access to fields of `o`. Then $ReadSpan(\xi, \texttt{tx}, \texttt{o})$ and $WriteSpan(\xi, \texttt{tx}', \texttt{o})$ do not intersect. This is referred to as the *valid read spans (VRS)* property.

Representing the VRS property as a sequential assertion check: The modeling of possible interference scenarios in the VRS property is more complicated than that for NOWS. In `CheckVRS` in Fig. 5, `Tx` represents the transaction that has object `obj` open for read. Before `Tx` runs `ValidateRead` on `obj`, an arbitrary number of other threads can run `OpenForUpdate` and `CloseSTMWord` on `obj` and `Tx` itself can open `obj` for write. The function `InterfereWith` models an arbitrary sequence of such interfering actions by other transactions. `CheckVRS` models interference by other threads as a sequential program. For every sequence of actions that `Tx` and an arbitrary number of other interleaved offending threads and transactions perform, there is a corresponding behavior of the sequential procedure `CheckVRS` where `Tx_bad` in each execution of the loop in `InterfereWith` is chosen appropriately.

Experience: While checking the VRS property, we found out that it did not hold of our model, which was based on the pseudo-code published in [5] and augmented with features extracted from the C# code for Bartok. One possible interleaving of offending threads was not covered in `ValidateRead` and a transaction that should have been invalidated was not (See Appendix for details). Closer examination of the current version Bartok STM code revealed that the implementation code was structured differently and did not contain this error. We were also provided "challenge bugs" – bugs that existed in earlier versions. We were provided two such challenge bugs and both of these bugs were detected as assertion violations in `CheckVRS`.

In order for Boogie to complete the verification task, we had to provide pre- and post-conditions for `InterfereWith`, which required a post-condition for the while loop in `InterfereWith`. The loop post-condition stated that if `interferedWith` is set to true, then the version number of `obj` is increased in the loop. With the proper corrections, `CheckVRS` was shown to be free of assertion violations. Boogie runs for both succeeding and failing assertion checks completed in under five minutes.

4.6 Verifying the "Correct Undo's" Property

We sketch the verification of this property. More details are available in the Appendix. The verification builds on the NOWS and VRS properties, which, along with the abstraction of allowing reads in rolled-back transactions to return arbitrary values imply that for an object in the write set of a rolled back transaction `Tx`, no write accesses are performed by another thread while `Tx` is being rolled back, and all reads during this time (since they conflict with `Tx`) are allowed to return arbitrary values. Therefore, all accesses by other threads to objects owned by `Tx` during rollback can be commuted to the right of transaction `Tx`. This results in a completely serial execution of the undo function in the Bartok STM. We say such executions have the *serial undo (SU) property.* For serially executed rollback operations, the OTFJ semantics and the Bartok implementation code are identical in structure and are easily proven to be equivalent.

4.7 Putting it All Together

So far we have proven that every Bartok STM execution is partial-order equivalent to a coarse atomic Bartok STM execution ξ^{Bk} that has the NOWS, VRS, SU and correct undo's properties. To relate such a a ξ^{Bk} to an OTFJ execution ξ, we perform the following steps. We denote the resulting transformation $h(\xi^{Bk}) = \xi$.

- For successfully committing transactions, insert a `commit` action immediately after the last action belonging to the first successful `ValidateRead` in ξ.
- For every transaction $\overline{\texttt{l}}.\texttt{l}$ that aborts and rolls back, insert a $\tau = undoLLE(\overline{\texttt{l}}\texttt{l})$ immediately before every write action to a program variable performed in the context of the transaction rollback. Insert the action $(\texttt{t}, \overline{\texttt{l}}.\texttt{l}, undo)$ immediately after the last such write to a program variable.

- Group any sequence of actions $\overline{\varepsilon} = \varepsilon_0, \varepsilon_1, \varepsilon_2, ..., \varepsilon_k$ performed by a single thread as part of a single STM function into a single atomic action.

Note that h leaves intact all actions acting on programmer-visible variables.

THEOREM 3. *For every Bartok STM execution ξ^{Bk} satisfying the NOWS, VRS, SU, and correct undo's properties, $h(\xi^{Bk})$ is an OTFJ exceution that has the EW, VR and correct undo's properties. Therefore, $h(\xi^{Bk})$ is purely serializable.*

The proof of Theorem 3 is given in the Appendix. Theorem 3 implies that the checks performed on the Bartok STM are sufficient to prove correctness, i.e., that every sequence of reads and updates of programmer-visible variables in a Bartok STM execution could have been produced by a purely serializable OTFJ execution.

5. Previous Work

Earlier work on verifying TM's has mostly concentrated on the top, algorithmic-level. Jagannathan et al. [6], Moore et al. [8], Abadi et al. [1] have explored conditions under which a finer-grained small-step semantics for TM's is a correct refinement of a coarse-grained, serial semantics for programs using TM's. These studies are similar to the top-level proof we present in Section 3. Our work takes this small-step semantics as a starting point and checks that the TM implementation satisfies it.

Cohen et al. [2] write an abstract model for a TM (the specification module) and an algorithm-level model (the implementation module) in TLA+. They then either deductively verify correctness or apply model checking for limited configurations, i.e., a small number of transactions, number of instructions in each transaction, etc. This work can be seen as a way to partly automate our top-level proof while applying it to different styles of STM's. The emphasis in our work, on the other hand, is the mechanical verification that the algorithm-level description is correctly implemented in STM code.

6. Conclusion

We presented a technique that makes possible modular verification of whether an STM implementation is a correct refinement of its algorithm-level description, which is separately shown to be serializable. We applied the technique to the Bartok STM implementation and were able to detect subtle implementation bugs. While we applied this approach to only one STM style and one particular implementation, we believe that it can be applied to other STM styles as well. Future work includes generalization of the technique to other STM algorithms, investigation of abstract interpretation techniques to automate more of the mechanical proof, and a formalization of the STM-garbage collector interface.

Acknowledgments

The author would like to thank Rustan Leino, Tim Harris, Shaz Qadeer, Mike Barnett, Dave Detlefs, Mike Magruder, Yossi Levanoni, and Jim Larus for helpful discussions and feedback, and support with Spec# and the Bartok STM implementation.

References

[1] Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of transactional memory and automatic mutual exclusion. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 63–74, New York, NY, USA, 2008. ACM.

[2] Ariel Cohen, John W. O'Leary, Amir Pnueli, Mark R. Tuttle, and Lenore D. Zuck. Verifying correctness of transactional memories. In *Proceedings of the 7th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 37–44. November 2007.

[3] Cormac Flanagan, Stephen N. Freund, and Shaz Qadeer. Exploiting purity for atomicity. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 221–231, New York, NY, USA, 2004. ACM.

[4] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2):5, 2007.

[5] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 14–25, New York, NY, USA, 2006. ACM.

[6] Suresh Jagannathan, Jan Vitek, Adam Welc, and Antony Hosking. A transactional object calculus. *Sci. Comput. Program.*, 57(2):164–186, 2005.

[7] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[8] Katherine F. Moore and Dan Grossman. High-level small-step operational semantics for transactions. In *35th ACM Symposium on Principles of Programming Languages*. Jan 2008.

Appendix

Pseudocode for certain Bartok STM functions is provided in Fig. 6

6.1 Proofs of Theorems and Lemmas

PROOF. *(Theorem 1):* For a given execution ξ of program P, we will prove the existence of such an ξ^{sr} by induction on $m = |CompLbls(\xi)|$, the number of labels of completed transactions. We will prove that each such ξ is equivalent to a serial execution ξ^{sr} where transactions occur in the same serialization order as ξ.

For $m = 0$, the claim is trivially true. Suppose the claim holds for $m = q$ and consider an execution ξ with $q + 1$ completed transactions. Let ξ_q be the prefix of ξ up to and including the q-th completion action. Then $\xi = \xi_q \,.\, \xi^{sf}$ for some execution suffix ξ^{sf}. By the inductive hypothesis, ξ_q is equivalent to a serial execution ξ_q^{sr}. Then ξ is equivalent to an execution $\xi^{\bullet} = \xi_q^{sr}.\xi^{sf}$.

Let α_i and α_j, the i-th and j-th actions in $\xi^{\bullet}$ be the q-th and $q + 1$-st completion actions in $\xi^{\bullet}$ executed by threads $\mathtt{t}_i$ and $\mathtt{t}_j$. Note that the serialization order, therefore, the order of completion actions in ξ and $\xi^{\bullet}$ are the same by the inductive hypothesis.

Let us write $\xi^{\bullet}$ as $\xi^{\bullet} = \xi_{(0,i)} \,.\, \alpha_i \,.\, \xi_{(i,j)} \,.\, \alpha_j \,.\, \xi_{(j,\infty)}$. All actions in $\xi_{(i,j)}$ not executed by $\mathtt{t}_j$ must have their completion actions later than α_j, i.e., not in $\xi^{\bullet}$. This is the case regardless of whether $\mathtt{t}_i = \mathtt{t}_j$. By the lemma above, all such actions can be commuted to the right of α_j while preserving their relative order. Thus $\xi^{\bullet}$ is equivalent to an execution of the form $\xi^{\bullet,sr} = \xi_{(0,i)}.\alpha_i.\xi'_{(i,j)}.\alpha_j.\xi'.\xi_{(j,\infty)}$, where, by construction, $\xi'_{(i,j)}$ is the subsequence of $\xi_{(i,j)}$ consisting of actions executed by $\mathtt{t}_j$ as part of the transaction completing at α_j and ξ' contains (in the same order) all other actions in $\xi_{(i,j)}$. By construction, $\xi^{\bullet,sr}$ is serial and preserves the serialization order in ξ. □

PROOF. *(Theorem 2):* Let ξ be as stated in the theorem. By Theorem 1, ξ is equivalent to a serial execution ξ^{sr}. Since ξ^{sr} is an execution of P, ξ^{sr} also has the valid undo's property. It is straightforward to see from the semantics of a serially-executed transaction that is rolled back that it does not modify the program or the program state. Therefore, in any serial execution ξ^{sr}, any execution fragment from the execution of an `onacid` action to the corresponding `undo` can be removed to yield another legal serial execution ξ^{sr} of the same program. By repeated application of this fragment removal procedure, one obtains a serial, conflict-free (i.e. no undo's) execution $\hat{\xi}$ that is equivalent modulo undo's to ξ^{sr}. □

```
void DTMOpenForUpdate(tm_mgr tx,
                      object obj) {

  // ** Atomic STM Metadata Read **
  word stm_word = GetSTMWord(obj);

  if (!IsOwnedSTMWord(stm_word)) {
    entry -> obj = obj;
    entry -> stm_word = stm_word;
    entry -> tx = tx;

    word new_stm_word =
           MakeOwnedSTMWord(entry);

    // ** Atomic compare and swap **
    // ** of STM Metadata         **
    if (OpenSTMWord(obj, stm_word,
                    new_stm_word)) {

      // Open succeeded:
      // Advance our log pointer
      entry ++;
    } else {
      // Open failed
      BecomeInvalid(tx);
    }
  } else if (tx ==
              GetOwnerFromSTMWord(stm_word)) {
      // Object already open for
      // update by current transaction
  } else {
      // Object already open for
      // update by another transaction: abort
      BecomeInvalid(tx);
  }
}

void CloseUpdatedObject(tm_mgr tx,
                        object obj,
                        update_entry *entry) {
  word old_stm_word = entry -> stm_word;
  word new_stm_word =
                GetNextVersion(old_stm_word);
  CloseSTMWord(obj, new_word);
}
```

```
void ValidateReadObject(tm_mgr tx, object obj,
                        read_entry *entry) {
  snapshot old_snapshot = entry -> stm_snapshot;

  // **     Atomic STM Metadata Read        **
  snapshot cur_snapshot = GetSTMSnapshot(obj);

  word cur_stm_word = SnapshotToWord(cur_snapshot);

  if (old_snapshot == cur_snapshot) {
    if (!IsOwnedSTMWord(cur_stm_word)) {
      // V1: Snapshot unchanged, no conflict
    } else if (GetOwnerFromSTMWord(cur_stm_word) == tx) {
      // V2: Opened by us for update before read
    } else {
      // V4: Opened for update by another tx
      BecomeInvalid(tx);
    }
  } else { // Snapshots mismatch: slow-path test on STM word

    word old_stm_word = SnapshotToWord(old_snapshot);
    if (!IsOwnedSTMWord(old_stm_word)) {
      if (old_stm_word == cur_stm_word) {
        // V1: OK: STM word inflated during the transaction
      } else if (!IsOwnedSTMWord(cur_stm_word)) {
        // V5: Conflicting update by another tx
        BecomeInvalid(tx);
      } else if (GetOwnerFromSTMWord(cur_stm_word) == tx) {
        // We opened the object for update...
        update_entry *update_entry =
                          GetEntryFromSTMWord(cur_stm_word);
        if (update_entry -> stm_word !=
                            SnapshotToWord(old_snapshot)) {
          // V5: ...but another tx opened and closed the
          // object for update before we opened it
          BecomeInvalid(tx);
        } else { } // V3: No intervening access by another tx
      } else { BecomeInvalid(tx);}  // V5: The object was
                                    // opened by another tx
    } else if (GetOwnerFromSTMWord(cur_stm_was) == tx) {
      // V2: Opened by us for update before read
    } else {
      // V4: STM word unchanged, but previously open for
      // update by another transaction
      BecomeInvalid(tx);
    }
  }
}
```

Figure 6. Pseudocode for some functions in the Bartok STM implementation.

PROOF. *(Lemma 2):*

Each of the methods listed performs a single atomic (CAS) operation of a single `STM_Word` or a single read and later a compare-and-swap of a single `STM_Word`. All of the rest of the actions performed by the function are thread-local variable accesses, which can be commuted to be adjacent to any action in the function. We designate as the *representative* action either the successful CAS operation, or, in functions where the CAS fails, the earlier read operation. In either case, using the abstract semantics for TFJ-Bk, all actions in the function can be commuted so they are adjacent to the representative action in the execution. For functions that execute a single CAS operation, all other actions are independent from all others performed by other threads. Therefore, they can all be moved towards the CAS operation to yield an equivalent execution. For functions which first perform a read and then a CAS, there are two cases to consider. In case the CAS succeeds, there are no writes to the `STM_Word` between the read and the CAS, thus, the preceding argument applies and all actions in the function can be moved towards the CAS. In case the CAS fails, since the failing ND-CAS operation commutes with all actions from other threads, all actions can be moved towards the initial atomic read to yield a coarse atomic execution. □

PROOF. *(Theorem 3):* The key difference between the Bartok STM and OTFJ executions is that certain atomic OTFJ actions τ were implemented as sequences of actions $\overline{\varepsilon} = \varepsilon_0, \varepsilon_1, \varepsilon_2, ..., \varepsilon_k$ in the Bartok STM. But the NOWS, VRS and correct undo's properties and the partial-order equivalences described in this section allow us to conclude that for each Bartok STM execution with the NOWS, VRS and correct undo's properties, there is an equivalent Bartok STM execution where all the actions in a sequence of the form $\overline{\varepsilon} = \varepsilon_0, \varepsilon_1, \varepsilon_2, ..., \varepsilon_k$ appear consecutively. By grouping such sequences into single atomic actions and observing that the STM-state transformer predicates are left unspecified for OTFJ, we conclude that every such $h(\xi^{Bk})$ is an OTFJ execution.

It is straightforward to see that if a Bartok STM execution ξ^{Bk} has the NOWS property, then $h(\xi^{Bk})$ has the exclusive writes property. In $h(\xi^{Bk})$, the commit action following a write comes before the `CloseSTMWord` action, therefore, all write actions happen in the write span of a transaction. The NOWS property implies that write spans do not overlap as required by the exclusive writes property. Since h inserts a `commit` action right before the first successful `ValidateRead`, the VRS property implies the VR property for the corresponding OTFJ execution in a straightforward manner. The argument presented later in the appendix for undo operations similarly implies that $h(\xi^{Bk})$ has the correct undo's property. □

6.2 The Bug in the STM Pseudocode

While checking the VRS property on the Bartok STM implementation, we found out that it in fact did not hold of our model, which was based on the pseudo-code published in [5] and augmented with features extracted from the C# code for Bartok. One possible interleaving of offending threads was not covered in `ValidateRead` and a transaction that should have been invalidated was not. Cases `V1-V5` of `ValidateRead` in Fig. 6 omitted a possibility. In the omitted case, another transaction (`Tx_other`) has `obj` open for write when `Tx` executes `DTMOpenForRead` and `Tx_other` runs `CloseSTMWord` before `Tx` runs `ValidateRead`. In this interleaving, case `V2` is executed erroneously whereas the transaction `Tx` should have been invalidated. Closer examination of the current version Bartok STM code revealed that the implementation code was structured differently and did not contain this particular error.

6.3 Verifying the "Correct Undo's" Property of the Bartok STM

Consider a Bartok STM execution ξ^{Bk} and a transaction `tx` that is rolled back in ξ^{Bk}. Since all executions of $undoLLE$ within `tx` for all objects `o` in the write set of `tx` are contained in the write span of `o`, no other object performs a write access to `o` or `o`'s STM metadata during the lifetime of `tx`. We will transform ξ^{Bk} to another partial-order equivalent Bartok STM execution $\xi^{\widetilde{Bk}}$ in which all actions implementing the transaction rollback appear consecutively. Because of abstraction (ii), $undoLLE$ actions modifying an object `o` commute with read actions that fall in the write span of `tx` because these transactions that these read actions belong to will eventually fail. Then, all of these read actions can be moved to after the last action by transaction `tx`.

The abstraction and equivalence argument above allows us to only consider Bartok STM executions in which transactions that are rolled back perform the entire roll-back of the log uninterrupted by other threads, in a sequential manner.

We now state and prove the correct undo last log entry property for Bartok STM executions (see Fig. 6). The following sequence of Bartok STM actions, when executed without interference by other threads, leaves the STM and program state unchanged: perform a field update, add the corresponding entry to the tail of the log, undo the field update, remove the entry from the tail of the log. This property is checked in a straightforward manner by the code for `CheckUndoLogEntry()` in Fig. 5. Repeated applications of this property enable us to prove that the state of the transaction manager `Tx` and all objects in the write set of `Tx` are the same right before `Tx` starts in ξ^{Bk} and right after `Tx` performs the first `CloseSTMWord`. This valid Bartok undo's property in a straightforward manner implies the correct undo's property for OTFJ programs.

Every write performed by a transaction is entered into the "update log". The update log is a linked list that stores the updates in chronological order. Each log entry corresponds to an object and its field and stores the old and new values of the field. When a transaction is being rolled back, the log is traversed from the tail toward the head. For each entry, the stored old value of the field is written into the object. Since the exclusive writes property ensures that this operation is carried out uninterfered by other threads, the proof obligation becomes a sequential one. The assertion checked in Figure 6 implies that adding a field update to the tail of the update log and then undoing it leaves all interesting program and STM state unchanged. By applying this lemma inductively for an arbitrary-length update log, we infer that the state of an object `o` modified by this transaction `tx` before the transaction starts and right after `tx` releases control of `o` are the same.