2009 Third IEEE International Conference on Secure Software Integration and Reliability Improvement

On the Verification of Strong Atomicity in Programs using STM

Yong Li[1,2] Yu Zhang[1,2] Yiyun Chen[1,2] Ming Fu[1,2]
[1]*Department of Computer Science & Technology*
University of Science & Technology of China
Hefei, 230027,China
{*liyong, fuming*}*@mail.ustc.edu.cn*
[2]*Software Security Laboratory*
Suzhou Institute for Advanced Study, USTC
SuZhou, 215123, China
{*yuzhang, yiyun*}*@ustc.edu.cn*

***Abstract*—Transactional memory (TM) provides an easy-using and high-performance parallel programming model for multicore systems. It simplifies parallel programming by supporting that transactions appear to execute atomically and in isolation. Despite the large amount of recent works on various TM implementations, very little has been devoted to precisely guarantee that these implementations have implemented the atomicity and isolation properties. In previous work we have proposed a framework on the correctness of STM programs by formally certifying the shared memory invariant at assembly level. Now the framework is extended and we certify the strong atomicity property of programs using STM in this paper. In particular, we formalize the strong atomicity as the shared memory consistence of states in our model and use a notion of "local guarantee" to check the shared memory consistence for verification. Our work provides a foundation for certifying realistic transactional programs and makes an important advance toward generating proof-carrying code.**

***Keywords*-transactional memory; strong atomicity; proof-carrying code;**

I. Introduction

The advent of multicore processors has brought concurrency into mainstream applications, however, it also brings great challenges to programmers for the concurrency management. They traditionally used locks to enforce synchronized concurrent memory accesses. However, locks are well-known software engineering issues that make parallel programming too complicated and may lead to problems such as deadlock, priority inversion, or convoying. Transactional memory provides an alternative concurrency management model that avoids these pitfalls associated with locks and significantly eases parallel programming.

TM simplifies concurrency management by supporting parallel tasks that appear to execute atomically and in isolation. In TM system, behavior of transactions must satisfy the following properties: a) *atomicity*: either the whole transaction executes or none of it; b) *isolation*: partial memory updates are not visible to other transactions [1].

Transactions should be atomic and isolated with respect to each other, but their relationship to non-transactional code is less clear. There are two models of atomicity semantics: *strong atomicity semantics* and *weak atomicity semantics*. Strong atomicity semantics is a semantics in which transactions are atomic and strongly isolated with respect both to other transactional and non-transactional codes. In essence, strong atomicity implicity treats each instruction appearing outside transactions as its own singleton transaction. Weak atomicity semantics is a semantics in which transactions are atomic and weakly isolated only with respect to other transactions.

There have been several proposals for supporting transactional memory either by hardware [1], [2], [3] (HTM) or software [5], [6], [7], [8] (STM) with both approaches having their pros and cons. HTM provides a significant performance advantage, and enforces the strong atomicity semantics. However, HTM requires complicated hardware support and usually restricts the size of transactional code blocks. STM can easily support unbounded transactions, nested transactions with partial rollbacks, and conditional signaling [9]. However, STM lacks for performance so prior STM systems mostly implement weak atomicity, in which non-transactional codes go directly to shared memory and bypass the STM access protocols.

In STM programming, programmers mark the regions that should be executed atomically at high level; then the STM compiler packages operations in the region with libraries and translates to the low level. At the low level, not only the well synchronized concurrent shared memory accesses, but also the strong atomicity semantics must be well enforced in program behaviors. In previous work [13] we have proposed a framework for certifying concurrent programs using STM at assembly level. In the framework we have modeled a software implementation of TM system based on storable locks and certified the shared memory invariant. It mainly focuses on the correct shared memory accesses in and out of transactions. However, the other significant properties such as the strong atomicity behaviors are entirely overlooked. In this paper, we attempt to extend our previous framework by formally certifying the strong atomicity semantics in concurrent programs using STM system. The atomicity and strong isolation properties are formalized as the consistence of visible shared memory between the beginning and the time after rolling back of a transaction in the extended framework. The consistence is certified by using a local guarantee that describes valid shared memory transitions. The local guarantees for the beginning and the time after rolling back of a transaction are constant while the others

978-0-7695-3758-0/09 $25.00 © 2009 IEEE
DOI 10.1109/SSIRI.2009.8

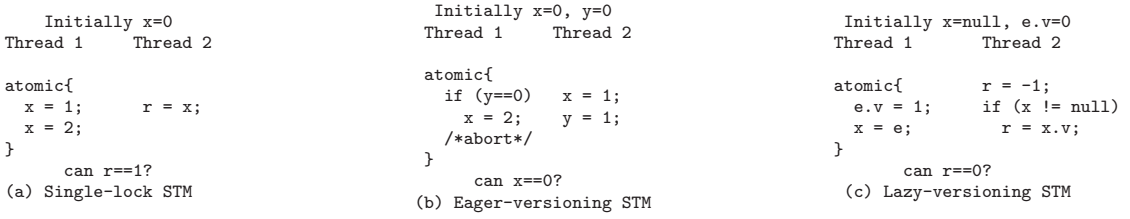

Figure 1. Anomalies on various STM implementations

between them can be generated automatically.

And just as most Foundational-PCC (FPCC) [19] systems, we provide these certifications just at the assembly level in the work. However, codes and specifications at high levels can be translated to our level by a certifying compiler, even taking the strong atomicity into consideration. In the translation, the certifying compiler initializes the local guarantees for the beginning and the time after rolling back of each transaction, then the others are generated automatically. And due to the assembly level, the breakdown of strong atomicity in objective codes caused by translation errors in compilation can be rejected and the compiler can be excluded from the trusted computing base. This paper makes the following novel contributions:

- It presents a train of thought that translates the strong atomicity in high level codes into the shared memory consistence at assembly level by a certifying compiler. Then we introduce local guarantees to certify the consistence in the extended framework.
- It presents a new design of the program logic after the added local guarantees in program specification. The soundness of the extended framework has also been proved.
- Our verification is fully mechanized in the Coq proof assistant [14]. We follow the FPCC style to give the soundness proof of the whole verification framework.

The rest of the paper is organized as follows. In section II we characterize weak atomicity behaviors and analyze the essence. In section III we summarize our previous framework on formal reasoning about transactional programs. Then the framework is extended with the atomicity proof in section IV, presenting the strong atomicity is indeed enforced in programs. We use an example to illustrate the verification in section V. Finally, we discuss related work and conclude in section VI.

II. Weak atomicity behaviors

Strong atomicity provides a simple and intuitive view of transactional atomicity, which may be more difficult to implement efficiently. In contrast, weak atomicity provides a less intuitive model, but it may be easier to implement efficiently so prior STM systems mostly implement it. Under weak atomicity, there are many unexpected behaviors between transactional and non-transactional codes accessing the same shared data with at least one write access. In Figure 1 we present three examples of isolation violation for various STM implementations respectively.

- Single-lock STM: A transaction holds the global lock before execution to prevent interleaves from other transactions. Figure 1(a) illustrates an intermediate dirty read where thread 2 may read the intermediate value 1 from x. This intermediate dirty read may also exist in the eager-versioning STM, but can not occur in the lazy-versioning STM. In the single-lock STM system, there exists a global lock used for synchronization between transactions. However, non-transactional codes don't hold the global lock before accessing shared data, so the anomaly happens.
- Eager-versioning STM [6], [7]: A transaction executes as though no other transaction is interleaved. It writes new values in place and logs old values, transaction can retry, undo changes when commit failed. Figure 1(b) illustrates a lost update where a non-transactional update is lost due to a transaction's rollback. Assume thread 1 executes first and updates x to 2, then thread 2 executes and updates both x and y. Next transaction in thread 1 finds the happened conflict due to the modification of y by thread 2. It rolls back and restores x's value back to 0 so thread 2's update of x is lost.
- Lazy-versioning STM [8], [9]: Like eager-versioning STM, but it leaves old values in place and buffers new values. The new values will be updated on commit. The two anomalies above can not occur here, but it also has anomalies. Figure 1(c) illustrates overlapped writes where the commit operation isn't atomic to non-transactional codes. Thread 1 initializes a field in the object el and then publishes the object by writing it to shared variable x. However, a lazy-versioning STM copies buffered values to the shared memory in no particular order. If thread 1 copies new value of x to memory first, then thread 2 executes and sees the published object in x but can not see the initialized value of its field. So r==0 holds in that case.

From these three examples we see that weak atomicity may lead to unexpected behaviors so it is important to implement the strong atomicity semantics in STMs. These anomalies of various STM systems we present above are all between transactional and non-transactional codes for the

lack of synchronization of shared data. To enforce the isolation property between transactional and non-transactional codes, there must be some mechanism which prevents non-transactional accessing when a shared memory update has arisen but has not been commit in transaction. These mechanisms are bypassed in weak atomicity semantics due to the performance while are required in the strong atomicity semantics. Therefore, with strong atomicity semantics non-transactional codes cannot observe the intermediate state of transactions so these anomalies will not occur. Shpeisman *et al.* [18] implement the strong atomicity by adding read and write barriers in non-transactional codes. In our framework described in next section, we modeled a software TM system implementation based on storable locks. It uses storable locks for synchronization of shared data both in and out of transactions to implement the strong atomicity semantics. And in section IV we use program logic for the verification of atomicity property to ensure that programs have indeed enforced the strong atomicity.

III. The STM implementation

In this section, we review our previous framework and some key techniques used in the framework. It is at assembly level for certifying safety properties in STM programs. The whole framework contains two parts: abstract machine model and program logic. Section III-A presents a simplified STM system based on storable locks. Section III-B discusses the program logic for proving the shared memory invariant. The detailed presentation of the framework is in our latest paper [13].

A. Abstract machine

(*World*)	$\mathbb{W}$	::=	$(\mathbb{C},\mathbb{M},[\mathbb{T}_1,\ldots,\mathbb{T}_n])$
(*Thread*)	$\mathbb{T}$	::=	$(\mathbb{R},\mathtt{pc},\mathbb{X})$
(*ThreadState*)	$\mathbb{S}$	::=	$(\mathbb{M},\mathbb{R},\mathtt{pc},\mathbb{X})$
(*CodeHeap*)	$\mathbb{C}$	::=	$\{\mathtt{f}\rightsquigarrow\mathfrak{i}\}^*$
(*Memory*)	$\mathbb{M},\mathbb{H}_r,\mathbb{H}_w$	$\in$	*Address* $\rightharpoonup$ *Word*
(*RegFile*)	$\mathbb{R}$	$\in$	*Register* $\rightarrow$ *Word*
(*Register*)	$\mathtt{r}$	::=	$\mathtt{r}_0 \mid \ldots \mid \mathtt{r}_{31}$
(*Address*)	$\mathtt{f},\mathtt{l},\mathtt{pc}$	::=	i (*nat nums*)
(*Word*)	$\mathtt{w}$	::=	i (*nat nums*)
(*Status*)	$\mathbb{U}$	::=	act | cmt | abt
(*Bstate*)	$\mathbb{B}$	::=	$(\mathbb{R},\mathtt{pc})$
(*Xstate*)	$\mathbb{X}$	::=	$\varepsilon \mid (\mathbb{H}_r,\mathbb{H}_w,\mathbb{B},\mathbb{U})$
(*Instr*)	$\mathfrak{i}$	::=	$\ldots \mid \mathsf{cas}_t\ \mathtt{r}_d,\mathtt{r}_t,\mathtt{w}(\mathtt{r}_s) \mid \mathsf{lw}\ \mathtt{r}_d,\mathtt{w}(\mathtt{r}_s)$ $\mid \mathsf{sw}\ \mathtt{r}_d,\mathtt{w}(\mathtt{r}_s) \mid \mathsf{lw}_t\ \mathtt{r}_d,\mathtt{w}(\mathtt{r}_s) \mid \mathsf{sw}_t\ \mathtt{r}_d,\mathtt{w}(\mathtt{r}_s)$ $\mid$ begin | validate $\mathtt{r}_d$ | commit
(*InstrSeq*)	$\mathbb{I}$	::=	$\mathfrak{i};\mathbb{I} \mid \mathsf{j}\ \mathtt{f} \mid$ rollback
(*ProgSpec*)	ϕ	::=	$(\mathtt{m},[\psi_1,\ldots,\psi_n])$
(*CdHpSpec*)	ψ	::=	$\{\mathtt{l}\rightsquigarrow\mathtt{a}\}^*$
(*StatePred*)	$\mathtt{a}$	$\in$	*ThreadState* $\rightarrow$ *Prop*
(*MemPred*)	$\mathtt{m}$	$\in$	*Memory* $\rightarrow$ *Prop*

Figure 2. Syntax and Verification Constructs

The syntax and verification constructs of the machine is presented in Figure 2. A complete world consists of a code heap $\mathbb{C}$, a shared memory $\mathbb{M}$ and numbers of threads made up of the register file $\mathbb{R}$, the program counter pc and transactional mechanism $\mathbb{X}$. The transactional mechanism is made up of a read set $\mathbb{H}_r$, a write set $\mathbb{H}_w$, a backup file $\mathbb{B}$ and a transactional status $\mathbb{U}$ when in transaction. We introduce three kinds of status act, cmt and abt for transactions, denoting the various phases during the execution of a transaction. In status act, transaction reads data from the shared memory, does some computation and then buffers write attempts. Next if there are no conflicts, the status of the transaction turns to cmt. Then the write buffers commit and the transaction end. Otherwise the status of the transaction turns to abt and the transaction rolls back. In this model we use storable locks for synchronization of shared memory among threads. The storable lock is implemented by a non-reentrant spin-lock mutex which reserves a word in memory to flag whether the mutex is locked or not. It uses a cas_t instruction for locking while the normal write instruction sw for unlocking. The locking behaviors are limited in transactions. Locking instruction cas_t can only be executed in status act while the unlocking instruction sw can be anywhere. Once locking failed in transaction, the status turns to abt and the transaction rolls back immediately.

The operational semantics of this model is presented in Figure 3. It is a partial function $\mathsf{Next}_{(\mathtt{pc},\mathfrak{i})}$ that computes the next thread state of $\mathbb{S}$ performed by a given instruction $\mathfrak{i}$ when executed from location pc. The macro $\mathsf{Npc}_{(\mathtt{pc},\mathfrak{i})}$ showed in Figure 4 is a total function that computes the pc of next instruction to be executed after the current instruction $\mathfrak{i}$ is completed.

if $\mathfrak{i}=$	then $\mathsf{Npc}_{(\mathtt{pc},\mathfrak{i})}(\mathbb{M},\mathbb{R},\mathtt{pc},\mathbb{X})=$
beq $\mathtt{r}_s,\mathtt{r}_t,\mathtt{f}$	$\begin{cases}\mathtt{f} & \text{if } \mathbb{R}(\mathtt{r}_s)=\mathbb{R}(\mathtt{r}_t)\\ \mathtt{pc}+1 & \text{if } \mathbb{R}(\mathtt{r}_s)\neq\mathbb{R}(\mathtt{r}_t)\end{cases}$
bne $\mathtt{r}_s,\mathtt{r}_t,\mathtt{f}$	$\begin{cases}\mathtt{pc}+1 & \text{if } \mathbb{R}(\mathtt{r}_s)=\mathbb{R}(\mathtt{r}_t)\\ \mathtt{f} & \text{if } \mathbb{R}(\mathtt{r}_s)\neq\mathbb{R}(\mathtt{r}_t)\end{cases}$
j f	f
jr $\mathtt{r}_s$	$\mathbb{R}(\mathtt{r}_s)$
rollback	$\mathbb{X}.\mathbb{B}.\mathtt{pc}$
Others	$\mathtt{pc}+1$

Figure 4. Auxiliary NextPC Macro

B. Program logic

We equip this machine with a construct ϕ (*Program Specification*) for expressing user-defined requirements. A program specification is made up of a global invariant which defines a criterion of the shared memory, and several code heap specifications for threads. The program logic is a combination of *concurrent separation logic* and *permission accounting in separation logic*.

Concurrent separation logic (CSL) [10] is a well known logic that provides a simple but powerful technique for reasoning about shared memory concurrent programs. In

if $\iota =$	then $\mathsf{Next}_{(\mathtt{pc},\iota)}(\mathbb{M},\mathbb{R},\mathtt{pc},\mathbb{X}) =$	
lw $\mathtt{r}_d,\mathtt{w}(\mathtt{r}_s)$	$(\mathbb{M},\mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{M}(\mathbb{R}(\mathtt{r}_s)+\mathtt{w})\},\mathtt{pc}+1,\mathbb{X})$	where $\mathbb{R}(\mathtt{r}_s)+\mathtt{w} \in \mathsf{dom}(\mathbb{M})$
sw $\mathtt{r}_d,\mathtt{w}(\mathtt{r}_s)$	$(\mathbb{M}\{\mathbb{R}(\mathtt{r}_s)+\mathtt{w} \rightsquigarrow \mathbb{R}(\mathtt{r}_d)\},\mathbb{R},\mathtt{pc}+1,\mathbb{X})$	where $\mathbb{R}(\mathtt{r}_s)+\mathtt{w} \in \mathsf{dom}(\mathbb{M})$
beq $\mathtt{r}_s,\mathtt{r}_t,\mathtt{f}$	$(\mathbb{M},\mathbb{R},\mathtt{f},\mathbb{X})$ $(\mathbb{M},\mathbb{R},\mathtt{pc}+1,\mathbb{X})$	if $\mathbb{R}(\mathtt{r}_s)=\mathbb{R}(\mathtt{r}_t)$ if $\mathbb{R}(\mathtt{r}_s)\neq\mathbb{R}(\mathtt{r}_t)$
bne $\mathtt{r}_s,\mathtt{r}_t,\mathtt{f}$	$(\mathbb{M},\mathbb{R},\mathtt{pc}+1,\mathbb{X})$ $(\mathbb{M},\mathbb{R},\mathtt{f},\mathbb{X})$	if $\mathbb{R}(\mathtt{r}_s)=\mathbb{R}(\mathtt{r}_t)$ if $\mathbb{R}(\mathtt{r}_s)\neq\mathbb{R}(\mathtt{r}_t)$
j f	$(\mathbb{M},\mathbb{R},\mathtt{f},\mathbb{X})$	
begin	$(\mathbb{M},\mathbb{R},\mathtt{pc}+1,(\emptyset,\emptyset,(\mathbb{R},\mathtt{pc}),\mathsf{act}))$	where $\mathbb{X}=\varepsilon$
validate $\mathtt{r}_d$	$(\mathbb{M},\mathbb{R}\{\mathtt{r}_d \rightsquigarrow 1\},\mathtt{pc}+1,(\mathbb{H}_r,\mathbb{H}_w,\mathbb{B},\mathsf{cmt}))$ $(\mathbb{M},\mathbb{R}\{\mathtt{r}_d \rightsquigarrow 0\},\mathtt{pc}+1,(\mathbb{H}_r,\mathbb{H}_w,\mathbb{B},\mathsf{abt}))$	if $\mathbb{X}.\mathbb{H}_r \subseteq \mathbb{M} \wedge \mathbb{X}.\mathbb{U}=\mathsf{act}$ if $\mathbb{X}.\mathbb{H}_r \not\subseteq \mathbb{M} \wedge \mathbb{X}.\mathbb{U}=\mathsf{act}$
commit	$(\mathbb{M},\mathbb{R},\mathtt{pc}+1,\varepsilon)$	where $\mathbb{X}.\mathbb{U}=\mathsf{cmt}$
rollback	$(\mathbb{M},\mathbb{R}',\mathtt{pc}',\varepsilon)$	where $\mathbb{X}.\mathbb{U}=\mathsf{abt} \wedge \mathbb{X}.\mathbb{B}=(\mathbb{R}',\mathtt{pc}')$
lw_t $\mathtt{r}_d,\mathtt{w}(\mathtt{r}_s)$	$(\mathbb{M},\mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{H}_w(\mathbb{R}(\mathtt{r}_s)+\mathtt{w})\},\mathtt{pc}+1,\mathbb{X})$ $(\mathbb{M},\mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{H}_r(\mathbb{R}(\mathtt{r}_s)+\mathtt{w})\},\mathtt{pc}+1,\mathbb{X})$ $(\mathbb{M},\mathbb{R}\{\mathtt{r}_d \rightsquigarrow \mathbb{M}(\mathbb{R}(\mathtt{r}_s)+\mathtt{w})\},\mathtt{pc}+1,$ $(\mathbb{H}_r\{\mathbb{R}(\mathtt{r}_s)+\mathtt{w} \rightsquigarrow \mathbb{M}(\mathbb{R}(\mathtt{r}_s)+\mathtt{w})\},\mathbb{H}_w,\mathbb{B},\mathbb{U}))$	if $\mathbb{R}(\mathtt{r}_s)+\mathtt{w} \in \mathsf{dom}(\mathbb{X}.\mathbb{H}_w)$; if $\mathbb{R}(\mathtt{r}_s)+\mathtt{w} \in \mathsf{dom}(\mathbb{X}.\mathbb{H}_r)$; if $\mathbb{R}(\mathtt{r}_s)+\mathtt{w} \in \mathsf{dom}(\mathbb{M})$, where $\mathbb{X}\neq\varepsilon$
sw_t $\mathtt{r}_d,\mathtt{w}(\mathtt{r}_s)$	$(\mathbb{M},\mathbb{R},\mathtt{pc}+1,(\mathbb{H}_r,\mathbb{H}_w\{\mathbb{R}(\mathtt{r}_s)+\mathtt{w} \rightsquigarrow \mathbb{R}(\mathtt{r}_d)\},\mathbb{B},\mathbb{U}))$	where $\mathbb{X}\neq\varepsilon$
cas_t $\mathtt{r}_d,\mathtt{r}_t,\mathtt{w}(\mathtt{r}_s)$	$(\mathbb{M}\{\mathbb{R}(\mathtt{r}_s)+\mathtt{w} \rightsquigarrow \mathbb{R}(\mathtt{r}_t)\},\mathbb{R}\{\mathtt{r}_t \rightsquigarrow \mathbb{R}(\mathtt{r}_d)\},\mathtt{pc}+1,\mathbb{X})$ $(\mathbb{M},\mathbb{R}\{\mathtt{r}_t \rightsquigarrow \mathbb{M}(\mathbb{R}(\mathtt{r}_s)+\mathtt{w})\},\mathtt{pc}+1,(\mathbb{H}_r,\mathbb{H}_w,\mathbb{B},\mathsf{abt}))$	if $\mathbb{R}(\mathtt{r}_d)=\mathbb{M}(\mathbb{R}(\mathtt{r}_s)+\mathtt{w}) \wedge \mathbb{X}.\mathbb{U}=\mathsf{act}$; if $\mathbb{R}(\mathtt{r}_d)\neq\mathbb{M}(\mathbb{R}(\mathtt{r}_s)+\mathtt{w}) \wedge \mathbb{X}.\mathbb{U}=\mathsf{act}$; where $\mathbb{R}(\mathtt{r}_d)+\mathtt{w} \in \mathsf{dom}(\mathbb{M})$

Figure 3. Operational Semantics of the Machine

CSL, shared memory is partitioned and each part is protected by a unique lock. For each part of the partitions, an invariant is assigned to specify its well-formedness. When the lock is acquired, the thread takes advantage of mutual-exclusion provided by lock and treats the partial memory as private. Before releasing the lock, it must ensure that the part of memory is well-formed with regard to the corresponding invariant. With CSL, shared memory accesses must be put in conditional critical region to treat the part of memory as private, both in and out of transactions, then those anomalies in section II can not occur.

Permission Accounting in Separation Logic (PASL) [11] is a lightweight logical approach to race-free sharing of memory, based on the notion of permission to access. In PASL each cell in the shared memory associates with a permission set (zero and one in our model). A total permission (zero) can be split into two read-only permissions (one) as needed. On lock acquiring, one part with the read-only permission is moved to the thread's private memory while the other one is left in shared memory for other threads' speculative reads. When updating, it first combines the private memory with the shared memory to form a total permission, then updates both parts.

By the program logic that incorporates CSL and PASL, we can deal with speculative shared memory accesses in transaction. Our specification asserts the machine-level behavior of the program and is general enough for various safety requirements. Interested readers are referred to the previous work [13] for a complete modeling of the framework and a certified example.

IV. Atomicity proof

In this section, we extend our previous framework with the formal atomicity proof to enforce strong atomicity. In the previous work we mainly focus on the correct concurrent shared memory accesses and certify the shared memory invariant. However, the shared memory invariant is a well-formed characterization of the shared memory but lacks the ability for description of the relation between two different states. So it is impossible to certify the strong atomicity in our previous framework.

As we have mentioned, strong atomicity requires that transactions are atomic and strongly isolated. First, we formalize the atomicity property as the consistence of shared memory between thread states of the beginning and the time after rolling back of a transaction. Then aiming at the consistence, we introduce local guarantees in the program specification to describe valid shared memory transitions. The local guarantees for the beginning and the time after rolling back of each transaction are generated by the certifying compiler while the others can be generated automatically. In the atomicity proof, it implicitly requires to restore the modified data to previous state before the corresponding lock releasing in status act or abt, which is exactly the strong isolate property that we need.

By adding the local guarantee in the program specifications, the verification constructs and inference rules for program logic have to be extended too.

A. Inference rule

Verification constructs for program logic of our extended framework are introduced in Figure 5.

$$
\begin{array}{rrcl}
(\mathit{WorldSpec}) & \phi & ::= & (\mathtt{m},[\psi_1,\ldots,\psi_n]) \\
(\mathit{CdHpSpec}) & \psi & ::= & \{\mathtt{l} \rightsquigarrow (\mathtt{a},\mathtt{g})\}^* \\
(\mathit{StatePred}) & \mathtt{a} & \in & \mathit{ThreadState} \to \mathit{Prop} \\
(\mathit{Guarantee}) & \mathtt{g} & \in & \mathit{ThreadState} \to \mathit{Memory} \to \mathit{Prop} \\
(\mathit{MemPred}) & \mathtt{m} & \in & \mathit{Memory} \to \mathit{Prop} \\
(\mathit{Interp}_a) & [\![-]\!]_a & \in & \mathit{StatePred} \times \mathit{Guarantee} \to \mathit{StatePred} \\
(\mathit{Interp}_g) & [\![-]\!]_g & \in & \mathit{StatePred} \times \mathit{Guarantee} \to \mathit{Guarantee}
\end{array}
$$

Figure 5. Verification Constructs for Program Logic

The whole world specification ϕ contains a global shared memory invariant $\mathtt{m}$ and code heap specifications $\psi_1, \ldots, \psi_n$ for each thread. Compared with Figure 2, the code heap specification ψ here adds a guarantee $\mathtt{g}$ to each instruction sequence. The local guarantee $\mathtt{g}$, just as in CCAP [16] and CMAP [17], describes valid shared memory transitions – it is safe for the current transaction to roll back only after making a memory transition allowed by $\mathtt{g}$. Finally $[\![-]\!]_a$ and $[\![-]\!]_g$ take out the state predicate and guarantee from the pair $(\mathtt{a}, \mathtt{g})$ respectively.

The memory predicate $\mathtt{m}$ takes a style of PASL which is shown in Figure 6.

$$\begin{array}{rcl} \mathtt{m} & ::= & \mathtt{l} \overset{\mathtt{u}}{\longmapsto} \mathtt{v} \mid \mathsf{emp} \mid \mathtt{m}_1 * \mathtt{m}_2 \mid \mathtt{m}_1 \wedge \mathtt{m}_2 \mid \mathtt{m}_1 \vee \mathtt{m}_2 \\ & & \exists x.\mathtt{m} \mid \forall x.\mathtt{m} \end{array}$$

Figure 6. Permission Accounting in Separation Logic

$$\begin{array}{rcl} \mathtt{a} & ::= & \varepsilon \mid [\mathtt{m}] \mid [\mathtt{m}]_r \mid [\mathtt{m}]_w \mid [\mathtt{r}] = \mathtt{v} \mid [\mathtt{r}]_b = \mathtt{v} \\ & & \mid [\mathtt{pc}] = \mathtt{v} \mid [\mathtt{pc}]_b = \mathtt{v} \mid \mathbb{U} = \mathsf{act/cmt/abt} \\ & & \mid \mathtt{a}_1 \wedge \mathtt{a}_2 \mid \mathtt{a}_1 \vee \mathtt{a}_2 \mid \exists x.\ \mathtt{a} \mid \forall x.\ \mathtt{a} \end{array}$$

$$\begin{array}{rcl@{\qquad}rcl} \varepsilon & \overset{\text{def}}{=} & \lambda\mathbb{S}.\ \mathbb{S}.\mathbb{X} = \varepsilon & [\mathtt{r}] = \mathtt{v} & \overset{\text{def}}{=} & \lambda\mathbb{S}.\ \mathbb{S}.\mathbb{R}(\mathtt{r}) = \mathtt{v} \\ [\mathtt{m}] & \overset{\text{def}}{=} & \lambda\mathbb{S}.\ \mathtt{m}\ \mathbb{S}.\mathbb{M} & [\mathtt{r}]_b = \mathtt{v} & \overset{\text{def}}{=} & \lambda\mathbb{S}.\ \mathbb{S}.\mathbb{X}.\mathbb{B}.\mathbb{R}(\mathtt{r}) = \mathtt{v} \\ [\mathtt{m}]_r & \overset{\text{def}}{=} & \lambda\mathbb{S}.\ \mathtt{m}\ \mathbb{S}.\mathbb{X}.\mathbb{H}_r & [\mathtt{pc}]_b = \mathtt{v} & \overset{\text{def}}{=} & \lambda\mathbb{S}.\ \mathbb{S}.\mathbb{X}.\mathbb{B}.\mathtt{pc} = \mathtt{v} \\ [\mathtt{m}]_w & \overset{\text{def}}{=} & \lambda\mathbb{S}.\ \mathtt{m}\ \mathbb{S}.\mathbb{X}.\mathbb{H}_w & [\mathtt{pc}] = \mathtt{v} & \overset{\text{def}}{=} & \lambda\mathbb{S}.\ \mathbb{S}.\mathtt{pc} = \mathtt{v} \\ \exists x.\ \mathtt{a} & \overset{\text{def}}{=} & \lambda\mathbb{S}.\ \exists x.\ \mathtt{a}\ \mathbb{S} & \mathtt{a}_1 \wedge \mathtt{a}_2 & \overset{\text{def}}{=} & \lambda\mathbb{S}.\ \mathtt{a}_1\mathbb{S} \wedge \mathtt{a}_2\mathbb{S} \\ \forall x.\ \mathtt{a} & \overset{\text{def}}{=} & \lambda\mathbb{S}.\ \forall x.\ \mathtt{a}\ \mathbb{S} & \mathtt{a}_1 \vee \mathtt{a}_2 & \overset{\text{def}}{=} & \lambda\mathbb{S}.\ \mathtt{a}_1\mathbb{S} \vee \mathtt{a}_2\mathbb{S} \end{array}$$

$$\mathbb{U} = \mathsf{act/cmt/abt} \overset{\text{def}}{=} \lambda\mathbb{S}.\ \mathbb{S}.\mathbb{X}.\mathbb{U} = \mathsf{act/cmt/abt}$$

Figure 7. Assertion Language for Thread State

The assertion language for thread state is presented in Figure 7. To encode the specification and proofs, we take the use of Coq [14] and the underlying CiC [15] for mechanical verification. We encode the syntax using inductive definitions, and define the operational semantics and inference rules as a collection of relations.

We introduce some useful auxiliary definitions for program logic in Figure 8. And the macro $\mathsf{En}\ (\iota)$ for required transactional status of each instruction ι is presented in Figure 9. They are the same as ones in our previous paper.

$$\begin{array}{lcl} \mathtt{a} \Rightarrow \mathtt{a}' & \overset{\text{def}}{=} & \forall \mathbb{S}.\ \mathtt{a}\ \mathbb{S} \rightarrow \mathtt{a}'\ \mathbb{S} \\ \mathtt{Domeq}\ \mathbb{M}\mathbb{M}' & \overset{\text{def}}{=} & \forall \mathtt{l}.\ (\mathtt{l} \in \mathsf{dom}(\mathbb{M}) \wedge \mathtt{l} \in \mathsf{dom}(\mathbb{M}')) \vee \\ & & \quad (\mathtt{l} \notin \mathsf{dom}(\mathbb{M}) \wedge \mathtt{l} \notin \mathsf{dom}(\mathbb{M}')) \\ \mathtt{a} \triangleleft (\mathbb{R}, \mathtt{pc}, \mathbb{X}) & \overset{\text{def}}{=} & \lambda\mathbb{M}.\ \mathtt{a}\ (\mathbb{M}, \mathbb{R}, \mathtt{pc}, \mathbb{X}) \\ \mathtt{a} \circledast \mathtt{m} & \overset{\text{def}}{=} & \lambda(\mathbb{M}, \mathbb{R}, \mathtt{pc}, \mathbb{X}).\ (\mathtt{a} \triangleleft (\mathbb{R}, \mathtt{pc}, \mathbb{X}) * \mathtt{m})\ \mathbb{M} \end{array}$$

Figure 8. Auxiliary Definition for Program Logic

We use the following judgement forms to define the inference rules:

$$\begin{array}{ll} \phi, [(\mathtt{a}_1, \mathtt{g}_1), \ldots, (\mathtt{a}_n, \mathtt{g}_n)] \vdash \mathbb{W} & \text{(well-formed world)} \\ \psi, \mathtt{m} \vdash \mathbb{C} : \psi' & \text{(well-formed code heap)} \\ \psi, \mathtt{m} \vdash \{(\mathtt{a}, \mathtt{g})\}\mathtt{pc} : \mathbb{I} & \text{(well-formed instr. sequences)} \end{array}$$

if $\iota =$	then $\mathsf{En}\ (\iota) =$
$\mathsf{lw}_t\ \mathtt{r}_d, \mathtt{r}_s(\mathtt{w})$	$\lambda\mathbb{S}.\ \mathbb{S}.\mathbb{X} \neq \varepsilon$
$\mathsf{sw}_t\ \mathtt{r}_d, \mathtt{r}_s(\mathtt{w})$	$\lambda\mathbb{S}.\ \mathbb{S}.\mathbb{X} \neq \varepsilon$
$\mathsf{cas}_t\ \mathtt{r}_d, \mathtt{r}_t, \mathtt{r}_s(\mathtt{w})$	$\lambda\mathbb{S}.\ \mathbb{S}.\mathbb{X}.\mathbb{U} = \mathsf{act}$
begin	$\lambda\mathbb{S}.\ \mathbb{S}.\mathbb{X} = \varepsilon$
validate $\mathtt{r}_d$	$\lambda\mathbb{S}.\ \mathbb{S}.\mathbb{X}.\mathbb{U} = \mathsf{act}$
commit	$\lambda\mathbb{S}.\ \mathbb{S}.\mathbb{X}.\mathbb{U} = \mathsf{cmt}$
rollback	$\lambda\mathbb{S}.\ \mathbb{S}.\mathbb{X}.\mathbb{U} = \mathsf{abt}$
Others	True

Figure 9. Required Transactional Status for Instructions

The inference rules for the extended framework are presented in Figure 10. A world is well-formed with regard to a world specification ϕ and pairs of thread state predicates and guarantees $(\mathtt{a}_1, \mathtt{g}_1), \ldots, (\mathtt{a}_n, \mathtt{g}_n)$ for each thread when the following conditions hold:

- There exists a code heap specification ψ_i for each thread and a global invariant $\mathtt{m}$ in the world specification ϕ. For each thread, the code heap is well-formed regarding ψ_i and $\mathtt{m}$, moreover, the thread precondition pair $(\mathtt{a}_i, \mathtt{g}_i)$ is satisfied at the point of $\mathtt{pc}_i$.
- There is a $n+1$ parts partition of the shared memory $\mathbb{M}$, where $\mathbb{M}_s$ satisfies the global invariant $\mathtt{m}$ and $\mathbb{M}_1, \ldots, \mathbb{M}_n$ satisfy each thread state predicate $\mathtt{a}_i$ respectively.

A code heap is well-formed only if each instruction sequence in the code heap is well-formed.

Next, an instruction sequence is well-formed if it is composed of a single instruction ι and another instruction sequence $\mathbb{I}$ and both of them are well-formed (rule-INSQ). A well-formed instruction (rule-INSN) requires :

- Thread state predicate restrict as previous: instruction ι can execute for all thread states specified by the current thread state predicate $\mathtt{a}$ and the global invariant $\mathtt{m}$. Furthermore, the new modified thread state must satisfy the thread state predicate for the target address of instruction ι given by ψ and reestablish the global invariant $\mathtt{m}$.
- Added restrict on local guarantee: if the current transaction is in status act or abt and the modified thread state satisfies the guarantee of the target address, then the original state satisfies $\mathtt{g}$.

Here we left rules of instructions begin, commit and rollback alone. Rule-COMMIT checks the domain of private memory at current state with the beginning of transaction to enforce that all locks have been released before commit. Next rule-ROLLBACK records memory at the state of rolling back and goes back to check the consistence with memory of the beginning in rule-BEGIN.

Suppose the thread state transition sequence of the transaction(begin with begin and end with rollback) is $\mathbb{S}_0, \ldots, \mathbb{S}_n$. To show that the atomicity of the transaction is enforced through the guarantee $\mathtt{g}$, we enforce the following chain of

$\boxed{\phi,[(\mathsf{a}_1,\mathsf{g}_1),\ldots,(\mathsf{a}_n,\mathsf{g}_n)]\vdash\mathbb{W}}$ (Well-formed world)

$$\frac{\begin{array}{c}\phi=(\mathtt{m},[\psi_1,\ldots,\psi_n])\\ \mathbb{M}=\mathbb{M}_s\uplus\mathbb{M}_1\uplus\ldots\uplus\mathbb{M}_n\\ \mathtt{m}\ \mathbb{M}_s\quad \mathsf{a}_k(\mathbb{M}_k,\mathbb{R}_k,\mathsf{pc}_k,\mathbb{X}_k)\\ \psi_k,\mathtt{m}\vdash\mathbb{C}:\psi_k\quad \psi_k,\mathtt{m}\vdash\{(\mathsf{a}_k,\mathsf{g}_k)\}\mathsf{pc}_k:\mathbb{C}[\mathsf{pc}_k]\quad\text{for all } k\end{array}}{\phi,[(\mathsf{a}_1,\mathsf{g}_1),\ldots,(\mathsf{a}_n,\mathsf{g}_n)]\vdash(\mathbb{C},\mathbb{M},[(\mathbb{R}_1,\mathsf{pc}_1,\mathbb{X}_1),\ldots,(\mathbb{R}_n,\mathsf{pc}_n,\mathbb{X}_n)])}\ (\textsc{world})$$

$\boxed{\psi,\mathtt{m}\vdash\mathbb{C}:\psi'}$ (Well-formed code heap)

$$\frac{\forall(\mathsf{pc},(\mathsf{a},\mathsf{g}))\in\psi':\quad\psi,\mathtt{m}\vdash\{(\mathsf{a},\mathsf{g})\}\mathsf{pc}:\mathbb{C}[\mathsf{pc}]}{\psi,\mathtt{m}\vdash\mathbb{C}:\psi'}\ (\textsc{cdhp})$$

$\boxed{\psi,\mathtt{m}\vdash\{(\mathsf{a},\mathsf{g})\}\mathsf{pc}:\mathbb{I}}$ (Well-formed instr. sequences)

$$\frac{\psi,\mathtt{m}\vdash\{(\mathsf{a}',\mathsf{g}')\}\mathsf{pc}+1:\mathbb{I}\quad\psi\uplus\{\mathsf{pc}+1\rightsquigarrow(\mathsf{a}',\mathsf{g}')\},\mathtt{m}\vdash\{(\mathsf{a},\mathsf{g})\}\mathsf{pc}:\iota}{\psi,\mathtt{m}\vdash\{(\mathsf{a},\mathsf{g})\}\mathsf{pc}:\iota;\mathbb{I}}\ (\textsc{insq})$$

$$\frac{\begin{array}{c}\iota\notin\{\mathsf{begin},\mathsf{commit},\mathsf{rollback}\}\\ \mathsf{a}\circledast\mathtt{m}\Rightarrow(\lambda\mathbb{S}.\ [\![\psi(\mathsf{N}\mathsf{pc}_{(\mathsf{pc},\iota)}\mathbb{S})]\!]_a\circledast\mathtt{m})\ (\mathsf{Next}_{(\mathsf{pc},\iota)}\mathbb{S}))\wedge\mathsf{En}\ (\iota)\\ \forall\mathbb{S},\mathbb{M}.\ (\mathsf{a}\circledast\mathtt{m})\ \mathbb{S}\rightarrow\mathbb{S}.\mathbb{X}.\mathbb{U}=\mathsf{act}/\mathsf{abt}\rightarrow\\ [\![\psi(\mathsf{N}\mathsf{pc}_{(\mathsf{pc},\iota)}\mathbb{S})]\!]_g\ (\mathsf{Next}_{(\mathsf{pc},\iota)}\mathbb{S})\ \mathbb{M}\rightarrow\mathsf{g}\ \mathbb{S}\ \mathbb{M}\end{array}}{\psi,\mathtt{m}\vdash\{(\mathsf{a},\mathsf{g})\}\mathsf{pc}:\iota}\ (\textsc{insn})$$

$$\frac{\begin{array}{c}\mathsf{a}\Rightarrow(\lambda\mathbb{S}.\ [\![\psi(\mathsf{pc}+1)]\!]_a\ (\mathsf{Next}_{(\mathsf{pc},\iota)}\mathbb{S}))\wedge\mathsf{En}\ (\mathsf{begin})\\ \forall\mathbb{S},\mathbb{M}.\ [\![\psi(\mathsf{pc}+1)]\!]_g\ \mathbb{S}\ \mathbb{M}\rightarrow\mathbb{S}.\mathbb{M}=\mathbb{M}\end{array}}{\psi,\mathtt{m}\vdash\{(\mathsf{a},\mathsf{g})\}\mathsf{pc}:\mathsf{begin}}\ (\textsc{begin})$$

$$\frac{\begin{array}{c}\mathsf{a}\Rightarrow(\lambda\mathbb{S}.\ [\![\psi(\mathsf{pc}+1)]\!]_a\ (\mathsf{Next}_{(\mathsf{pc},\iota)}\mathbb{S}))\wedge\mathsf{En}\ (\mathsf{commit})\\ \forall\mathbb{S},\mathbb{S}'.\ \mathsf{a}\ \mathbb{S}\rightarrow[\![\psi(\mathbb{S}.\mathbb{X}.\mathbb{B}.\mathsf{pc})]\!]_a\ \mathbb{S}'\rightarrow\mathtt{Domeq}\ \mathbb{S}.\mathbb{M}\ \mathbb{S}'.\mathbb{M}\end{array}}{\psi,\mathtt{m}\vdash\{(\mathsf{a},\mathsf{g})\}\mathsf{pc}:\mathsf{commit}}\ (\textsc{commit})$$

$$\frac{\begin{array}{c}\mathsf{a}\Rightarrow(\lambda\mathbb{S}.\ [\![\psi(\mathsf{N}\mathsf{pc}_{(\mathsf{pc},\iota)}\mathbb{S})]\!]_a\ (\mathsf{Next}_{(\mathsf{pc},\iota)}\mathbb{S}))\wedge\mathsf{En}\ (\mathsf{rollback})\\ \forall\mathbb{S}.\ (\mathsf{a}\circledast\mathtt{m})\ \mathbb{S}\rightarrow\mathsf{g}\ \mathbb{S}\ \mathbb{S}.\mathbb{M}\end{array}}{\psi,\mathtt{m}\vdash\{(\mathsf{a},\mathsf{g})\}\mathsf{pc}:\mathsf{rollback}}\ (\textsc{rollback})$$

Figure 10. Inference Rules

implication relations:

$$\mathsf{g}_n\ \mathbb{S}_n.\mathbb{M}\rightarrow\mathsf{g}_{n-1}\ \mathbb{S}_{n-1}\ \mathbb{S}_n.\mathbb{M}\rightarrow\ldots\rightarrow\mathsf{g}_1\ \mathbb{S}_1\ \mathbb{S}_n.\mathbb{M}\rightarrow\mathsf{g}_0\ \mathbb{S}_0\ \mathbb{S}_n.\mathbb{M}$$

where each g_i is the intermediate specification used at each verification step. Each arrow on the chain is enforced by rule-INSN. The head of the chain is enforced by rule-ROLLBACK while the end of the chain is enforced by rule-BEGIN, therefore we can finally reach the conclusion of $\mathbb{S}_0.\mathbb{M}=\mathbb{S}_n.\mathbb{M}$, which is the atomicity property.

Next the strong isolation property is also implied in the atomicity proof. Suppose a thread state transition $\mathbb{S}_i\rightarrow\mathbb{S}_{i+1}$ is created by an instruction ι, which is the instruction sw for unlocking. Then the domain of memories in $\mathbb{S}_i$ and $\mathbb{S}_{i+1}$ are not equal due to the resource releasing. However, the local guarantee g_{i+1} must be satisfied in both thread states. So it can not mention the partial memory which is in $\mathbb{S}_i$ but not in $\mathbb{S}_{i+1}$. Otherwise the assertion $\mathsf{a}_{i+1}\circledast\mathtt{m}$ and guarantee g_{i+1} will specify different memories and the atomicity proof is breakdown. In order to not be mentioned in the local guarantee, the memory block must be restored to the previous state when the corresponding lock is released. So that its partial updates are invisible to other threads and the isolation property is well enforced. We will show an example to illuminate the details in the next section.

CSL well synchronized shared memory accesses both in and out of transactions except for the speculative read lw_t. It seems to break the isolation of concurrent threads due to the speculative read's accessing shared memory without synchronization. However, the speculative read is only allowed in transactions and always followed by a conflict detection validate to check the validity of values. Instruction validate requires that locations that have been read speculatively must have been privatized. In this way, lw_t just delays memory privatization, not indeed pass it. So together with CSL and the atomicity proof, we can ensure that the strong atomicity is certified in our framework.

B. Soundness

The soundness of our extended framework inference rules with respect to the operational semantics for the machine is established following the syntactic approach of proving type soundness [20]. From the "progress" and "preservation" lemmas, we can guarantee that given a well-formed world under compatible assumptions, the current instruction sequence will be able to execute without getting "stuck". Furthermore, any safety property derivable from the global invariant will hold throughout the execution and the strong atomicity is well enforced. We define $\mathbb{W}\mapsto^n\mathbb{W}'$ as the relation of n-step ($n\geq 0$) world transitions. The soundness of the framework is formally stated as Theorem 4.3.

Lemma 4.1 (Progress)
: For any $\mathbb{W}=(\mathbb{C},\mathbb{M},[\mathbb{T}_1,\ldots,\mathbb{T}_n])$, if $\phi,[(\mathsf{a}_1,\mathsf{g}_1),\ldots,(\mathsf{a}_n,\mathsf{g}_n)]\vdash\mathbb{W}$, then for any thread $\mathbb{T}_i$, there exists $\mathbb{M}',\mathbb{T}_i'$, such that $(\mathbb{M},\mathbb{T}_i)\hookrightarrow(\mathbb{M}',\mathbb{T}_i')$.

Lemma 4.2 (Preservation)
: If $\phi,[(\mathsf{a}_1,\mathsf{g}_1),\ldots,(\mathsf{a}_n,\mathsf{g}_n)]\vdash\mathbb{W}$, and $\mathbb{W}\mapsto\mathbb{W}'$, then exists $(\mathsf{a}_1',\mathsf{g}_1'),\ldots,(\mathsf{a}_n',\mathsf{g}_n')$, such that $\phi,[(\mathsf{a}_1',\mathsf{g}_1'),\ldots,(\mathsf{a}_n',\mathsf{g}_n')]\vdash\mathbb{W}'$

Theorem 4.3 (Soundness)
: If $\phi,[(\mathsf{a}_1,\mathsf{g}_1),\ldots,(\mathsf{a}_n,\mathsf{g}_n)]\vdash\mathbb{W}$, then for any n $\geq$ 0, there exists a world $\mathbb{W}'$ and $(\mathsf{a}_1',\mathsf{g}_1'),\ldots,(\mathsf{a}_n',\mathsf{g}_n')$ such that $\mathbb{W}\mapsto^n\mathbb{W}'$ and $\phi,[(\mathsf{a}_1',\mathsf{g}_1'),\ldots,(\mathsf{a}_n',\mathsf{g}_n')]\vdash\mathbb{W}'$.

We have implemented the complete framework [21] including the proofs for these two lemmas and the soundness theorem in the Coq proof assistant so we are confident that the framework is indeed sound.

V. Example

Our framework is a realization of established verification techniques at the assembly level for concurrent programs. In this section, we give an example to demonstrate the mechanized verification of strong atomicity property for concurrent assembly code using our machine model.

A simple example of Fibonacci program is presented in Figure 11, which is the concurrent code that computes the next element of a Fibonacci sequence. The routine computes

the Fibonacci number by storing the last two numbers of the sequence into internal variables prev and curr. The variables prev and curr are shared between threads so it needs synchronization for access which is hidden in the CommitTransaction( ).

```
int fib( ){
    do{
        tx = StartTransaction();
        val_1 = StmRead(tx, &prev);
        val_2 = StmRead(tx, &curr);
        temp = val_1 + val_2;
        StmWrite(tx, &prev, val_2);
        StmWrite(tx, &curr, temp);
    }while(!CommitTransaction(tx));
}
```

Figure 11. Fibonacci Program

The assembly code for routine fib is presented in Figure 12. In the code it use a storable lock mutex for synchronization which relates to the inline synchronization in CommitTransaction( ). Together with the assembly code, we also present the set of precondition pairs and the shared memory invariant of the program Fibonacci for verification in Figure 12.

The shared memory invariant $\mathtt{m}$ and assertions $\mathtt{a}_i$ are the same as our previous paper [13]. Besides, we introduce guarantees $\mathtt{g}_0, \mathtt{g}_1, \mathtt{g}_2$ for the atomicity proof. Just as mentioned in section IV, it records the shared memory before the time of rolling back in the local guarantee and then goes backwards to check the consistence at the beginning. $\mathtt{g}_0$ is a constant truth predicate which is usually used in status cmt or outside transactions. $\mathtt{g}_1$ states that consistence is satisfied, it is safe now to roll back the transaction. Next instruction cas_t acquires the storable lock mutex. If succeed, cell mutex of the shared memory is modified and the local guarantee turns to $\mathtt{g}_2$ which states that cell mutex must restore to zero before rolling back. Then the lock is released at label ulk, the local guarantee turns back to $\mathtt{g}_1$ again. Finally the control transfers to rollback, it initializes the parameter $\mathbb{M}$ in the local guarantee with the shared memory at current state and passes it back for checking. After the whole verification, we can confirm that atomicity is indeed enforced.

The property strong isolation of transactions is also implied in the atomicity proof. In the atomicity proof it requires that the memory block must be restored to the previous state before its corresponding lock is released in rolling back, so the partial updates is not visible to other threads. Here we modify the pre-example to show the breakdown of strong isolation property due to the violation in atomicity proof when the memory block have not been restored to initial state before unlocking.

In Figure 13 we add an instruction to modify cell curr but do not restore it to previous value before releasing mutex. After the modification, the local guarantee changes to $\mathtt{g}_3$ which denotes that it is safe to roll back only after

$$
\begin{aligned}
\mathtt{m} &\triangleq (\exists n.\ \mathsf{mutex} \longmapsto 0 * \mathsf{prev} \longmapsto \mathsf{fib}(n) * \mathsf{curr} \longmapsto \mathsf{fib}(n+1)) \vee \\
&\quad (\exists n, n'.\ \mathsf{mutex} \longmapsto 1 * \mathsf{prev} \overset{1}{\longmapsto} n * \mathsf{curr} \overset{1}{\longmapsto} n') \\
\mathtt{a}_0 &\triangleq [\mathsf{emp}] \wedge \varepsilon \\
\mathtt{a}_1 &\triangleq [\mathsf{emp}] \wedge [\mathsf{emp}]_r \wedge [\mathsf{emp}]_w \wedge [\mathsf{pc}]_b = \mathsf{fib} \wedge \mathbb{U} = \mathsf{act} \\
\mathtt{a}_2 &\triangleq [\mathsf{emp}] \wedge [\mathsf{t}_1] = 1 \wedge \exists v, v'. [\mathsf{curr} \mapsto v * \mathsf{prev} \mapsto v']_r \\
&\quad \wedge\ [\mathsf{curr} \mapsto v + v' * \mathsf{prev} \mapsto v]_w \wedge [\mathsf{pc}]_b = \mathsf{fib} \wedge\ \mathbb{U} = \mathsf{act} \\
\mathtt{a}_3 &\triangleq [\exists n.\ \mathsf{curr} \overset{1}{\longmapsto} \mathsf{fib}(n+1) * \mathsf{prev} \overset{1}{\longmapsto} \mathsf{fib}(n)] \\
&\quad \wedge\ \exists v, v'. [\mathsf{curr} \mapsto v * \mathsf{prev} \mapsto v']_r \wedge\ [\mathsf{curr} \mapsto v + v' * \mathsf{prev} \mapsto v]_w \\
&\quad \wedge\ [\mathsf{pc}]_b = \mathsf{fib} \wedge \mathbb{U} = \mathsf{act} \\
\mathtt{a}_4 &\triangleq \exists n.\ [\mathsf{curr} \overset{1}{\longmapsto} \mathsf{fib}(n+1) * \mathsf{prev} \overset{1}{\longmapsto} \mathsf{fib}(n)] \wedge [\mathsf{emp}]_r \\
&\quad \wedge\ [\mathsf{curr} \mapsto \mathsf{fib}(n+2) * \mathsf{prev} \mapsto \mathsf{fib}(n+1)]_w \wedge\ [\mathsf{pc}]_b = \mathsf{fib} \wedge \mathbb{U} = \mathsf{cmt} \\
\mathtt{a}_5 &\triangleq \exists n.\ [\mathsf{curr} \overset{1}{\longmapsto} \mathsf{fib}(n+2) * \mathsf{prev} \overset{1}{\longmapsto} \mathsf{fib}(n+1)] \wedge\ [\mathsf{emp}]_r \\
&\quad \wedge\ [\mathsf{curr} \mapsto \mathsf{fib}(n+2) * \mathsf{prev} \mapsto \mathsf{fib}(n+1)]_w \wedge [\mathsf{pc}]_b = \mathsf{fib} \wedge \mathbb{U} = \mathsf{cmt} \\
\mathtt{a}_6 &\triangleq [\exists n.\ \mathsf{curr} \overset{1}{\longmapsto} \mathsf{fib}(n+1) * \mathsf{prev} \overset{1}{\longmapsto} \mathsf{fib}(n)] \wedge [\mathsf{emp}]_r \\
&\quad \wedge\ [\exists v, v'. \mathsf{curr} \mapsto v + v' * \mathsf{prev} \mapsto v]_w \wedge [\mathsf{pc}]_b = \mathsf{fib} \wedge\ \mathbb{U} = \mathsf{abt} \\
\mathtt{a}_7 &\triangleq [\mathsf{emp}] \wedge [\mathsf{pc}]_b = \mathsf{fib} \wedge \mathbb{U} = \mathsf{abt} \\
&\quad \wedge\ ([\mathsf{emp}]_r \wedge [\mathsf{emp}]_w \vee\ \exists v, v'. [\mathsf{curr} \mapsto v * \mathsf{prev} \mapsto v']_r \\
&\quad \wedge\ [\mathsf{curr} \mapsto v + v' * \mathsf{prev} \mapsto v]_w) \\
\mathtt{g}_0 &\triangleq \lambda \mathbb{S}, \mathbb{M}.\ \mathsf{TRUE} \\
\mathtt{g}_1 &\triangleq \lambda \mathbb{S}, \mathbb{M}.\ \mathbb{M} = \mathbb{S}.\mathbb{M} \\
\mathtt{g}_2 &\triangleq \lambda \mathbb{S}, \mathbb{M}.\ \mathbb{M} = \mathbb{S}.\mathbb{M}\{\mathsf{mutex} \rightsquigarrow 0\}
\end{aligned}
$$

```
1     .word          0
prev  .word          0
curr  .word          1

fib : -{(a0, g0)}
      begin
      -{(a1, g1)}
      lw_t           t1, curr(r0)
      lw_t           t2, prev(r0)
      addu           t2, t1, t2
      sw_t           t2, curr(r0)
      sw_t           t1, prev(r0)
      addiu          t1, r0, 1
      -{(a2, g1)}
      //   acquire locks
      cas_t          r0, t1, mutex(r0)
      bne            r0, t1, rb
      -{(a3, g2)}
      validate       t2
      beq            r0, t2, ulk
      -{(a4, g2)}
      lw_t           0_1, curr(r0)
      sw             t1, curr(r0)
      lw_t           t2, prev(r0)
      sw             t2, prev(r0)
      -{(a5, g0)}
      //   release locks
      sw             r0, mutex(r0)
      commit
      -{(a0, g0)}
      j              fib

      //   release locks
ulk : -{(a6, g2)}
      sw             r0, mutex(r0)
      j              rb
rb :  -{(a7, g1)}
      rollback
```

Figure 12. Assembly Code with Assertions of Fibonacci

restore cell mutex to zero and cell curr to $\mathsf{fib}(n+1)$. Next the storable lock mutex is released and the post condition is $(\mathtt{a}_7, \mathtt{g}_4)$. $\mathtt{g}_4$ denotes that cell curr must be restored to $\mathsf{fib}(n+1)$ before rolling back. However, when the condition $(\mathtt{a}_7, \mathtt{g}_4)$ is applied in the rule of the next instruction the proof breaks down. Cell curr mentioned in $\mathtt{g}_4$ may not be in the memory described by the assertion $\mathtt{a}_7 \circledast \mathtt{m}$ because that

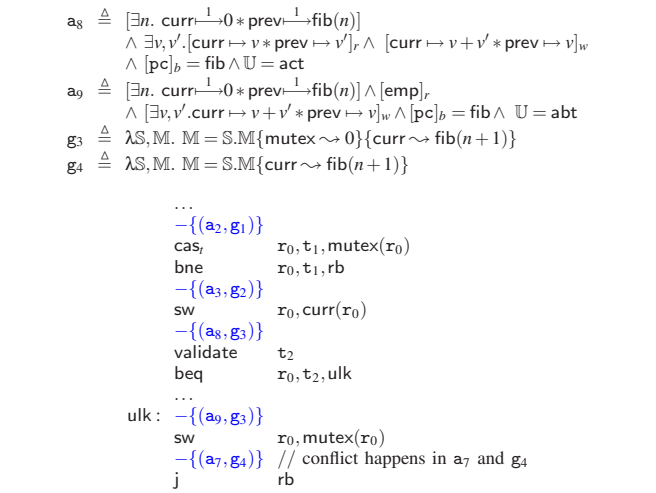

Figure 13. Violation of Atomicity

thread does not hold the storable lock anymore. Then g_4 is a false predicate due to the conflict between a_7 and g_4 and the proof can not go ahead. In order to finish the atomicity proof it must first restore curr and then g_4 will not mention cell curr anymore. In this way when a lock is released not in status cmt, it implicitly acquires that the protected memory block looks like unchanged before. Then the strong isolation entirely holds since the intermediate memory updates are invisible to other threads. In summary, both the atomicity and strong isolation properties are guaranteed in the atomicity proof.

VI. Related work and conclusions

Transactional memory, as applied to programming languages, was first studied by Herlihy and Moss [1]. The primary goal is to make it easier to perform general atomic updates of multiple independent memory words, avoiding the problems of locks. It is a hardware implementation and rely on the assumption that transactions have short durations and small data sets. Shavit and Touitou [4] proposed the first software implementation handling transactions with statically known read and write sets. Next Blundell [12] distinguishes subtleties of transactional memory atomicity semantics. It defines strong atomicity and weak atomicity semantics to characterize the atomicity property between transactional and non-transactional codes. Presently prior software TM systems mostly implement weak atomicity and it allows violation of a transaction's isolation if there is a data race between transactional and non-transactional codes. To enforce the strong atomicity semantics in STM, Shpeisman [18] implements non-transactional data accesses via read and write barriers. Recently, Guerraoui [22] presents opacity, an extension of the classical database serializability property with additional requirement that non-committed transactions are prevent from accessing inconsistent states. However, all these work above are lack of a formal reasoning of these correctness properties in STM implementations.

O'Hearn [10] proposed CSL for a high-level parallel language based on the separation logic [23]. It explicitly separates the private and shared memories and uses conditional critical regions (CCR) to permit the ownership transfer. CSL can specify well synchronized shared memory accesses and we adopt it in the program logic to formal reasoning strong atomicity. Recently, Brookes [24] provides a grainless semantics to CSL for parallel programs that share mutable states; Bornat *et al.* [11] proposed a refinement of CSL with fine-grained resource accounting.

In [25] we have presented a PCC framework for certifying concurrent programs using transactional memory. It treats the whole commit operation as a single primitive and there are no shared memory accesses outside transactions, so the strong atomicity semantics is enforced obviously. Next we refine the model by splitting the huge commit operation into several thin instructions and introduce storable locks for synchronization both in and out of transactions in [13]. At the same time the strong atomicity semantics is not enforced for sure and it depends on the actual implementation. In this paper we mainly focus on the formal reasoning the strong atomicity of STM programs. Actually we introduce a local guarantee for each instruction to specify valid memory transition before rolling back to enforce the strong atomicity. It is the first framework on formal certifying strong atomicity semantics in STM programs.

Acknowledgments

We would like to thank Prof. Zhong Shao (Yale University) and anonymous for their inspiring discussions and suggestions on this paper. This research war supported by the National Natural Science Foundation of China under (Grant No.60673126, No.90718026) and Intel China Research Center. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

[1] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *SIGARCH Comput. Archit. News*, pages 289–300, 1993.

[2] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *HPCA'05: Proceeding of the 11th International Symposim on High-Performance Computer Architecture*, pages 316–327, Washington, DC, 2005. IEEE Comp. Soc.

[3] K. E. Moore and D. Grossman. Log-based transactional memory. PhD thesis, Madison, WI, USA, 2007. Adviser-David A. Wood.

[4] N. Shavit and D. Touitou. Software transactional memory. In *PODC'95: Proceeding of the fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 204–213, New York, 1995. ACM Press.

[5] T. Harris and K. Fraser Language support for lightweight transactions In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402, New York, 2003. ACM Press.

[6] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP'06: Procedding of the eleventh ACM SIGPLAN Symposium on Principles and Practice of parallel programming*, pages 187–197, New York, 2006. ACM Press.

[7] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC'03: Proceedings of the twenty-second annual Symposium on Principles of Distributed Computing*, pages 92–101, New York, 2003. ACM Press.

[8] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *Lecture Notes in Computer Science*, pages 194–208, 2006. Springer Berlin.

[9] T. Harris, S. Marlow and P. Jones. Composable memory transactions. In *PPoPP'05: Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of parallel programming*, New York, 2005. ACM Press.

[10] P. W. OHearn. Resources, coucurrency, and local reasoning. *Theor. Comput. Sci.*, pages 271–307, 2007.

[11] R. Bornat, C. Calcagno, P.O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *POPL'05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 259–270, New York, 2005. ACM Press.

[12] C. Blundell, E. Lewis, and M. K. Martin. Subtleties of transactional memory atomicity semantics. In *IEEE Comput. Archit. Lett.*, page 17, 2006

[13] Yong Li, Yu Zhang, Yiyun Chen and Ming Fu. Formal reasoning concurrent programs using a lazy-STM system. In *http://ssg.ustcsz.edu.cn/vsync/papers/frcptm/*.

[14] Coq Development Team. The Coq proof assistant reference manual. Coq release v8.1, October 2006.

[15] C. Paulin-Mohring. Inductive definitions in the system Coq-rules and properties. In *Proc. TLCA*, volume 664 of *LNCS*, 1993.

[16] D. Yu and Z. Shao. Verification of safety properties for concurrent assembly code. In *ICFP'04: Proceedings of the 2004 ACM SIGPLAN International Conference on Functional Programming*, pages 175–188, Utah, September 2004. ACM Press.

[17] X. Feng and Z. Shao. Modular verfication of concurrent assembly code with dynamic thread creation and termination. In *ICFP'05: Proceedings of the 2005 ACM SIGPLAN International Conference on Functional Programming*, pages 254–267, New York, 2005. ACM Press.

[18] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. F. Moore, and B. Saha. Enforcing isolation and ordering in STM. In *PLDI'07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 78–88, New York, 2007. ACM Press.

[19] A. W. Appel Foundational proof-carrying code In *Proc. 16th Annu. IEEE Symp. on Logic in Computer Science*, pages 247–258, 2001. IEEE Comp. Soc.

[20] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. In *Infromation and Computation*, pages 38–94, 1994.

[21] Y. Li. Coq implementation for On the Verification of Strong Atomicity of Programs Using STM. In *http://ssg.ustcsz.edu.cn/vsync/papers/ovsaps/*.

[22] R. Guerraoui and M. Kapalka. On the Correctness of Transactional Memory In *PPoPP'08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, New York, 2008, ACM Press.

[23] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS'02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55-74, Washington, DC, 2002. IEEE Comp. Soc.

[24] S. Brookes. A grainless semantics for parallel programs with shared mutable data. In *MFPS'06: Proceeding of the 21st Annual Conference on Mathematical Foundational of Programming Semantics*, pages 277–307, Washington, DC, 2006. IEEE Comp. Soc..

[25] L. Li, Y. Zhang, Y. Chen, and Y. Li. Certifying concurrent programs using transactional memory. In *Journal of Computer Science and Technology*, 24(1):110 121, Jan.2009.