Formalising of Transactional Memory using Interval Temporal Logic (ITL)

Amin El-kustaban, Ben Moszkowski and Antonio Cau
Software Technology Research Laboratory
De Montfort University, England
Email: {amin, benm, acau}@dmu.ac.uk

***Abstract*—Transactional memory (TM) is a promising lock-free technique that can avoid the problems associated with locking. It provides a more general and flexible way than other lock-based techniques by allowing programs to atomically read and modify disparate memory locations as a single operation. We propose a general specification model for an abstract TM with verification for various correctness conditions of concurrent transactions. This model has been constructed as a base to develop a general and flexible formal framework that can prove and validate the correctness of TM systems. Interval Temporal Logic (ITL) is used to build and verify this model, since it offers a powerful tool supporting logical reasoning about time intervals as well as programming and simulation.**

I. Introduction

The primary challenge in a system running multiple processes is how to control access to shared data in order to ensure correct behaviour and data consistency. Memory synchronization to deal with this challenge can involve lock-based, lock-free or wait-free techniques. However, using locks can lead to deadlock, convoying and priority inversion problems [1]. Although lock-free and wait-free techniques could be used to avoid the problems with locks, at the present they are still complex to use and compose [2], [3].

Transactional memory (TM) is a promising lock-free technique that offers a high-level abstract parallel programming model for future chip multiprocessor (CMP) systems. Moreover it adapts the popular well established paradigm of transaction, thus providing a general and flexible way of allowing programs to atomically read and modify disparate memory locations as a single operation [4].

There are several recent proposals for implementing the TM in hardware [1], [5], [6], software and a hybrid hardware-software combination [7], [8]. However, a formal underpinning encompassing specification, design and implementation of TM still needs much effort. In addition, the formal verification of any newly suggested TM system is required to check that the new proposed ideas satisfy the correctness conditions of TM [2], [9]. Some researchers have proposed different formal frameworks for proving the safety of a TM system but these are still hard to understand and use [10], [11], [12].

In this paper, we propose a general and flexible formal TM model based on the ITL to facilitate fully formal proofs and verify the correctness of different TM systems.

We firstly present an abstract TM model and its safety conditions which are based on well-known published papers on generalising the safety of TM such as [10], [13], [14]. The specification validation of the proposed abstract TM model is provided in another concurrent work. We then verify that the proposed TM model satisfies these safety conditions.

The paper is organised as follows. Related work is discussed in Sect. II. A brief description of Interval Temporal Logic (ITL) is given in Sect. III. We then introduce in Sect. IV a TM computational model. Several specifications of standard correctness conditions of TM are illustrated in Sect. V. A verification of the abstract TM correctness is given in Sec.VI.

II. Related work

Earlier work on the TM's formalisation and verification can be divided into the following two parts:

- Pure semantics for describing general correctness of the TM systems with some illustrations for special properties (e.g. sequential specifications and opacity). Scott's paper [13] was the first to offer sequential specifications that could to capture many semantics of transactional memory that were defined on the conventional notion of the sequential histories. He presents practical policies for detecting conflicts and arbitration functions for ensuring the progress of transactions. Guerraoui and Kapalka[14] present an opacity as a safety property for TM systems. They extend the notion of serialisability to include the concept that the aborted transactions should not access an inconsistent state of the memory, which can be doomed in Software Transactional Memory (STM) (due to infinite loops, or exceptions).
- A compositional method for defining the TM semantics and proving that a transactional memory system satisfies its specifications. Cohen et al. [10] and Tasiran [15] introduce a methodology, supported by tools, to formally verify the correctness of a TM system. They use an automated theorem prover to obtain mechanical proofs. Guerraoui et al.[16] and Emmi et al[17] propose an algorithm capable of verifying that TM systems satisfy strict serialisability with respect to opacity as a safety condition. These researches (except Cohen) focus only on the STM systems and neglect the hybrid and hardware transactional memory.

III. Interval Temporal Logic

Interval Temporal Logic (ITL) is an important temporal logic for both propositional and first order logical reasoning about intervals of time. ITL is useful in the formal description of linear discrete systems for several reasons. It is a flexible notation for discrete linear order. Also, ITL has capability of handling both sequential and parallel composition unlike most temporal logic. A powerful and extensible specification framework is also offered by ITL for reasoning about properties involving safety, liveness and projected time. In addition, an executable with animation framework for experimenting and developing ITL specification is provided by Tempura [18], [19], [20].

A. Syntax and Semantics

The syntax of ITL (including integer expressions and first order formulae) is defined in Table I, where: z denotes an integer value, a is a static (global) variable which do not vary over time, A is a state variable which can change within an interval, v a static or state variable, g is a function symbol, h is a predicate symbol, and f is a formula.

TABLE I
Syntax of ITL

Expressions
$exp ::= \; z \mid a \mid A \mid g(exp_1, \ldots, exp_n) \mid \quad A \mid finA$
Formulae
$f ::= \; h(exp_1, \ldots, exp_n) \mid \neg f \mid f_1 \wedge f_2 \mid \forall v \cdot f \mid skip \mid f_1;\; f_2 \mid f^*$

Finite and infinite sequence of states can be represented in ITL using an interval σ, which is the key notion of ITL. An interval σ is divided into a finite or infinite sequence of one or more states $\sigma_0\sigma_1 \ldots$. Where each state σ_i maps each variable to some value. The length, $|\sigma|$, of an interval σ is equal to one less than the number of states in the interval.

The informal semantics of the various useful ITL constructs and some derived constructs are defined in Table II and as follows:

- A: value of A in the next state.
- $finA$: value of A in the last state.
- $skip$: unit interval (length 1).
- $f_1;\;\; f_2$: holds if the interval can be decomposed (”*chopped*”) into a prefix and suffix interval, such that f_1 holds over the prefix and f_2 over the suffix, or if the interval is infinite and f_1 holds for that interval.
- f^* : holds if the interval is decomposable into a finite number of intervals such that for each of them f holds, or the interval is infinite and can be decomposed into an infinite number of finite intervals for which f holds.

B. ITL and TM

We base our framework on ITL and its programming language subset Tempura. Our selection is based on a number of points:

- Transactions basically are a set of intervals that can be mapped with a specific beginning and ending state. Each transaction in the history of concurrent transactions can be expressed and reason in a compositional way using subinterval ITL notations such as and .
- The order of transactions with their relevant operations is significant for ensuring the conflict-free property. The $skip$ and $chop$ operators are powerful, showing by ITL can easily handle the transactions' order.
- Specifying the end time of each committed transaction is also important for verifying the strict serialisable safety condition. The ITL's fin and $chop$ operators can be used for this purpose.

TABLE II
ITL derived constructs

$true$	$\widehat{=}$	$0 = 0$	True value.
$false$	$\widehat{=}$	$\neg true$	False value.
$f_1 \vee f_2$	$\widehat{=}$	$\neg(\neg f_1 \wedge f_2)$	Or.
$f_1 \supset f_2$	$\widehat{=}$	$\neg f_1 \vee f_2$	Implies.
$\exists v.f$	$\widehat{=}$	$\neg \forall v. \neg f$	Exists.
inf	$\widehat{=}$	$true;\; false$	Infinite interval.
$finite$	$\widehat{=}$	$\neg inf$	Finite interval.
f	$\widehat{=}$	$skip;\; f$	Next.
$more$	$\widehat{=}$	$true$	$\geq$2 states .
f	$\widehat{=}$	$finite;\; f$	Eventually.
f	$\widehat{=}$	$\neg \quad \neg f$	Henceforth.
f	$\widehat{=}$	$(f;\; true)$	Some subinterval.
f	$\widehat{=}$	$\neg \quad \neg f$	All subintervals.
f	$\widehat{=}$	$(more \wedge f)$	Some nonempty subinterval.
f	$\widehat{=}$	$(more \supset f)$	All nonempty subintervals.
f	$\widehat{=}$	$(f \wedge finite);\; true$	Some finite prefix.
f	$\widehat{=}$	$\neg \quad \neg f$	All finite prefix.
$fin\; f$	$\widehat{=}$	$(empty \supset f)$	Final state.
$halt\; f$	$\widehat{=}$	$(empty \equiv f)$	Exactly in the final state.

IV. TM computational model

In this section we present an abstract model to specify TM which is similar to [13], [10], [14]. The main difference is that we represent the history of events as a time interval and each sequence of events as a subinterval. This simplifies dealing with various TM correctness properties. For example, we can prove certain properties which were only assumed in work by others [10].

A. Processes and Transactions

An interval σ is a (in)finite sequence of one or more states $s_0, s_1, s_2, ..., s_n$. Each state has concurrent observable events E^i_j and each event belongs to process j and transaction i. A sequence of events forms a transaction T that is issued sequentially by a process P. Process P_j cannot invoke a new transaction $T^{i_1}_j$ until the preceding transaction $T^{i_0}_j$ terminates. Also, a transaction T^i_j, which has a unique identifier $id(i, j)$ (that will help in capturing the properties of each transaction that invokes by the same process), cannot invoke the next operation (E^i_j) until the previous operation E^i_j gets a response and cannot invoke an operation after it gets a commit or abort response, Where:

$$\begin{array}{ll} tm_{spec} & \widehat{=} \bigwedge_{j=0}^{m} P_j \\ P_j & \widehat{=} T_j^0;\ T_j^1;\ ...;\ T_j^i \\ T_j^i & \widehat{=} E_{j_0}^i;\ E_{j_1}^i;\ ...;\ E_{j_s}^i \end{array}$$

Events and Objects

The atomic read and write events of this model can access a set of base objects obj. An object is a high-level representation of memory and initially all values of these objects are uninitialised and hence equal to $\perp$. An event is either an invocation by a transaction or a response as follows:

- $R_p^t(x)$: a read operation, by transaction t which is issued by process p, responds with the current value u of object x as $\widehat{R}_p^t(x,u)$.
- $W_p^t(y,u')$: a write value u' operation to object y by transaction t which is issued by process p, responds with ok .
- $tryCom_p^t$: a commit request, by transaction t which is issued by process p. If the attempt to commit succeeds, the response is com_p^t(or the notation $\oplus_p^t$) and it changes the write set to become permanent. If it fails the response is $abort_p^t$(or the notation $\otimes_p^t$) and it discards all changes to the write set.
- $tryAbort_p^t$: an abort request, by transaction t which is issued by process p, the response is $\otimes_p^t$ and it discards all changes to the write set.

B. Transition Behaviour

The states transition of the tm_{spec} are, for better readability, partitioned into two tables: Table III lists all possible invocation states of the tm_{spec}, while Table IV lists all possible responses for invocation of transactional operations states. Both tables describe the preconditions under which the transition can be taken and describe the effects of the transition on other variables. The *skip* formula, in the precondition column of both tables, describes that we use an interval of two states.

TABLE III
THE INVOCATION ACTIONS OF tm_{spec}'S TRANSACTIONAL OPERATIONS

Event*		Preconditions	Actions
$R_p^t(x)$	s_0	P_p=$free$ $\wedge$ T_p^t=*idle*	*skip* $\wedge$ P_p=*busy* $\wedge$ *ConflictDetRes*$(p,t,\varepsilon,\varepsilon_r)$
	s_1	P_p=*busy* $\wedge$ T_p^t=*active*	*skip* $\wedge$ *stable*(P_p) $\wedge$ *ConflictDetRes*$(p,t,\varepsilon,\varepsilon_r)$
	s_2	P_p=*busy* $\wedge$ T_p^t=*doomed*	*skip* $\wedge$ *stable*(P_p) $\wedge$ *stable*(T_p^t)
	s_3	*Otherwise*	*skip* $\wedge$ *error out*
$W_p^t(y,u')$	s_4	same as *read*	same as *read* event
$tryCom_p^t$	s_5	T_p^t=*active*	*skip* $\wedge$ *ConflictDetRes*$(p,t,\varepsilon,\varepsilon_r)$
	s_6	$\neg T_p^t$=*active*	*skip* $\wedge$ T_p^t=*doomed*
$tryAbort_p^t$	s_7	-	-

* For better readability the conditions for each state's actions are divided into two columns: event and preconditions.

In Table III, the transitions $s_0 \dots s_4$ deal with *read* and *write* transactional operations. Instead of adding a new operation for opening a transaction, s_0 is concerned with the beginning of a new transaction as follows: If process p is *free* and transaction t is *idle*, then p can invoke t by transferring the status of p to *busy* and t to *active*. This prevents other transactions being created until the existing one terminates. Transaction t can invoke an operation at the beginning state of t and while its status is not equal to *finished*. The action of (s_1) is an invocation for an operation using the same *active* transaction that may become *doomed* in the next state if a conflict is detected by the *ConflictDetRes*(), while the action of (s_2) is an invocation for an operation using the same *doomed* transaction that should stabilise its status in order to eventually abort.

The *ConflictDetRes*$(p,t,\varepsilon,\varepsilon_r)$ formula checks whether transaction t in process p conflicts (where ε specifies the type of detection) with the concurrent transactions that have been issued by other processes and then resolves this conflict (where ε_r specifies the type of resolution). The possible types of conflict detection such as lazy, eager and mixed, and types of conflict resolution such as eager and lazy arbitration are explained in detail in the next section. As shown in Table III we can note the replication of using the *ConflictDetRes*() to preserve the generality of this *TM* model. For example, if $\varepsilon = \varepsilon_l$ (Lazy), then the *ConflictDetRes*() will be activated at commit time ($trycom$), while the others will be neglected.

TABLE IV
THE RESPONSE ACTIONS OF tm_{spec}'S TRANSACTIONAL OPERATIONS

Event*		Preconditions	Actions
$R_p^t(x)$	$\widehat{s}_0$	$\exists$ { *a local write* $W_p^t(x,u') \wedge$ *no write in between*}	*skip* $\wedge$ u=u' $\wedge$ E_p^t= $\widehat{R}(x,u)$
	$\widehat{s}_1$	$\neg\widehat{s}_0$ $\wedge$ $\phi \equiv$ (*InconsRead*() $\wedge$ T_p^t=*doomed*)	*skip* $\wedge$ u=$\perp$ $\wedge$ $AbortTran(p,t)$ $\wedge$ E_p^t=$\otimes$
	$\widehat{s}_2$	$\neg\widehat{s}_0 \wedge \neg\phi$	*skip* $\wedge$ u=*Mem[x]* $\wedge$ E_p^t= $\widehat{R}(x,u)$
$W_p^t(x,u')$	$\widehat{s}_3$	-	*skip* $\wedge$ *Assign temporary* u' *to* x $\wedge$ E_p^t= *ok*
$tryCom_p^t$	$\widehat{s}_4$	T_p^t=*active*	*skip* $\wedge$ $CommitTran(p,t)$ $\wedge$ E_p^t=$\oplus$
	$\widehat{s}_5$	T_p^t=*doomed*	*skip* $\wedge$ $AbortTran(p,t)$ $\wedge$ E_p^t=$\otimes$
$tryAbort_p^t$	$\widehat{s}_7$	-	*skip* $\wedge$ $AbortTran(p,t)$ $\wedge$ E_p^t=$\otimes$

* For better readability the conditions for each state's actions are divided into two columns: event and preconditions.

Transitions $\widehat{s}_0 \dots \widehat{s}_2$, in Table IV, list the three possible actions for responding to a transactional *read* operation $R_p^t(x)$. $\widehat{s}_0$ returns u', if there is a previous $W_p^t(x,u')$ operation for the same object x and it is issued by the same transaction t and process p. $\widehat{s}_1$ returns $\perp$ and aborts transaction t, if there isn't a previous $W_p^t(x,u')$, transaction t status is equal to *doomed* and there is an inconsistent read state *InconsRead*(), where:

$$\begin{array}{ll} InconsRead(p,t) & \widehat{=} \ \left(\widehat{R}_p^t(x,u) \wedge u = v\ \right) \wedge \quad (\neg W_p^t(x)) \\ & \wedge\ fin\left(R_p^t(y) \wedge \neg(u = v)\right) \end{array}$$

The $InconsRead()$ with $T_p^t = doomed$ indicates that transaction t has a conflict because a response value of one of the previous R operations in the same transaction has been changed by another committed transaction and the doomed transaction t may access an inconsistent value. $\widehat{s}_2$ returns a value of x from memory if there isn't a previous $W_p^t(x, u')$ and there isn't an inconsistent read access. We formalise these three possible actions in one ITL formula called $ValidRead()$.

Transitions $\widehat{s}_4 \ldots \widehat{s}_6$ respond to $tryCom$ and $tryAbort$ tm_{spec} instructions. The actions are either committing a transaction (making all the temporary updates permanent using $CommitRes()$) or aborting a transaction (undoing any update using $AbortRes()$).

V. TM Safety Properties

Many TM correctness conditions have been proposed with varying degrees of accuracy. The basic correctness property for concurrent transactions is serialisability. A TH (transactional history) is serialisable if the result of all committed concurrent transactions in TH that are generated by a TM system is identical to a result in some STH (sequential transactional history) which represents the same transactions executed serially (more details in this section). In this section, we use ITl to formalise some correctness conditions that can lead to the serialisability property and other criteria which have been considered for TM .

A. Conflict Free

A conflict appears when concurrently executing transactions perform operations on the same location and at least one of them modifies the data. Scott [13] presents practical policies for detecting conflicts to describe the STH 's characteristic of different classes of TM systems. Also, he introduces arbitration functions to ensure progress by specifying which of the two conflicting transactions will fail.

Conflict Detection: We formalise different classes of conflict detecting which are denoted by (ε):

- Lazy Conflict (ε_l): process $p's$ transaction t and process $q's$ transaction s conflict if there exist operations W an object in s and R the same object in t such that s commits before the end of t, see Figure 1.

$$\begin{aligned}\varepsilon_l \quad &\widehat{=} fin(tryCom_q^s) \wedge (\quad W_q^s(y) \wedge \quad R_p^t(y) \\ &\qquad \wedge fin(T_p^t = active))\end{aligned}$$

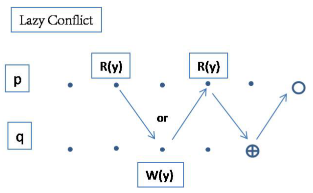

Fig. 1. Lazy conflict.

- Eager Conflict (ε_e): process $p's$ transaction t and process $q's$ transaction s conflict if t and s have a lazy conflict or if there exist operations R an object in s and W the same object in t such that W precedes R or vice versa, but neither transaction has ended, see Figure 2.

$$\begin{aligned}\varepsilon_e \quad &\widehat{=} \varepsilon_l \vee \Big((fin(R_q^s(y)) \wedge (\quad W_p^t(y) \wedge fin(T_p^t = active))) \\ &\qquad \vee (fin(W_p^t(y)) \wedge (\quad R_q^s(y) \wedge fin(T_q^s = active))) \Big)\end{aligned}$$

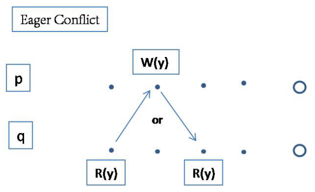

Fig. 2. Eager conflict.

Conflict Resolution: Transactional memory systems have a contention management policy (arbitration) to resolve a conflict between two transactions by aborting one of them. Scott [13] suggests two arbitration functions:

- Eagerly aggressive arbitration(ε_{re}): whoever started early fails.

$$\varepsilon_{re} \quad \widehat{=} fin \otimes_p^t \wedge (\quad T_p^t = active; \quad T_q^s = idle)$$

- Lazily aggressive arbitration(ε_{rl}): whoever tries to commit first wins.

$$\varepsilon_{rl} \quad \widehat{=} fin \otimes_p^t \wedge (\neg \quad tryCom_p^t \wedge \quad tryCom_q^s)$$

Definition 1. *Transaction t in process p is called conflict-free if there is no transaction s in process q such that t and s is conflicting with t to which t loses at arbitration.*

$$\begin{aligned}ConflictFree(\varepsilon, \varepsilon_r) \quad &\widehat{=} \neg \quad ((\varepsilon) \supset \neg(\varepsilon_r)) \\ \varepsilon \quad &\widehat{=} \varepsilon_l \vee \varepsilon_e \\ \varepsilon_r \quad &\widehat{=} \varepsilon_{re} \vee \varepsilon_{rl}\end{aligned}$$

B. Read-Consistency

1) Local read consistency: Each committed or aborted transaction in tm_{spec} satisfies local read consistency iff each read operation is responded to with a value that has been written by a previous write operation for the same variable and in the same transaction.

$$\begin{aligned}Local \quad &\widehat{=} \neg \quad (\varphi \supset u \neq u') \\ \varphi \quad &\widehat{=} \Big((W_p^t(x, u') \wedge skip); \quad (\neg W_p^t(x)); \; (\widehat{R}_p^t(x, u) \wedge empty) \Big)\end{aligned}$$

2) Doomed Consistency: Kapalka and Guerraoui [14] extend the notion of strict serialisability to include the concept that even aborted transactions should not access an inconsistent state of the memory, which can cause infinite loops, or exceptions (divided by zero). In this model we add this extension (doomed consistency) as one of the safety conditions that can lead finally to strict serialisability with the property that even aborted transactions do not observe an inconsistent state.

Each transaction in tm_{spec} satisfies the doomed consistency iff a later R operations does not access an inconsistent state that comes when the response value of one of the previous R operation in the same transaction has been changed.

$$
\begin{aligned}
\mathit{Doomed} \quad & \mathrel{\widehat{=}} \neg \quad (\psi \supset u \neq \bot) \\
\psi \quad & \mathrel{\widehat{=}} \quad (\neg W_p^t(x)) \wedge \Big(\big((R_p^t(y) \wedge \mathit{empty}; \quad W_q^s(y)) \\
& \qquad \vee (W_q^s(y) \wedge \mathit{empty}; \quad R_p^t(y))\big) \\
& \qquad \wedge \mathit{fin}(\oplus_q^s \wedge T_p^t = \mathit{active})\Big) \; ; \; \mathit{fin}(\widehat{R}_p^t(x,u) \wedge \otimes_p^t)
\end{aligned}
$$

3) Global read consistency: A transaction in tm_{spec} satisfies the global consistency iff each $R(x,u)$ in this successful transaction (conflict free or not doomed) returns the most recent $W(x,u'')$ in a committed transaction.

$$
\begin{aligned}
\mathit{Global} \quad & \mathrel{\widehat{=}} \neg \quad (\alpha \supset u \neq u'') \\
\alpha \quad & \mathrel{\widehat{=}} \quad (\neg W_p^t(x)) \wedge \Big(\big((W_q^s(x,u'') \wedge \mathit{skip}; \quad (\neg W_q^s(x))) \\
& \qquad \wedge \mathit{fin}\oplus_q^s\big) \wedge \quad (\neg W_j^i(x)) \wedge \mathit{fin}\oplus_j^i\Big) \\
& \quad ; \; \mathit{fin}\big(\widehat{R}_p^t(x,u) \wedge \neg(T_p^t = \mathit{doomed} \wedge \mathit{InconsRead}())\big)
\end{aligned}
$$

C. Strict serialisability

Papadimitriou [21] augments the strength of serialisability by adding the requirement of the real time order of the committed transactions. We formalise this property as follows: Let σ' be obtained from σ by serialising the concurrent committed transactions in TH. Since we have preserved each transaction in an independent list in the tm_{spec} proposed model, which means each transaction with its events is considered as one block, we do not need to reorder events to transfer the TH to the STH. Instead, the events of each transaction can be collected by specifying the process and transaction for each event.

Definition 2. *The TH can be strictly serialised, if we can obtain σ' from σ with respect to $Ser(TH)$ as follows:*

$$
\begin{aligned}
\mathit{Ser}(TH) \quad & \mathrel{\widehat{=}} (T_p^t;\, T_q^s) \\
& \mathrel{\widehat{=}} T_p^t \| T_q^s \\
& \quad \wedge \{\textit{The order of transactions over } \sigma' \textit{ is the} \\
& \qquad \textit{order of the committing events for the} \\
& \qquad \textit{same transactions } (\mathit{fin}\oplus_p^t;\ \mathit{fin}\oplus_q^s) \textit{ over } \sigma\} \\
& \quad \wedge \{\forall (R_p^t \wedge R_q^s) \textit{ over } \sigma \textit{ respects the RC property}\} \\
& \quad \wedge \{\forall (\oplus_p^t \wedge \oplus_q^s) \textit{ over } \sigma \textit{ respects the ConflictFree()}\}
\end{aligned}
$$

Where RC represents the read consistency safety condition:

$$RC \mathrel{\widehat{=}} \mathit{Local} \wedge \mathit{Doomed} \wedge \mathit{Global}$$

VI. Verification of Abstract TM

We simplify the deductive verification approach by viewing the tm_{spec} model from the viewpoint of TM safety properties. This simplification approach shifts the burden of the verification from a global level to the local components that may violate the safety properties. In actual fact, the main safety condition in the TM community is the strict serialisability with respect to doomed consistency. However, the verification of this property depends on others two properties which are the read-consistency and conflict-free. In the tm_{spec} case, the two main components that can affect the read consistency, detection conflict and correct resolution of conflict are $\mathit{ValidRead}()$ and $\mathit{ConflictDetRes}()$. We therefore firstly verify that each of these components satisfies the corresponding property as follows:

$$
\begin{aligned}
\vdash \quad & \mathit{ValidRead}() \quad \supset RC & (1) \\
\vdash \quad & \mathit{ConflictDetRes}(\varepsilon, \varepsilon_r) \quad \supset \mathit{ConflictFree}(\varepsilon, \varepsilon_r) & (2)
\end{aligned}
$$

To prove these implications, we simplify and divide the lhs of each part to reduce the formulae size and then we prove that each formula satisfies the rhs by using propositional reasoning, definition of ITL operators and ITL inference rules, see the Appendix. Moreover, some definitions are used in this verification approach to cover the relationship between the model's components. For example, the following definitions show the relationship between the $\mathit{CommitTran}()$ and the $\mathit{ValidRead}()$.

Definition 3. *A tm_{spec} model permanently updates its memory location $\mathit{Memory}[x]$ with value u'' by $\mathit{CommitTran}()$ formula only iff β is satisfied :*

$$
\begin{aligned}
\beta \equiv \Big(& \big((W_q^s(x,u'') \wedge \quad (\neg W_q^s(x))) \wedge \mathit{fin}\oplus_q^s\big) \wedge \quad (\neg W_j^i(x)) \\
& \wedge \mathit{fin}\oplus_j^i \Big)
\end{aligned}
$$

Definition 4. *Status of a transaction T_p^t in tm_{spec} is changed from active or idle to doomed only iff a conflict with another transaction T_q^s has been detected by a $\mathit{ConflictDetRes}()$ formula. Transaction T_p^t which has a doomed status must eventually be aborted.*

Moreover, as there are no other events that interleave an invocation and its response, we will regard, for the sake of simplicity, the invocation and its response as a single form, such as: (where no_{ev} means no event)

$$
\begin{aligned}
\forall x, u \quad & \Big(\mathit{fin}\widehat{R}_p^t(x,u) \equiv \quad (R_p^t(x) \wedge (\quad no_{ev_p^t}) \wedge \widehat{R}_p^t(x,u))\Big) \\
\forall y, u' \quad & \Big(\mathit{fin}W_p^t(y,u') \equiv \quad (W_p^t(y,u') \wedge (\quad no_{ev_p^t}) \wedge ok_p^t)\Big)
\end{aligned}
$$

Because of the lack of space, the proofs of the first part of Lemma 1 will be presented in the appendix. In this section we only present the simplification of Lemma 1 and prove of strict serialisability as follows:

Lemma 1

The ITL formula of $\mathit{ValidRead}()$

$$
\begin{aligned}
\mathrel{\widehat{=}} \quad & \Big(\mathit{fin}(\widehat{R}_p^t(x,u)) \\
& \quad \wedge \big((f_0 \wedge f_1) \vee (\neg f_0 \wedge \quad (\neg W_p^t(x)) \\
& \qquad \wedge ((f_2 \wedge f_3) \vee (\neg f_2 \wedge f_4)))\big)\Big)
\end{aligned}
$$

where

$$
\begin{aligned}
f_0 \quad & \mathrel{\widehat{=}} \quad W_p^t(x,u') \wedge \quad (\neg W_p^t(x)) \\
f_1 \quad & \mathrel{\widehat{=}} \quad u = u' \\
f_2 \quad & \mathrel{\widehat{=}} \quad \mathit{fin}(T_p^t = \mathit{doomed} \wedge \mathit{InconsRead}()) \\
f_3 \quad & \mathrel{\widehat{=}} \quad u = \bot \wedge \mathit{fin}\otimes_p^t \\
f_4 \quad & \mathrel{\widehat{=}} \quad u = \mathit{Memory}[x]
\end{aligned}
$$

We will use the u' symbol to represent a local write value, the u'' symbol to represent a global write value. Also, let us divide the $\mathit{ValidRead}()$ formula into three parts and prove each one:

$$
\begin{aligned}
\varphi' \mathrel{\widehat{=}} \quad & \Big(\mathit{fin}\widehat{R}_p^t(x,u) \wedge (f_0 \wedge f_1)\Big) \supset \text{RC 1.1} \\
\psi' \mathrel{\widehat{=}} \quad & \Big(\mathit{fin}\widehat{R}_p^t(x,u) \wedge \neg f_0 \wedge \quad (\neg W_p^t(x)) \wedge (f_2 \wedge f_3)\Big) \supset \text{RC 1.2} \\
\alpha' \mathrel{\widehat{=}} \quad & \Big(\mathit{fin}\widehat{R}_p^t(x,u) \wedge \neg f_0 \wedge \quad (\neg W_p^t(x)) \wedge (\neg f_2 \wedge f_4)\Big) \supset \text{RC 1.3}
\end{aligned}
$$

As mentioned previously, we prove Lemma 1 and 2 to led us prove the strict serialisability with respect to doomed consistency which is considered the main standard safety condition. According to the serialisability specification in the previous section, it depends on the correctness of the previous two main components. Since we prove the correctness of these components, the strict serialisability can be proved simply using a contradiction as follows:

Theorem 1. *The TH σ can be serialised with respect to $Ser(TH)$ if there is no overlap and conflict over σ.*

Proof: let p and q be processes, t and s be transactions, u and $u' \in \mathbb{N}$ and x a memory location, such that t and s have overlap and conflict $\big((W_p^t(x,u);\ \widehat{R}_q^s(x,u')) \wedge fin\oplus_q^s;\ fin\oplus_p^t\ \big)$. We can assume that σ can be serialised with respect to $ser(TH)$. According to Definition 2, σ' can be obtained from σ and satisfies that each commit respects the $ConflictDetRes()$ formula (that satisfies $ConflictFree()$) which is in violation here. Since transaction s will detect a conflict with t and resolve it by changing the status of t to doomed, the response to the request to commit by transaction t will respond with abort $(fin\otimes_p^s)$ instead of $(fin\oplus_p^s)$. Consequently, we conclude that σ can not be serialised. ■

Appendix
Proof of Lemma 1.1

Proof: we can start by simplifying the lhs of φ'

$$\begin{array}{lll}
\varphi' & \widehat{=} & \Big(fin(\widehat{R}_p^t(x,u)) \wedge (f_0 \wedge f_1)\Big) \\
 & \equiv\{ \textit{ redistributing the brackets } \} & \\
lhs & \widehat{=} & \Big((fin(\widehat{R}_p^t(x,u)) \wedge f_0) \wedge f_1\Big) \\
 & \equiv\{ \textit{ propositional reasoning}\} & \\
 & \equiv & \Big(\neg\Big((fin(\widehat{R}_p^t(x,u)) \wedge f_0) \supset \neg f_1\Big)\Big) \\
 & \equiv \{\textit{definition of} \quad \} & \\
 & \equiv \neg & \neg\Big(\neg\Big((fin(\widehat{R}_p^t(x,u)) \wedge f_0) \supset \neg f_1\Big)\Big) \\
 & \equiv \neg & \Big((fin(\widehat{R}_p^t(x,u)) \wedge f_0) \supset \neg f_1\Big)
\end{array}$$

$$\begin{array}{ll}
f_0 & \widehat{=}\ W_p^t(x,u') \wedge \quad (\neg W_p^t(x)) \\
 & \equiv \{\textit{definition of ITL operators} \quad \textit{and ITL Inference Rules.}\} \\
 & \equiv W_p^t(x,u') \wedge (skip; \quad (\neg W_p^t(x))) \\
 & \equiv (W_p^t(x,u') \wedge skip); \quad (\neg W_p^t(x)) \\
f_1 & \equiv u = u' \\
\neg f_1 & \equiv u \neq u'
\end{array}$$

Then , by applying the definition of ITL operators fin, the lhs will be:

$$\begin{array}{ll}
lhs & \equiv \neg \quad \Big(\Big(\ W_p^t(x,u') \wedge skip; \quad (\neg W_p^t(x));\ \widehat{R}_p^t(x,u) \wedge empty\Big) \\
 & \qquad \supset u \neq u'\ \Big)
\end{array}$$

And , by assuming that u has a local write value u' we can simplify the rhs of $\varphi'(RC)$:

$$\begin{array}{ll}
rhs & \equiv Local \wedge Doomed \wedge Global \\
 & \equiv \neg \quad \Big[\Big(\Big(\ W_p^t(x,u') \wedge skip; \quad (\neg W_p^t(x)); \\
 & \qquad \widehat{R}_p^t(x,u) \wedge empty\Big) \supset u \neq u'\ \Big) \\
 & \qquad \wedge(\psi \supset True) \wedge (\alpha \supset True)\Big] \\
 & \equiv \neg \quad \Big[\Big(\ \Big(\ W_p^t(x,u') \wedge skip; \quad (\neg W_p^t(x)); \\
 & \qquad \widehat{R}_p^t(x,u) \wedge empty\Big) \supset u \neq u'\ \Big) \\
 & \qquad \wedge True \wedge True\Big]
\end{array}$$

■

Acknowledgment

We would like to thank Hussein Zedan and all members of the Software Technology Research Laboratory for valuable discussion.

References

[1] M. Herlihy and J. E. B. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993, pp. 289–300.

[2] J. R. Larus and R. Rajwar, *Transactional Memory.* Morgan and Claypool, 2006.

[3] J. Parri, "An introduction to transactional memory," in *ELG7187 Topics In Computers: Multiprocessor Systems On Chip*, fall 2010.

[4] J. Larus and C. Kozyrakis, "Is tm the answer for improving parallel programming?" *Communication of the ACM*, vol. 51, no. 7, pp. 80–88, July 2008.

[5] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional memory coherence and consistency," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*. IEEE Computer Society, Jun 2004, p. 102.

[6] A. M. El-Kustaban, A. H. El-Mahdy, and O. M. Ismail, "A cmp with transactional memory: Design and implementation using fpga technology." in *IMECS'07*, 2007, pp. 1680–1685.

[7] N. Shavit and D. Touitou, "Software transactional memory," in *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, Aug 1995, pp. 204–213.

[8] A. Shriraman, M. F. Spear, H. Hossain, V. Marathe, S. Dwarkadas, and M. L. Scott, "An integrated hardware-software approach to flexible transactional memory," in *Proceedings of the 34rd Annual International Symposium on Computer Architecture*, Jun 2007.

[9] T. Harris, A. Cristal, O. Unsal, E. Ayguade, F. Gagliardi, B. Smith, and M. Valero, "Transactional memory: An overview," *Micro, IEEE*, vol. 27, no. 3, pp. 8 –29, may-june 2007.

[10] A. Cohen, J. W. O'Leary, A. Pnueli, M. R. Tuttle, and L. D. Zuck, "Verifying correctness of transactional memories," in *Proceedings of the 7th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, November 2007, pp. 37–44.

[11] R. Guerraoui, T. A. Henzinger, M. Kapalka, and V. Singh, "Generalizing the correctness of transactional memory," in *Preliminary Program and Challenge Problems Exploiting Concurrency Efficiently and Correctly CAV 2009 Workshop*, Grenoble, France, 2009.

[12] A. Sinha and S. Malik, "Runtime checking of serializability in software transactional memory," in *IPDPS*. IEEE, 2010, pp. 1–12.

[13] M. L. Scott, "Sequential specification of transactional memory semantics," in *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, Jun 2006.

[14] R. Guerraoui and M. Kapalka, "On the correctness of transactional memory," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*, 2008.

[15] S. Tasiran, "A compositional method for verifying software transactional memory implementations," Microsoft Research, Tech. Rep. MSR-TR-2008-56, apr 2008.

[16] R. Guerraoui, T. A. Henzinger, B. Jobstmann, and V. Singh, "Model checking transactional memories," in *PLDI*, 2008, pp. 372–382.

[17] M. Emmi, R. Majumdar, and R. Manevich, "Parameterized verification of transactional memories," in *PLDI '10 Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2010.

[18] B. Moszkowski, "Some very compositional temporal properties," in *E.-R. Olderog, Programming Concepts, Methods and Calculi.* IFIP Transactions, 1994, pp. 307–326.

[19] A. Cau, B. Moszkowski, and H. Zedan. (2011) Interval temporal logic. [Online]. Available: http://www.tech.dmu.ac.uk/~STRL/ITL/index.html

[20] B. Moszkowski, *Executing Temporal Logic Programs.* Cambridge, England: Cambridge University Press, 1986.

[21] H. Papadimitriou, "The serializability of concurrent database updates," *ACM*, vol. 26, no. 4, pp. 631–653, 1979.