Specification Analysis of Transactional Memory using ITL and AnaTempura

Amin El-kustaban, Ben Moszkowski and Antonio Cau

Abstract—**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. In this paper, we propose a general and executable specification model for an abstract TM with validation for various correctness conditions of concurrent transactions. This model is constructed within a flexible transition framework that allows the testing of a TM model with animation. Interval Temporal Logic (ITL) and its programming language subset AnaTempura are used to build, execute, and validate this model. To demonstrate this work, we selected a queue example to be executed and illustrated with animation.**

Index Terms—**transactional memory, validation, formal specification, interval temporal logic.**

I. Introduction

The primary challenge in a system which runs multiple processes, is how to control access to shared data in order to ensure correct behaviour and data consistency. Memory synchronisation for dealing with this challenge can involve lock-based, lock-free or wait-free techniques. However, using locks can lead to problems with deadlock, convoying and priority inversion [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 can avoid lock-based problems and simplify parallel programming by transferring the burden of correct synchronisation from a programmer to a compiler and/or hardware. Moreover, TM offers a method that enables parts of a program to execute with atomicity and isolation, independently of other, concurrently executing tasks [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 implementation is required, in order 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 implementation, but these are still difficult to understand and use [10], [11].

In this paper, we propose a general and executable formal TM model based on the ITL specification of an abstract TM model, which will appear in another concurrent work, to fully validate the correctness of TM systems . We firstly present an executable specification of the abstract TM model and its safety conditions. Then, we test the correctness of the proposed TM model by executing a common concurrent data structure example on this model with animation (using (Ana)Tempura, a part of the ITL Workbench). We are not dealing here with the formal verification of properties which requires that all possible behaviours of system satisfy the properties. We are only concerned with validating properties which requires that only interesting behaviours satisfy the properties. The main aspects of this proposed model are the correctness conditions such as read consistency and strict serialisability, conflict detections policies and arbitration functions. This model and its aspects are based on well-known published papers in generalizing the correctness of TM such as [10], [12], [13].

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 validation of the abstract TM correctness is given in Sec.VI. We conclude with some remarks in Sec. VIII.

A. El-kustaban, B. Moszkowski and A. Cau are with Software Technology Research Laboratory, De Montfort University, England, e-mail: {amin, benm, acau}@dmu.ac.uk.

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 [12] 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[13] present an opacity as a safety property for TM implementations. 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 implementation satisfies its specifications. Cohen et al. [10] and Tasiran [14] introduce a methodology, supported by tools, to formally verify the correctness of a TM implementation. They use an automated theorem prover to obtain mechanical proofs. Guerraoui et al.[15] and Emmi et al[16]

propose an algorithm capable of verifying that TM implementations satisfy strict serialisability with respect to opacity as a safety condition. These researches (except Cohen) focus only on the STM implementations 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 [17], [18].

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
$exp ::= \; z \mid a \mid A \mid g(exp_1, \ldots, exp_n) \mid \bigcirc A \mid finA$
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:

- $\bigcirc A$: 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.

IV. Abstract TM model

In this section we present an abstract model to specify TM similar to [12], [10], [13]. The main difference is that

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.
$\bigcirc f$	$\widehat{=}$	$skip;\; f$	Next.
$more$	$\widehat{=}$	$\bigcirc true$	$\geq$2 states .
$\Diamond f$	$\widehat{=}$	$finite;\; f$	Eventually.
$\Box f$	$\widehat{=}$	$\neg \Diamond \neg f$	Henceforth.
$\Diamond\!\!\!\!\!{\scriptstyle a}\;\, f$	$\widehat{=}$	$\Diamond(f;\; true)$	Some subinterval.
$\boxed{a}\, f$	$\widehat{=}$	$\neg \Diamond\!\!\!\!\!{\scriptstyle a}\;\, \neg f$	All subintervals.
$\Diamond\!\!\!\!\!{\scriptstyle m}\;\, f$	$\widehat{=}$	$\Diamond(more \wedge f)$	Some nonempty subinterval.
$\boxed{m}\, f$	$\widehat{=}$	$\Box(more \supset f)$	All nonempty subintervals.
$\Diamond\!\!\!\!\!{\scriptstyle f}\;\, f$	$\widehat{=}$	$(f \wedge finite);\; true$	Some finite prefix.
$\boxed{f}\, f$	$\widehat{=}$	$\neg \Diamond\!\!\!\!\!{\scriptstyle f}\;\, \neg f$	All finite prefix.
$fin\ f$	$\widehat{=}$	$\Box(empty \supset f)$	Final state.
$halt\ f$	$\widehat{=}$	$\Box(empty \equiv f)$	Exactly in the final state.

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 just assumed in work by others [10].

A. Processes and Transactions

An interval σ is (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. A 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 us for capturing the properties of each transaction that invokes by the same process), cannot invoke a next operation ($\bigcirc E^i_j$) until the previous operation E^i_j gets a response and cannot invoke an operation after it gets a commit or abort response.

B. 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 response as follows:

- $R^t_p(x)$: a read operation, by transaction t which is issued by process p, responds with the current value u of object x as $\widehat{R}^t_p(x, u)$.
- $W^t_p(y, u')$: a write value u' operation to object y by transaction t which is issued by process p, responds with ok .
- $tryCom^t_p$: a commit request, by transaction t which is issued by process p. If the attempt to commit succeeds, the response is com^t_p(or the notation $\oplus^t_p$) and it changes the write set to become permanent. If it fails the response is $abort^t_p$(or the notation $\otimes^t_p$) and it discards all changes to the write set.

State variables:
P_p : Process status $\{free, busy\}$; where p: $(0 \leq p < num\_proc)$; initially $free$
T_p^t: Transaction status $\{idle, active, doomed, finished\}$; where t: $(0 \leq t < num\_tran)$; initially $idle$
E_p^t : An Array of list recording each event$\{no_{ev}, r, w, ok, tryCom, \oplus, \otimes\}$; initially no_{ev}
Mem_{obj} : Persistent memory $(0 \leq obj < num\_obj)$; initially $\perp$
Transaction operations:

```
$TranInvOp(p, t, op, \varepsilon, \varepsilon_r) \mathrel{\widehat{=}}$
$(skip$
 $\wedge$ if $(P_p = free) \wedge (T_p^t = idle)$
     then $(MakeProBusy(p)$
          $\wedge\ AddEv(p, t, op)$
          $\wedge\ ConflictDetRes(p, t, \varepsilon, \varepsilon_r))$
     else $(stable(P_p) \wedge AddEv(p, t, op)$
          $\wedge$ if $T_p^t = active$
              then $ConflictDetRes(p, t, \varepsilon, \varepsilon_r)$
              else $(stable(E_p^t) \wedge stable(T_p^t))))$
```

```
$TranResOp(p, t) \mathrel{\widehat{=}}$
$(skip$
 $\wedge$ if $E_p^t = w$
     then $AddEv(p, t, ok)$
     else if $E_p^t = r$
         then $(u = ValidRead(p, t)$
              $\wedge\ AddEv(p, t, u))$
         else $stable(E_p^t))$
```

```
$TranInvEnd(p, t, op, \varepsilon, \varepsilon_r) \mathrel{\widehat{=}}$
$(skip \wedge AddEv(p, t, op)$
 $\wedge$ if $(op = tryCom \wedge T_p^t = active)$
     then $ConflictDetRes(p, t, \varepsilon, \varepsilon_r)$
     else $\bigcirc T_p^t = doomed)$
```

```
$TranResEnd(p, t) \mathrel{\widehat{=}}$
$(skip$
 $\wedge$ if $(T_p^t = doomed)$
     then $AbortTran(p, t)$
     else $CommitTran(p, t))$
```

```
$CommitTran(p, t) \mathrel{\widehat{=}}$
$(\bigcirc T_p^t = finished$
 $\wedge\ AddEv(p, t, \oplus)$
 $\wedge\ MakeProFree(p)$
 $\wedge\ UpdateMemory())$
```

```
$AbortTran(p, t) \mathrel{\widehat{=}}$
$(\bigcirc T_p^t = finished$
    $\wedge\ MakeProFree(p)$
    $\wedge\ AddEv(p, t, \otimes))$
```

```
$MakeProBusy(p) \mathrel{\widehat{=}}$
$(\bigcirc P_p = busy)$
```

```
$MakeProFree(p) \mathrel{\widehat{=}}$
$(\bigcirc P_p = free \wedge \bigcirc FlushEvList(p))$
```

Fig. 1. Core part of TM executable specification

C. Executable Specification of TM

To ensure the validity of our proposed TM abstract model tm_{spec} , we build an executable specification for the tm_{spec} by refining the high-level TM abstract specification written in ITL into a set of Tempura (an executable subset of ITL) modules using the refinement rules in [17], [18]. Then, we simulate and analyse this model with the TM properties using AnaTempura .

As shown in Fig.1, the events list of the tm_{spec} are categorised into two parts: firstly *TranInvOp()*, for *read* and *write* invocation of transactional operations, and *TranResOp()*, for its response. Secondly, *TranInvEnd()*, for *tryComit* and *tryAbort* invocation of ending a transaction, and *TranResEnd()* for its response. In fact, this categorisation helps to ensure that each invocation event is followed by a response and each active transaction is eventually finished. Now, the sequence of invocation and response events that form transaction T_i and are issued by process P_j can be modelled using the ITL as follows:

$$\begin{aligned} T_j^i \mathrel{\widehat{=}} \quad & ((TranInvOp(j,i);\ TranResOp(j,i))^*; \\ & \ TranInvEnd(j,i);\ TranResEnd(j,i)) \end{aligned}$$

So, the sequence of transactions and the complete executable specification of the abstract TM model tm_{spec} is

$$\begin{aligned} tm_{spec} &\mathrel{\widehat{=}} \quad \textstyle\bigwedge_{j=0}^{n} P_j \\ P_j &\mathrel{\widehat{=}} \quad P_p = free \wedge T_p^t = idle \wedge E_p^t = no_{ev} \\ & \qquad \wedge \Big(\ (TranInvOp(j,i);\ TranResOp(j,i))^*; \\ & \qquad\qquad TranInvEnd(j,i);\ TranResEnd(j,i)\ \Big)^* \end{aligned}$$

The states variables are a set of processes with state values $\{free, busy\}$, a set of transactions with states values $\{idle, active, doomed, finished\}$, a set of events with possible values $\{no_{ev}, r, w, ok, tryCom, \oplus, \otimes\}$ and a shared memory. The $AddEv()$ works as an auxiliary function to record each operation op and its response in their process p event list E_p^t. This helps to check read consistency and detect conflicts between the concurrent active list at run time. Also, it stores the object and its value if op is write, read or response for read. The function $FlushEvList()$ clears the event list of the process p after finishing the execution of transaction t belonging to p and before initialising a new transaction.

The critical parts that perceive the safety properties of this model are the conflict detection and resolution formula $ConflictDetRes()$, that uses one of conflict detection types which are explained in detail in the next section, and the response actions of read operation $ValidRead()$. The formula $ValidRead(p,t)$ assigns an object x in the read response operation $\widehat{R}_p^t(x,u)$ to value u (initially $\perp$) that equals to one of the three followings choices: firstly, it equals to u' if there exists an operation $W_p^t(x,u')$ such that 1) $R_p^t(x)$ and $W_p^t(x)$ operations are issued by transaction t and process p, 2) $W_p^t(x)$ precedes $R_p^t(x)$ where their order satisfies $(W_p^t(x) \wedge fin R_p^t(x))$, and 3) no $W_p^t(x)$ in between, in order to preserve local consistency. Secondly, it equals to $\perp$ if there is no local write and there exists an operation $\widehat{R}_p^t(y,v)$ such that 1) $R_p^t(y)$ and $R_p^t(x)$ operations are issued by transaction t and process p, 2) $R_p^t(y)$ precedes $R_p^t(x)$ where their order satisfies $(R_p^t(y) \wedge fin R_p^t(x))$, and 3) a conflict is detected with other concurrent transactions. This prevents the later read operation $R_p^t(x)$ from accessing an inconsistent state. Finally, it equals to u'' if there are no local write and no conflict with other transactions is detected. The value u'' is equal to the value of location object x in the global memory; this preserves global consistency.

TABLE III
FORMAL TM SAFETY PROPERTIES

$\mathit{ConflictFree}(\varepsilon, \varepsilon_r)$	$\equiv \neg \Diamond_a(\mathit{ConflictDetection}(\varepsilon) \supset \neg \mathit{ConflictResolution}(\varepsilon_r))$
where ε	$\equiv \varepsilon_l \vee \varepsilon_e \vee \varepsilon_m$
ε_r	$\equiv \varepsilon_{re} \vee \varepsilon_{rl}$
ε_l	$\equiv fin(tryCom_q^s) \wedge (\Diamond W_q^s(y) \wedge \Diamond R_p^t(y) \wedge fin(T_p^t = active))$
ε_e	$\equiv \varepsilon_l \vee \Big((fin(R_q^s(y)) \wedge (\Diamond W_p^t(y) \wedge fin(T_p^t = active)))$
	$\vee \big(fin(W_p^t(y)) \wedge (\Diamond R_q^s(y) \wedge fin(T_q^s = active))\big) \Big)$
ε_m	$\equiv \varepsilon_l \vee \Big((fin(W_p^t(y)) \wedge (\Diamond(R_q^s(y) \wedge \bigcirc \Diamond W_q^s(y)) \wedge fin(T_q^s = active)))$
	$\vee \big(fin(W_q^s(y)) \wedge (\Diamond(R_q^s(y) \wedge \bigcirc \Diamond W_p^t(y)) \wedge fin(T_p^t = active))\big) \Big)$
ε_{re}	$\equiv fin \otimes_p^t \wedge(\Diamond T_p^t = active;\ \Diamond T_q^s = idle)$
ε_{rl}	$\equiv fin \otimes_p^t \wedge(\neg \Diamond tryCom_p^t \wedge \Diamond tryCom_q^s)$
Read consistency	
Local	$\equiv \neg \Diamond_a(\varphi \supset u \neq u')$
φ	$\equiv \Big((W_p^t(x, u') \wedge skip);\ \Box(\neg W_p^t(x));\ (\widehat{R}_p^t(x, u) \wedge empty)\Big)$
Doomed	$\equiv \neg \Diamond_a(\psi \supset u \neq \perp)$
ψ	$\equiv \Box(\neg W_p^t(x))$
	$\wedge\Big(\Big(\big(R_p^t(y) \wedge empty;\ \Diamond W_q^s(y)\big) \vee \big(W_q^s(y) \wedge empty;\ \Diamond R_p^t(y)\big)\Big)$
	$\wedge fin(\oplus_q^s \wedge T_p^t = active)\Big)\ ;\ fin(\widehat{R}_p^t(x, u) \wedge \otimes_p^t)$
Global	$\equiv \neg \Diamond_a(\alpha \supset u \neq u'')$
α	$\equiv \Box(\neg W_p^t(x)) \wedge \Big(\big((W_q^s(x, u'') \wedge skip;\ \Box(\neg W_q^s(x))\big) \wedge fin\oplus_q^s\big)$
	$\wedge \bigcirc \Box(\neg W_j^i(x)) \wedge fin\oplus_j^i\Big)\ ;\ fin(\widehat{R}_p^t(x, u) \wedge \neg(T_p^t = doomed))$

V. CORRECTNESS CONDITIONS OF TM

Many TM safety properties have been proposed with varying degrees of accuracy. The basic correctness property for concurrent transactions is serialisability. A TH (transactional history) is serializable if the result of all committed concurrent transactions in TH that is generated by a TM system is identical to a result in some STH (sequential transactional history) which represent 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 as shown in Table III. In addition, each property follows by a figure to illustrate its ITL formula.

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 [12] presents practical policies for detecting conflicts to describe the STH's characteristic of different classes of TM implementations. Also, he introduces arbitration functions to ensure progress by specifying which one of the two conflicting transactions must fail.

Conflict Detection: As shown in Table III, we formlise different classes of conflict detecting which is 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 Fig. 2.
- 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 Fig. 3.
- Mixed Conflict (ε_m): 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 W an object in t, R and W the same object in s such that R precedes the two W, but neither transaction has ended, see Fig. 4.

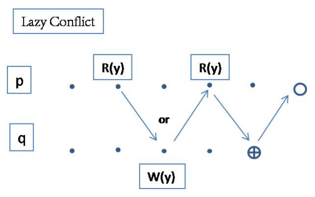


Fig. 2. Lazy conflict.

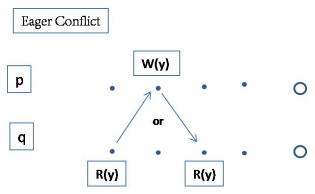


Fig. 3. Eager conflict.

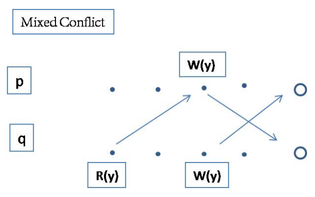


Fig. 4. Mixed conflict.

Conflict Resolution: Transactional memory implementations have a contention management policy (arbitration) to resolve a conflict between two transactions by aborting one of them. Scott [12] suggests two arbitration functions:

- Eagerly aggressive arbitration(ε_{re}): whoever started early fails.
- Lazily aggressive arbitration(ε_{rl}): whoever tries to commit first wins.

A 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.

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 with a value that has been written by a previous write operation for the same variable and in the same transaction.

2) Doomed Consistency: Kapalka and Guerraoui [13], extended the notion of strict serialisability to include the concept that even aborted transactions should not access an inconsistent state of the memory and, which can be doomed in TM due to 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.
Here is an example will initially y=4, x=2

$$\begin{array}{ll} p: & R^t_p(y);\ \widehat{R}^t_p(y,4);\ R^t_p(x);\ \widehat{R}(x,4);\ W^t_p(z,1/(y-x)) \\ q: & W^s_q(y,6);\ ok;\ W^s_q(x,4);\ ok;\ tryCom^s_q;\ \oplus^s_q \end{array}$$

The divided-by-zero state appears clearly in this example when the value of x is changed by transaction s, where x-y=0 and z=1/0. Each transaction in tm_{spec} satisfies the

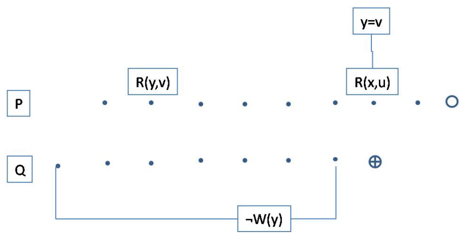


Fig. 5. Doomed consistency.

doomed consistency iff a later R operation 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, see Fig. 5.

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

C. Strict Serialisability

Papadimitriou [19] augmented the strength of serialisability by adding the requirement of the real time order of the committed transactions, see Fig. 6.

We formalize this property as follows: Let σ' be obtained from σ by serializing the concurrent committed transactions

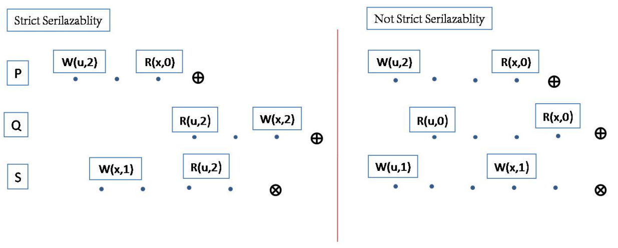


Fig. 6. Strict and Non-Strict serialisability.

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 a one block, we do not need to reorder between events to transfer the TH to STH. Instead, the events of each transaction can be collected by specifying the process and transaction for each event E^i_j.

Definition 1. *The TH can be strictly serialised, if we can obtain σ' from σ with respect to $Ser(TH)$ as follows:*

$$\begin{aligned} Ser(TH) \quad & \equiv (T^t_p;\ T^s_q) \\ & \equiv T^t_p \| T^s_q \\ & \quad \wedge \{\textit{The order of transactions over } \sigma' \textit{ is the} \\ & \qquad \textit{order of the committing events for the} \\ & \qquad \textit{same transactions } (fin\oplus^t_p;\ fin\oplus^s_q) \textit{ over } \sigma\} \\ & \quad \wedge \{\forall (R^t_p \wedge R^s_q) \textit{ over } \sigma \textit{ respects the ValidRead()}\} \\ & \quad \wedge \{\forall (\oplus^t_p \wedge \oplus^s_q) \textit{ over } \sigma \textit{ respects the ConflictDetRes()}\} \end{aligned}$$

VI. Testing of Abstract TM with Animation

To demonstrate, validate the correctness of the proposed abstract TM tm_{spec} and make such examinations for TM safety properties, we use tm_{spec} to execute one of the most highly studied concurrent data structures which is the lock-free FIFO queue [20], [1]. Many lock-free queue algorithms have been proposed such as compare-and-set (CAS). We will use an approach based on transactional memory, with some modifications, such as an added shared counter.

Testing example: A concurrent queue is an abstract data structure that consists of two parts. The producer part adds the element x to the rear terminal position, if the queue is not full. The consumer part retrieves the element from the front terminal position, if the queue is not empty. Consider the FIFO queue implementation shown in Fig. 7 and its ITL specification in Fig. 8 (because of lack of the space, we show just the producer part). It stores its elements in memory, which, for simplicity, we will assume a fixed queue size and two indices ($head = mem[0]$, $tail = mem[1]$). The first index points to the $head$ of the queue and the other points to the $tail$. Initially, both $head$ and $tail$ are equal and contain the location of the first room of the queue which equals 3 and the queue is empty. If the producer part, after reading $head$ and $tail$, finds that the queue is full, then it aborts the transaction. Otherwise, it will read and increment the shared counter (initially at $mem[2]$) at the point of memory entry $(tail - 1)$ and stores it at the memory entry $tail$, and then increments $tail$. If the consumer part, after reading $head$ and $tail$, finds that the queue is empty, by checking the equality of $head$ and $tail$, then it aborts the transaction. Otherwise, it will read the shared counter at the memory entry $head$, and then increment $head$.

```
proc Producer() $\equiv$
  $\{p_{head} = read(mem[head]);$
  $p_{tail} = read(mem[tail]);$
  if $(p_{tail} - p_{head} = Qsize)$
    then $Abort()$
    else
        $\{p_{shared} = read(mem[p_{tail} - 1]);$
        $write(mem[p_{tail}], p_{shared} + 1);$
        $write(mem[tail], p_{tail} + 1);$
        $tryCommit();\ \}$
        $\}$
```

Fig. 7. Producer part of concurrent queue algorithm

$$
\begin{array}{l}
\textit{PRODUCERspec} \mathrel{\widehat{=}} \\
P_p = free \wedge T_p^t = idle \wedge E_p^t = no_{ev} \wedge \\
((TranInvOp(p, t, R(head, \bot), \varepsilon, \varepsilon_r);\ TranResOp(p, t); \\
TranInvOp(p, t, R(tail, \bot), \varepsilon, \varepsilon_r);\ TranResOp(p, t); \\
QueueFullCheck())^*; \\
TranInvOp(p, t, R(p_{tail} - 1, \bot), \varepsilon, \varepsilon_r);\ TranResOp(p, t); \\
TranInvOp(p, t, W(p_{tail}, p_{shared} + 1), \varepsilon, \varepsilon_r);\ TranResOp(p, t); \\
TranInvOp(p, t, W(tail, p_{tail} + 1), \varepsilon, \varepsilon_r);\ TranResOp(p, t); \\
TranInvEnd(p, t, tryCommit, \varepsilon, \varepsilon_r);\ TranResEnd)^*
\end{array}
$$

Fig. 8. ITL specification of producer part

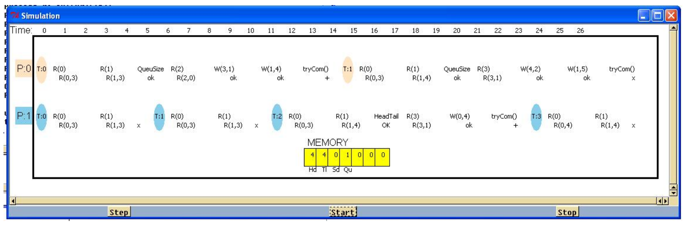
Animation: Some animation for our model is provided to make it more understandable and enable the reader to gain better insight into the TM system. The animator is written in Tcl/Tk [21] using Expect [22]. The Tempura file is accompanied by a Tcl/Tk file which defines the graphics. To execute the concurrent queue algorithm using the tm_{spec} executable specification and for seeking simplicity, just two concurrent processes are used to represent the producer and consumer parts. Also, two additional functions are used, *QueueFullCheck()* for the producer part that checks if the queue is full, and *QueueEmptyCheck()* for the consumer part that checks if the queue is empty. As shown in Fig. 9, the user interface for the graphical output is divided into four parts: Firstly, the timer grade which represents the number of state. Secondly, two processes where p_0 represents the producer specification and p_1 represents the consumer specification. Thirdly, the memory block. Finally, A space for showing the transaction number and its sequence of operations and responses. The graphical animation has facilities to execute this example step by step. So, we can notice clearly the memory synchronization in the output concurrent execution between the two processes and the validation of TM safety conditions.

VII. Conclusion

In this paper we present an executable specification for an abstract TM model and some TM correctness properties using ITL and AnaTempura. Moreover, a validation for this specification is presented using a concurrent data structures example. In reality, we are developing a formal framework which allows us to verify, analyse and capture the behaviour of new TM systems. In addition, we are working on developing refinement rules for verifying that a TM system satisfies the specification of the verified abstract TM.

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.



```
Time: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
P:0 T:0 R(0)     R(1)     QueuSize R(2)     W(3,1) W(1,4) tryCom() T:1 R(0)     R(1)     QueuSize R(3)     W(4,2) W(1,5) tryCom()
          R(0,3)   R(1,3) ok         R(2,0)   ok     ok     +            R(0,3)   R(1,4) ok         R(3,1)   ok     ok     x
P:1 T:0 R(0)     R(1)       T:1 R(0)     R(1)       T:2 R(0)     R(1)     HeadTail R(3)     W(0,4) tryCom() T:3 R(0)     R(1)
          R(0,3)   R(1,3) x       R(0,3)   R(1,3) x       R(0,3)   R(1,4) OK         R(3,1)   ok     +            R(0,4)   R(1,4) x
MEMORY
| 4 | 4 | 0 | 1 | 0 | 0 | 0 |
  Hd  Tl  Sd  Qu
```

Fig. 9. TM abstract tm_{spec} animation output

[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] W. N. S. I. Virendra J. Marathe and M. L. Scott., "Adaptive software transactional memory." in *In DISC 05: Proceedings of the nineteenth International Symposium on Distributed Computing*. LNCS,Springer, Sep 2005.

[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] 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.

[13] 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.

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

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

[16] 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.

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

[18] B. Moszkowski, "Reasoning about digital circuits," Ph.D. dissertation, Department of Computer Science, Stanford University, 1983.

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

[20] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed. San Francisco: CA: Morgan Kaufmann, 2003.

[21] P. P. in Tcl and Tk, *B.Welch*, 2nd ed. New Jersey: Upper Saddle River, 1997.

[22] D.Libes, *Exploring Expect*. OReilly and Associates, 1995.