Studying and Analysing Transactional Memory Using Interval Temporal Logic and AnaTempura

PhD Thesis

Amin Mohammed El-kustaban

Software Technology Research Laboratory
Faculty of Technology
De Montfort University
United Kingdom

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

May 2012

Declaration of Authorship

I, AMIN EL-KUSTABAN, declare that this thesis titled Studying and Analysing Transactional Memory Using Interval Temporal Logic and AnaTempura and the work presented in it are my own and original. It is submitted for the degree of Doctor of Philosophy at De Montfort University. The work was undertaken between July 2008 and February 2012.

To my lovely family..

Abstract

Transactional memory (TM) is a promising lock-free synchronisation technique which offers a high-level abstract parallel programming model for future chip multiprocessor (CMP) systems. Moreover, it adapts the well established popular paradigm of transactions and thus provides a general and flexible way to allow programs to read and modify disparate memory locations atomically as a single operation. In this thesis, we propose a general framework for validating a TM design, starting from a formal specification into a hardware implementation, with its underpinning theory and refinement. A methodology in this work starts with a high-level and executable specification model for an abstract TM with verification for various correctness conditions of concurrent transactions. This model is constructed within a flexible transition framework that allows verifying correctness of a TM system with animation. Then, we present a formal executable specification for a chip-dual single-cycle MIPS processor with a cache coherence protocol and integrate the provable TM system. Finally, we transform the dual processors with the TM from a high-level description into a Hardware Description Language (VHDL), using some proposed refinement and restriction rules. Interval Temporal Logic (ITL) and its programming language subset AnaTempura are used to build, execute and test the model, since they together provide a powerful framework supporting logical reasoning about time intervals as well as programming and simulation.

Acknowledgements

All thankfulness to God, the Beneficent, the Merciful, the One, on who all depend, and none is like Him.

Gratitude and many thanks must be expressed to :

- My first supervisor Dr. Ben Moszkowski for his constructive criticism, experienced guidance during the preparation of this thesis.
- My second supervisor Dr. Antonio Cau for hours of discussions and the patience with which he checked and corrected many technical errors.
- My advisor Professor Hussein Zedan for his inspirational leadership and advice.
- My family for their encouragement and full support during the preparation of this thesis.
- All our colleagues at the STRL for the valuable discussions during all these years.
- My friend Mr. Moussa Barkhadleh for his help and support throughout the period of my study.

Publications

1. Amin El-kustaban, Ben Moszkowski and Antonio Cau . Formalising of Transactional Memory using Interval Temporal Logic (ITL). To appear in *Proceedings of the Spring World Congress on Engineering and Technology (SCET 2012)* , IEEE , Xi'an, China, May 2012.

2. Amin El-kustaban, Ben Moszkowski and Antonio Cau. Specification Analysis of Transactional Memory using ITL and AnaTempura. In *Proceedings of The International MultiConference of Engineers and Computer Scientists 2012 (IMECS'12)*, pp176-181, Newswood Limited,Hong Kong, March 2012.

Contents

List of Figures

List of Tables

List of Abbreviations

CDP	**C**hip **D**ual **P**rocessor
CMP	**C**hip **M**ultiprocessor
FIFO	**F**irst **I**n **F**irst **O**ut
HTM	**H**ardware **T**ransactional **M**emory
ILP	**I**nstruction **L**evel **P**arallelism
ITL	**I**nterval **T**emporal **L**ogic
LT	**L**oad **T**ransaction
LTX	**L**oad **T**ransaction e**X**clusive
MESI	**M**odified **E**xclusive **S**hared **I**nvalid
MIMD	**M**ultiple **I**nstructions **M**ultiple **D**ata
MIPS	**M**illions **I**nstructions **P**er **S**econd
SMP	**S**ymmetric **M**ulti-**P**rocessor
ST	**S**tore **T**ransaction
STH	**S**equential **T**ransactional **H**istory
STM	**S**oftware **T**ransactional **M**emory
TH	**T**ransactional **H**istory
TM	**T**ransactional **M**emory

VHSIC **V**ery **H**igh **S**peed **I**ntegrated **C**ircuit

VHDL **V**HSIC **H**ardware **D**escription **L**anguage

VLSI **V**ery **L**arge **S**cale **I**ntegration

UMA **U**niform **M**emory **A**ccess

Chapter 1

Introduction

1.1 Motivation and Problem Statement

The technology revolution in Very Large Scale Integration (VLSI) has enabled today's researchers to design and implement Chip Multiprocessor (CMP), where two or more processors with a shared memory are integrated on a single chip. In actual fact, CMP or multi-core has become the mainstream architecture for microprocessor chips. In the next few generations, the number of processors that can be implemented on a single chip will significantly increase [1]. Consequently, parallel programs are required in order to gain the full features of multiple processors. 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 [2–4].

The memory synchronisation which deals 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 [5, 6]. Although lock-free and wait-free techniques could be used to avoid the problems with locks, at present they are still too complex to use and compose [7].

Transactional memory (TM) is a promising lock-free technique that can avoid lock-based problems and offer a high-level abstract parallel programming model for future CMP systems. In addition, TM can simplify parallel programming by transferring the burden of correct synchronisation from a programmer to a compiler and/or hardware. Moreover, it adapts the popular well established paradigm of transaction, thus providing a general and flexible way of allowing parts of a program to atomically read and modify disparate memory locations as a single operation, independently of others, while executing tasks concurrently [8, 9].

There have been several recent proposals on how to implement the TM in hardware [6, 10, 11], software and hybrid hardware-software combinations [12, 13]. However, a formal underpinning encompassing the specification, design, and implementation of TM still needs much effort. In addition, 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 [7, 14].

1.2 Research Objectives

The main aim of this investigation is to develop a unified formal framework for specifying, validating, verifying and implementing a TM system using a single well-defined formalism that

can capture the concurrent transaction's behaviour and reason about TM safety properties in a uniform manner.

To achieve this main aim, the following objectives are required:

- Produce a specification of an abstract transactional memory model and its executable version for validation.
- Develop a verification technique to proof the correctness satisfaction of a transactional memory model.
- Produce a specification of a shared memory environment.
- Develop a transactional memory design in a hardware description language.

The novelty of our approach is that it can correctly develop a TM system starting from a high-level specification which can then be transformed by a sequence of refinement steps down to a low-level hardware implementation, all in a single logical formalism, namely Interval Temporal Logic (ITL) [15–17], its executable subset, Tempura, and its simulation and the animation tool, AnaTempura [17, 18].

1.3 Research Methodology

The adopted research methodology follows the constructive research approach. The constructive method refers to contribution to knowledge being developed as a new solution for identified

problem. We develop a formal framework for known problems which are the formal specification, verification and implementation of transactional memory. The methodology of the proposed approach is made up of four steps as follows:

- Step 1: Background review

  The research study starts with a critically review of published work on the following: Firstly, shared memory environment and memory synchronisation techniques. Then, transactional memory both in term of definition and realisation. Finally, proposed formal frameworks of specification and verification of transactional memory. This review serves for the identification of our research aim. Moreover, it serves the purpose of understanding all approaches related to the research problem. A comprehensive study of previous work helps to recognise their weakness and boundaries.

- Step 2: Architecture

  This research stage concentrates on the design of the framework. The main components of the framework, the relation between them and how they can serve the research aim are identified. Moreover, the logical formalism and the main techniques that are used in the proposed framework are described in this stage.

- Step 3: Computational model and TM properties

  This stage of investigation focuses on producing an abstract transactional memory model. This model serves as a basis to our research aim. In this stage, the standard TM safety properties and other TM criteria are discussed. Reasoning about different TM aspects

helps to develop a general and flexible abstract TM model. In addition, the TM safety conditions are required for the correctness verification and validation of the abstract TM model.

- Step 4: Evaluation

  This step presents the capability of the proposed framework and its components in validation and verification of a chosen TM system from the literature.

1.4 Success Criteria

In order to measure the success of our research, the following success criteria are formulated:

- The formal specification of TM safety properties.
- The simplification of the formal verification for TM.
- The capability of the validation process in the proposed approach using ITL framework.
- The realisation capability of the proposed approach. For example, the possibility to build a TM system from high-level specification to low-level hardware.

1.5 Thesis Outline

This thesis report is organised into 7 chapters. We now briefly summaries each chapter:

- Chapter 1 gives a short overview and outlines the motivations, research objectives and methodology, success criteria and structure of this thesis.

- Chapter 2 presents a comprehensive and original description of the most relevant aspects of the memory synchronisation concept and transactional memory. The chapter starts with a brief overview of the classification of multiprocessors systems, lock-free and lock-based techniques. Then, the transactional memory basics are presented in the following sections. Finally, the formalisation of transactional memory and related work are discussed.

- Chapter 3 shows the proposed framework design and its stages. In addition, the syntax and semantics of the Interval Temporal Logic (ITL), as the formal foundation of the propose framework, are given. At the end of the chapter, the relationship between the ITL and transactional memory is discussed.

- Chapter 4 proposes a computational model for an abstract transactional memory and formalises different properties of transactional memory. In this chapter, animation, through testing is illustrated using AnaTempura. Moreover, a correctness verification for the abstract model is illustrated as well.

- Chapter 5 describes a specification of the original hardware transactional memory system and the cache coherency protocol that is used as a conflict detection method by this system. The validation and verification for this specification are discussed at the end of this chapter.

- Chapter 6 explains the structure and the executable specification of a chip-dual-processor. In addition, the integration of this chip with the TM system, that is correctly proven in chapter 5, is shown. In this chapter, refinement and restriction rules for transferring the high-level specification to the low-level hardware language is described.

- Chapter 7 discusses the significant conclusion of this research and presents several major areas and new directions for future work.

Chapter 2

Background

2.1 Multiprocessor

A multiprocessor is a system consisting of multiple processing units connected via an interconnection network and the software needed to make the processing units work together. Multiprocessor can enhance the throughput of the computers by executing more than one program at the same time. Moreover, the execution time of individual programs can (sometimes) be improved by executing them with multiple processors [19]. In this section the classification of multiprocessor and some background about chip multiprocessor is presented.

2.1.1 Computer Architecture Taxonomy

The well-known classification of computer architecture was developed by Flynn in 1966 [20]. A concept called *stream of information* is used in his classification approach. There are two kinds of information flow into a processor which are data and instructions. In actual fact, Flynn defined the following four classes of computer architecture [20](see also [19, 21]):

- Single Instruction-stream, Single Data-stream (SISD): The (most) popular and conventional computer architecture in the last decades was a uniprocessor or Von Neumann architecture which is classified as SISD computer. The instructions on SISD systems are executed sequentially via only one Central Processing Unit (CPU). This class has been prevalent in the computer industry for over fifty years, and many programming languages (e.g., Pascal and C), compilers, operating systems and programming methodology are based on this class .

- Single Instruction-stream, Multiple Data-stream (SIMD): This class is considered as a model of parallel computing that consists of two parts: instructional unit to issues instructions, and multiple processing elements to execute the same operation on different data. The two types in this class are array processors and vector computers. The array processors consist of a set of identical processors each having a local data memory. Processors are connected in a network and synchronously perform the same operation in parallel. Vector computer contain pipelined vector elements. These elements allow operations on all vectors at the same time. The Cray Y-MP vector machine in the Cray processor family is a popular system of this type.

- Multiple Instruction-streams, Single Data-stream (MISD): In this category, different instructions can be executed on the same data at the same time. In actual fact, no practical MISD machine has been developed so far.

- Multiple Instruction-stream, Multiple Data-stream (MIMD): This class is also considered as a model of parallel computing that consists of multiple small processors and a global memory connected together via some interconnection network. Several operations are executed in parallel on different data. The shared memory of MIMD class allows each processor to read or modify any location of its space. In addition, all the processors can work on the solution for a common problem by using the global memory.

  The class of MIMD architectures can be divided into two categories based on the type of memory organisation: distributed memory systems and shared memory systems.

  Distributed Memory

  In distributed memory architecture, which are commonly called multi-computer, each processor as an associated individual memory and can only access its own memory. Communication between processors is implemented by sending a message between them. For this reason, these architectures are often called message-passing machines. Systems employing distributed memory architecture can have an unlimited number of processors; which is an advantage of this type.

  Shared Memory

  In shared memory MIMD systems, or multiprocessor, each processor can access any memory address. They communicate through a bus and cache memory controller. The

shared memory may reside in one place, or it may be physically distributed in such a way that a one or more processors owns a part of the shared memory. Since the shared-memory multiprocessor systems have a common characteristic which is the identical access time of the memory for each processor, these systems are called Uniform Memory Access (UMA) or Symmetric Multi-Processor (SMP).

2.1.2 Single-Chip Multiprocessor

After years of advancement in integrated-circuit processing technology, it becomes possible to fabricate single chips with 1 billion transistors. In the past, most microprocessors designers used the increased transistor budgets to build larger and more complex uniprocessor. However, several problems arose which have made this approach to microprocessor design difficult to continue [1, 3]:

- The microprocessor designers used additional transistors to extract more Instruction Level Parallelism (ILP) from programs in order to perform more work per clock cycle. These processors can extract ILP by finding non-dependent instructions that appear near each other in the program code. Unfortunately, there is only a finite amount of ILP present in any particular sequence of instruction. Consequently, instructions from the same sequence are interdependent. Furthermore, the increasing use of visualisation and multimedia applications tends to increase the number of active processes or independent threads instead of ILP.

- The microprocessor area increases with the core's complexity. Moreover, the complexity increases the design time and verification cost. In addition, they must be designed and verified as single large units.
- The microprocessor can require increasingly long cycle times.

Recently, researchers propose two ways of using the increasing gate density and cost of wires in advanced integrated circuit effectively: simultaneous multithreading and chip multiprocessor [1].

2.2 Memory Synchronisation

In a multiprocessor and parallel programs environment, several processes can be running concurrently. For example, these could be a window manager, anti-virus program, word-process program, and internet application. When the processes require access to shared memory, the problem of how to ensure correct behaviour and data consistency arises. Memory synchronisation techniques solve this with *lock-based* and *lock-free* concepts.

2.2.1 Lock-Based Techniques

Lock-based ones are the conventional way to synchronise processes accessing a shared object through mutual exclusion that was firstly proposed by Dijkstra in 1965 [22–24]. Mutual exclusion concept is based on a shared variable together with routines to atomically acquire and

release the lock and guarantees that no more than one process can exclusively access and modify a certain section of code at a time.

Lock-based techniques are implemented through a combination of software algorithms and low level hardware primitives support for a type of atomic read-modify-write operation such as Test And Set (TAS) and Fetch And Add (FAA). However, lock-based concepts have a number of well known drawbacks:

- Deadlock: This can appear when processes acquire locks while waiting for the releasing of locks held by other processes, so that no process can make progress (see Fig. 2.1).

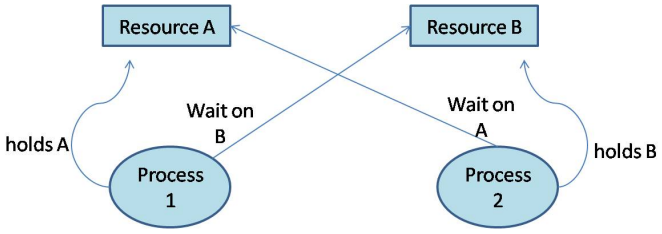

FIGURE 2.1: Deadlock problem.

- Convoying: This can appear when a process holding a lock enters into a delay situation such as an infinite loop, page fault or interrupt and blocks all other processes.

- Priority inversion: This appears when a high priority process is delayed and is waiting to acquire a lock held by a low priority process.

2.2.2 Lock-Free Techniques

Lock-free (non-blocking) concept addresses these problems by allowing multiple processes to read and modify shared data concurrently without corrupting it [25]. Lock-free techniques do not use mutual exclusion and therefore do not face the problems that locking can cause. They rely on hardware atomic primitives such as *Compare & Swap* (CAS) or the pair *Load-Linked Store-Conditional* (LL-ST) [26].

Load-Linked & Store-Conditional

This technique involves a pair of instructions that can be used to implement atomic operations to cache able memory location. The first instruction load-linked (LL) loads a memory location into a register. This can be followed by an arbitrary sequence of instructions not involving a memory operation. Then a second special instruction, store-conditional (SC), is used to store the same location. The SC only succeeds if no other processor has written to that register since when the LL instruction was last executed. Thus a successful SC indicates a successful read-modify-write operation to the memory location. If the SC fails, the entire operation must be retried. Success or failure of the store-conditional is indicated by condition codes. Microprocessor vendors whose support LL-SC technique, such as the MIPS family, advise the programmers to kept the number of instructions between LL and SC instructions small to reduce the probability of SC failure [27].

Implementations of lock-free methods do not block any process, even if some processes can be delayed, and guarantee that at least one process will make progress at any given time. However, lock-free implementations can exhibit starvation as the progress of other processes could cause one process to never finish [28].

Wait-freedom

Wait-free techniques are lock-free and prevent starvation as well. Every process is guaranteed to complete its task in a bounded number of steps [28]. A data structure is wait-free if and only if every operation on the structure completes after it has executed a finite number of steps, regardless of the execution speeds on other processes. Wait-free condition provides fault-tolerance: no process can be prevented from completing an operation by undetected halting failures of other processes, or by arbitrary variations in their speed [28]. However, wait-free techniques are more difficult to design and less efficient.

Obstruction-freedom

In a concurrent system, a non-blocking synchronisation algorithm is said to be obstruction free if and only if every operation on the structure is completed after executing a finite number of steps that do not contend with any concurrent operation for access to any memory location. Obstruction-freedom rules out the occurrence of deadlocks, but livelocks may occur if a group of processes abort each others' atomic operations and consequently no single one makes any progress [29].

2.3 Transactional Memory

Transactional memory (TM) is a promising lock-free technique that enables parts of a program to execute with atomicity and isolation, without regard to other concurrently executing tasks. Moreover, TM allows programs to read and modify disparate primary memory locations atomically as a single operation. In addition, TM supports lock-free implementations of complex data structures in a simple and efficient way [8].

2.3.1 Transaction notion

The concept of transactions is not new. Transactions have their roots in database systems and are commonly used in them, often defined as a series or list of actions. The actions that can be executed by a transaction include reads and writes of database objects [30]. Transactions allow a parallel program concurrent access and modification of shared data, yet still produce consistent, correct and deterministic results. The transaction notion is defined by the following four attributes of database transaction, known as ACID [5, 7, 30]:

- Atomicity: This ensures that either all of the operations in a transaction are executed successfully or none of them are. In other words, if one operation of the transaction fails, the entire transaction must fail without leaving behind any evidence that it has executed. A transaction that completes successfully, commits and one that fails aborts.

- Consistency: This property refers to the requirement that the data in database or memory should be in a consistent state. This means that if a transaction succeeds, only committed data will be stored permanently, else the old data before the change will be restored, leaving data in a predictable and consistent state.

- Isolation: This requires that execution of a transaction does not affect the result of concurrently executing transactions, and this result must be similar to a result in which these transactions are executed serially.

- Durability: This requires that once a transaction commits, its modifications to the data are stored on a durable media such as disk .

The difference between transactions in database and memory are the access time and the durability property. Data in database is stored on a disk rather than in memory which a much longer time to access is required. TM accesses main memory which cannot perform much computation at access time. The durability property is not important in TM since data in memory does not last after program terminates. This can simplify the TM implementation.

TM is proposed to avoid lock-based problems and simplify parallel programming by transferring the burden of correct synchronisation from a programmer to a compiler and/or hardware. However, not all memory synchronisations can be replaced by using transactions in a parallel program. Locking is often required to coordinate independent tasks, for example, by ensuring that one task waits for another to finish, or by limiting the number of processes performing a

task. Most TM techniques use busy-waiting in such situation because of aborts. This is inefficient since an aborted transaction rolls back its entire operation [7].

As an illustration of the transaction attributes, the following examples will be presented and used again in the execution and animation section for testing and validating our proposed model. These examples are based on standard ones for database [31, 32].

Example 1: Single Transaction

The transfer of money from one bank account A to another account B needs the following steps:

```
read(A)
write(A , A-100)
read(B)
write(B, B+100)
```

These steps will be composed into one transaction and executed as a single unit. The transaction correctness properties guarantee that a transaction either executes to completion or never happens at all.

Example 2: Bank Account

The same bank accounts are used but now more than one operation is performed at the same time. For example, 1000 is first deposited in an empty account A and then two transactions are made on this account at the same time, as follows: $T1$ transfers all of the 1000 from account

A to account B, and before the confirmation of the transfer is sent, $T2$ withdraws 100 from account A.

T1	
read(A)	
write(A, A-1000)	T2
read(B)	read(A)
write(B, B+1000)	write(A, A-100)
TryCommit()	TryCommit()

The transaction correctness properties guarantee that one of the two conflicting transactions will abort and appear as never having happened at all, and the other transaction will execute to completion and commit.

Example 3: Airline Reservation

Consider the example of an airline reservation system where multiple transactions can read the database at the same time. As an illustration, consider three transactions that are invoked in parallel from different terminals to make a reservation.

The first transaction needs to make a reservation for two seats together or not at all. The transaction starts reading, from the global database list, the number of seats that have status zero which means that these seats have not yet been reserved. Then, it writes one to the status of these seats to make them reserved. The second and third transaction needs to reserve just one seat each. So, they each read the number of some seat from the global database list that has

status zero and change it to one. The problem occurs where the seat number of the second or the third transaction is one of the two seat numbers that have been read by the first transaction.

T1	T2	T3
read(statusA)	read(statusA)	read(statusA)
write(statusA, 1)	write(statusA, 1)	write(statusA, 1)
read(statusB)	TryCommit()	TryCommit()
write(statusB, 1)		
TryCommit()		

Two situations appear in this example: the conflict between $T1$ and $T2$, and the conflict between $T3$ with $T1$ and $T2$. The correctness properties of the transaction system guarantee that just one of the three conflicting transactions will commit the change of a seat's status, with other transactions aborting and appearing as if they never happened at all.

2.3.2 Hardware, Software and Hybrid TM Implementations

Transactional memory can be implemented in hardware (HTM), software (STM) or as a hybrid hardware-software combination (HyTM) [6, 12, 13]. The maintenance and validation of read sets are considered the main overhead for software transactional memory systems. The hardware implementation improves the performance and reduces the program overhead. Nevertheless, the hardware transactional memory systems cannot support a large transactions because the limitation of cache capacity. The hybrid hardware-software transactional memory technique was proposed to address the limitation of hardware capacity [7].

Hardware Transactional Memory

Most proposals for HTM present two aspects of hardware systems that are strongly related to HTM. Firstly, hardware buffers such as a cache store speculative data (also known as memory consistency models). Secondly, cache coherency mechanism guarantee that multiple processors have a coherent view of locally cached data. HTM has some advantages over STM, such as high performance, lower overhead and better energy consumption [33]. The disadvantage of HTM is the size limitation of the transaction data set [7].

Many researchers have proposed a HTM such as TCC [10] and logTM [34] . However, Herlihy and Moss [6] wrote a widely cited paper that was the first hardware proposal to implement atomic read-modify-write disparate memory locations as a single operation (TM). Their approach incorporated a new transactional cache and instructions, and modified the cache coherency protocol and the snoopy bus arbitration.

Software Transactional Memory

Most recent work in STM systems, especially those integrated with a compiler has focused on reaching a level that makes them convenient for experimentation and prototyping. The performance of those systems has been enhanced by developing many programming techniques such as hashing methods, dynamic TM, conflict resolution policies and direct/deferred update. The advantages of STM are flexibility, modifiability and easy integration with existing programming

language [7]. Shavit and Touitou [35] wrote the first published paper to describe the implementation of software TM. The idea of this paper is that the concurrent objects being accessed by a transaction were pre-determined, preventing two transactions from deadlocking.

Hybrid Hardware/Software Transactional Memory

HyTM is a STM-based alternative to unbounded TM in a HTM. This approach proposes to overcome the disadvantages of limitation of HTM data sets, and complication of Unbounded TM (UTM) to be included in the multiprocessor chips as well as enhancing the performance of STM implementations using best effort HTM. The first proposed work on a HyTM model was by Lie in 2004 [36]. Lie avoids bounded transaction sizes in HTM by executing the transactions firstly in HTM, and if unsuccessful, it executes the transactions in STM. The results of this approach show that it can be easily integrated in existing hardware.

Chip Multiprocessor with Transactional Memory

To help researchers with fast software development and evaluation, many Chip Multiprocessor (CMP) have been implemented as prototypes, such as Stanford Hydra [37], Stanford ATLAS [2] and Berkeley RAMP [38]. However, the CMPs that have been implemented with hardware transactional memory are ATLAS [2] and RAMP [38].

Recently, researchers at Stanford University built ATLAS [2], the first prototype of a CMP with TM. The design has been mapped in a multi-FPGA board (Xilinx XC2VP70) and operates at

100MHz. The prototype includes 8 PowerPC cores and a ninth core that handles the operating system and input/output devices. ATLAS uses the TCC architecture for hardware-bus transactional memory as a simple mechanism to replace the complex cache coherency protocol. The data cache design is attached to the PowerPC cores through IBM's processor local bus, and has 32-byte cache lines that can be one, two, or four-way set associative (each way is 8 KB, resulting in cache sizes of 8, 16, or 32 KB). ATLAS was implemented to allow for fast software development and evaluation. It also allows the researchers to study the use of transactions in the operating system.

Researchers at Berkeley and Stanford built RAMP [38]. This prototype consists of eight CPUs with 32KB L1 data-cache with transactional memory support. The CPUs are hard coded PowerPC405 unit with emulated floating point units connected through the central control FPGA. A separate, 9th processor runs the operating system (PowerPC Linux). Like ATLAS, RAMP uses the TCC architecture for hardware-bus transactional memory. RAMP runs 100x faster than as simulator running on as Apple 2GHz G5 (PowerPC).

2.4 Formalisation of Transactional Memory

2.4.1 Motivation

Transactional memory (TM) is a an active research area and many recent works have proposed efficient implementation techniques to enhance its performance. Although some of the proposed

new ideas may improve the throughput of the TM, at the same time its correctness criteria can be lost in the process [39]. Even a minimal effort to formalise and verify of TM can detect the violation of TM correctness properties [40]. Guerraoui and Kapalka state that "Without such formalisation, it is impossible to check the correctness of these implementations" [41]. Thus, formalisation will help us to understand the new TM systems better, proving their correctness and classifying new policies [39].

Researchers have proposed various formal frameworks for proving that a TM implementation satisfies its specifications [42–45], but these are still hard to understand and use. In addition, most of these researchers assume the correctness criteria from database transactions (e.g. serialisability [30]). However, these criteria specify only some parameters of TM properties and do not clarify the semantics of conflict detections and contention management [7]. Recently, researchers have concentrated on generalising the correctness specification of TM for safe development of software on top of transaction memory [40].

2.4.2 Related Work

Earlier work on 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) [39, 41]).
- A compositional method for defining the TM semantics and proving that a transactional memory implementation satisfies its specifications (e.g. [40, 42, 45]).

Correctness of the TM systems

Scott [39] is the first to suggest sequential specifications to capture many semantics of transactional memory. The conventional notion of sequential histories (i.e. each invocation is immediately followed by its response) is considered in Scott's work and the transactional memory semantics are defined using these histories. Scott's approach has the following features:

- Transactional memory is modelled as mapping from objects to values.
- A sequential specification is defined that expresses the following requirements: Firstly, each read returns the right value in any successful transaction. Then, a commit succeeds if it ends an isolated transaction.
- The circumstances in which two transactions cannot both succeed are specified by presenting four practical policies for detecting conflicts: Lazy invalidation conflict, Eager W-R conflict, Mixed invalidation conflict and Eager invalidation conflict.
- Arbitration functions are provided to ensure progress of transactions.

Guerraoui and Kapalka [41] present a new safety condition called opacity to verify the correctness of a TM implementation and its graph characterisation. They extend the notion of serialisability (defined later in Subsection 4.3.3) to include the concept that aborted transactions should not access an inconsistent state of the memory, which can be doomed (can't finish successfully) in Software Transactional Memory (STM) (due to infinite loops, or exceptions). Their investigation followed these stages:

- Modelling a TM system, which is based on [46], in a formal way with their notion of opacity.
- Defining opacity as a safety property and deducing that the proposed TM model satisfies this property.
- Proving that opacity requires a lower complexity than other TM implementations which always observe a consistent state and impose an additional cost of per-operation validation.

Guerraoui and Kapalka [40, 43] extend this framework by handling non-transaction code with the opacity condition. In addition, they introduce a checker model using a strict serialisability with respect to opacity as a safety condition. However, two aspects are still missing from these papers; the nested transactions and the contention management strategies (i.e. when transaction should commit). These two aspects cannot be achieved by the opacity property alone. In addition, these works merely focused on the STM implementations and did not deal with the HTM and Hybrid implementations.

Compositional Method, for TM Verification

Cohen et al. [42] present a methodology, supported by tools, to formally verify that a TM implementation satisfies its specification. The notion of an admissible interchange of transaction operations is used in this work to model the approaches of conflict detection (which was characterized by Scott [39]) and to build a checker model. Their approach follows these stages:

- Proposing a general model for abstract TM, based on the model of fair discrete systems.
- Presenting proof rules, based on abstraction mapping, to verify that an implementation of a TM correctly specifies its abstract specification.
- Demonstrating the proof rule and the verification method by modelling TCC [10] in TLA+ [47] and proving its correctness with the model checker TLC [47].
- Extending the theorem prover TLPVS of [48] to obtain mechanical proof of correctness.

The methodology of Cohen et al. [42] is clear and the tools are well known. In addition, the abstract TM model is built upon strong safety conditions in [39]. Some of the loose ends had been subsequently completed (e.g. dealing with non-transaction operations) [49]. Others such as nested transactions have not been dealt with. Also, some safety conditions and conflict detection policies have been assumed such as read local consistency and mixed conflict detection. Moreover, there is no validation method for the specification of the proposed abstract TM model. In addition, its specification methods make the TM systems more abstract than real design and may miss many details as a result.

Tasiran [45] presents a compositional method for verifying software transactional memory implementations. His approach begins with previous work (e.g. [50]) on verifying that semantic-level descriptions ensure atomicity and serialisability. This previous work (which concentrates on the top algorithmic level and is called small-step semantics) is taken as a starting point, and then the Bartok STM implementation [51] that satisfies these properties is proven. The new addition of this work verifies that the algorithm-level description (actually programmed in $Java$ or

$C\#$) is correctly implemented by STM code. Assertions like those for sequential programs are used. This technique allows the properties required of the STM implementation to be checked using the $Spec\#$ language and the $Boogie$ verification tool. The OTFJ language is employed to model the program using software transactions and manual proof of correctness of STM implementation. It was a novel method in STM semantics and verification. Unfortunately, this work focuses only on the STM implementations and neglects the hybrid and HTM.

2.5 Summary

TM is a hot research area. It was developed to solve memory synchronisation problems in a shared memory environment such as a chip multiprocessor which is the mainstream architecture for microprocessor design. There have been several proposals for TM design and enhancement. However, designing TM without formalisation may violate its correctness.

In this chapter a comprehensive description of memory synchronisation concepts and transaction notations with examples are presented. In addition, the main kinds of TM implementation and some real existing chip multiprocessor with TM are illustrated. Moreover, the importance and benefits of TM formalisation are given. This chapter is concluded with brief descriptions and limitations of the recent works of TM's formalisation and its correctness verification methods.

Chapter 3

Preliminaries and Formal Logical Framework

3.1 Introduction

As we mentioned previously in Subsection 2.3.2, many new TM systems have recently been proposed in order to gain the full performance potential of multi-core systems. The correctness validation of some of these TM systems presently relys on micro-benchmarks and simulation tools without the use of formal proof techniques and validating in real shared memory environments [7]. As a result, correctness testing is not exhaustive. However, formal proof without using simulation in the design process is quite hard. Moreover, simulation provides powerful and more accessible tools for rapid prototyping and validating [52, 53].

A TM design framework soundly based upon formal techniques with support for simulation could improve reliability. Such a methodology should contain powerful logical operators to capture concurrent behaviours of transactions at specific points in time. In this chapter, we overview a comprehensive framework for specifying, validating, verifying and implementing a TM system in ITL work-bench.

3.2 Framework Design

Our proposed approach involves two main parts: The first part concerns the development of framework's main components which are a general and provable abstract TM model and TM safety properties. This model can serve as a basis for verifying the correctness of a hardware or software transactional memory system. However, we focus here only on the hardware transactional memory systems and leave the other TM types for the future work. The second part concerns the verification of the correctness of a TM system regarding the provable TM model from the first part and then transforming it from a high-level specification to a low-level hardware implementation.

The validation process in our framework is for a testing the proposed TM model by executing real examples and using simulation with animation. In actual fact, this stage has many features: First, it helps us to get the specification right. Second, it gives initial indicators for satisfying TM safety conditions. Finally, it makes the proposed TM model more understandable and enable the reader to gain better insight into this model.

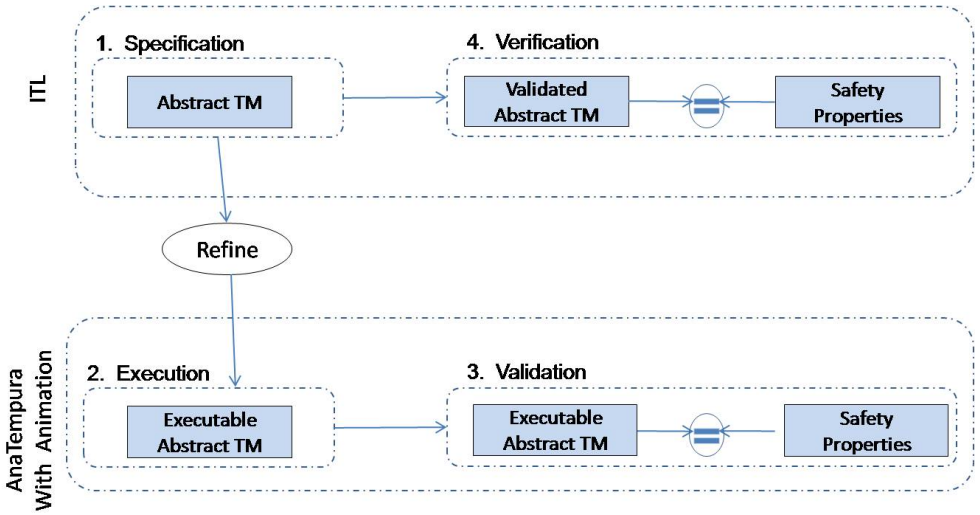

FIGURE 3.1: Part 1 of framework.

As shown in Fig. 3.1, the development of the framework's main parts involves four steps: First, a high-level abstract specification for a TM model and standard TM safety properties are expressed in ITL. Second, an executable version of the abstract TM model is refined from the first step by using a sound refinement calculus that can transform the ITL specification into a set of modules in Tempura (an executable subset of ITL) [18, 54]. Third, a validation for the abstract TM specification is simulated and animated by executing real examples on the executable TM version using AnaTempura. Steps one to three are repeated until the abstract TM model meets most of the TM aspects without violating the standard TM properties. However, the validation step is not enough to prove the correctness of the TM model, which requires that all possible behaviours of the TM model satisfy the properties, but does help to simplify the formal verification step. Fourth, the validated specification of the abstract TM model is formally proven against the TM safety properties by using propositional reasoning, ITL inference rules and the definition of ITL operators.

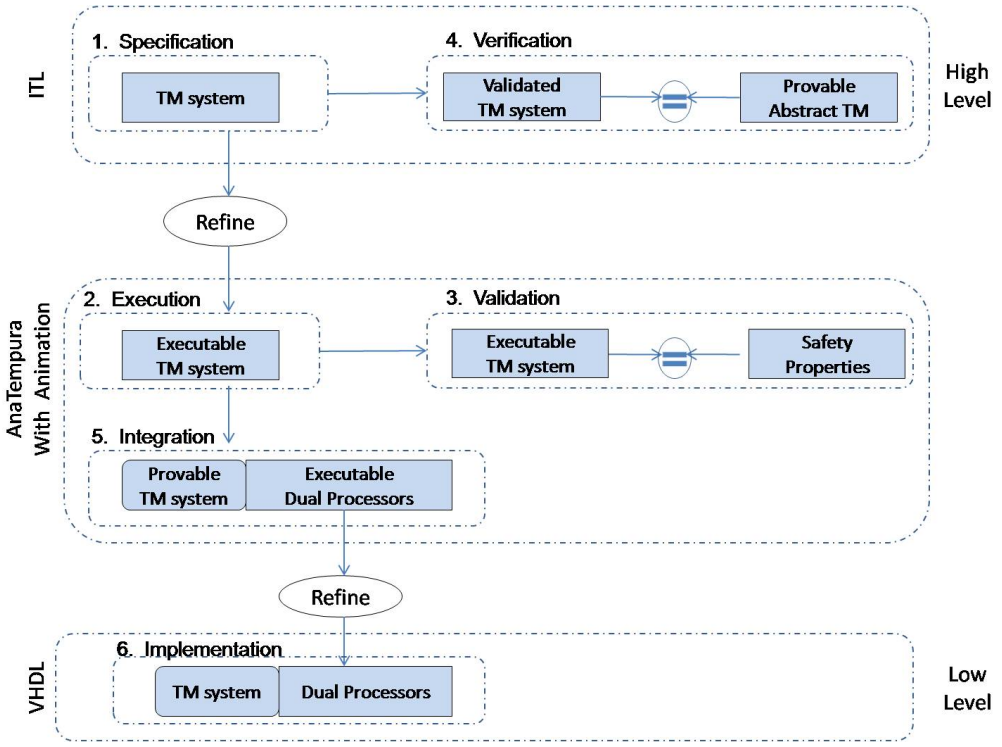

FIGURE 3.2: Part 2 of framework.

When we specify and validate the specification of a particular TM system, the first three steps of our framework's first part are imported in the second part, as shown in Fig. 3.2. However, the abstract TM model is replaced in the second part with a TM system. As soon as we get the right specification of the TM system, it is formally verified against the abstract provable TM model by using a refinement mapping technique in step 4. In actual fact, we propose to prove the TM system against the abstract TM model instead of the TM safety properties in order to simplify the formal verification step. To increase the degree of confidence and make a real evaluation, we then add the integration step to combine the provable TM system with a shared memory environment such as Chip Dual Processor (CDP). Finally, the CDP with the TM system is refined into a hardware description language such as VHDL using proposed

refinement and restriction rules. Once we obtain a description of the hardware, a commercially available synthesis tool can be used to produce a netlist which can then be implemented in silicon [52].

3.3 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, unlike most temporal logics, has the capability of handling both sequential and parallel composition. A powerful and extensible specification framework is also offered by ITL for reasoning about properties involving safety, liveness and projected time. In addition, Tempura and AnaTempura provide an executable framework with animation for experimenting and developing ITL specification [15, 17, 18, 55, 56].

3.3.1 Syntax of ITL

The syntax of ITL (integer expressions and first order formulae) is defined in Table 3.1, 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 3.1: Syntax of ITL

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

3.3.2 Semantics of ITL

Time is modelled as finite and infinite sequence of states 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.

We first describe the semantics informally and then give a rigorous definition of ITL's operators.

Informal Semantics

All formulae are evaluated over the whole interval. For example, $f_1 \wedge f_2$ is true over σ, iff f_1 and f_2 are true over σ. Similarly $\forall$ represents the universal quantifier. Here is the informal semantics of the various useful ITL constructs:

- $skip$: 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.

Formal Semantics

Let us assume the truth-values tt and $\not{f}\not{f}$ are associated with true and false, respectively. We also write $\sigma \sim_v \sigma'$ to denote that the intervals σ and σ' are identical with the possible exception of their mappings for the variable v. Moreover, let $\sigma = \sigma_0\sigma_1\sigma_2 \ldots$ be an interval and:

- A prefix interval of σ is $\sigma_0 \ldots \sigma_k$ (where $0 \leq k \leq |\sigma|$)

- A suffix interval of σ is $\sigma_k \ldots \sigma_{|\sigma|}$ (where $0 \leq k \leq |\sigma|$)

- A subinterval of σ is $\sigma_k \ldots \sigma_l$ (where $0 \leq k \leq l \leq |\sigma|$)

In addition, let $\mathcal{M}[\![\ldots]\!]$ be the meaning (semantic) function from formulae to $\{tt, \not{f}\not{f}\}$ then:

$\mathcal{M}_\sigma[\![\neg f]\!]$ =tt *iff* not $(\mathcal{M}_\sigma[\![f]\!] = tt)$

$\mathcal{M}_\sigma[\![f_1 \wedge f_2]\!]$ =tt *iff* $(\mathcal{M}_\sigma[\![f_1]\!] = tt)$ and $(\mathcal{M}_\sigma[\![f_2]\!] = tt)$

$\mathcal{M}_\sigma[\![skip]\!]$ =tt *iff* $|\sigma| = 1$

$\mathcal{M}_\sigma[\![\forall v \cdot f]\!]$ =tt *iff* for all σ' s.t. $\sigma \sim_v \sigma'$, $\mathcal{M}_{\sigma'}[\![f]\!] = tt$

$\mathcal{M}_\sigma[\![f_1;\ f_2]\!]$ =tt *iff* (exists a $k \leq |\sigma|$, such that

$(\mathcal{M}_{\sigma_0 \ldots \sigma_k}[\![f_1]\!] = tt)$ and $(\mathcal{M}_{\sigma_k \ldots \sigma_{|\sigma|}}[\![f_2]\!] = tt)$)

or (σ is infinite and $(\mathcal{M}_\sigma[\![f_1]\!] = tt$))

$\mathcal{M}_\sigma[\![f\,*]\!]$ =tt *iff* if σ finite

then (exist $n \geq 0$,$l_0, \ldots, l_n$ s.t. $l_0 = 0$ and $l_n = |\sigma|$ and

for all $0 \leq i < n, l_i < l_{i+1}$ and $(\mathcal{M}_{\sigma_{l_i} \ldots \sigma_{l_{i+1}}}[\![f]\!] = tt)$)

else (exist $n \geq 0$, $l_0, \ldots, l_n$ such that $l_0 = 0$ and

$\mathcal{M}_{\sigma_{l_n} \ldots \sigma_{|\sigma|}}[\![f]\!] = tt$ and

for all $0 \leq i < n, l_i < l_{i+1}$ and $(\mathcal{M}_{\sigma_{l_i} \ldots \sigma_{l_{i+1}}}[\![f]\!] = tt)$)

or

(exist an infinite number of l_i such that $l_0 = 0$ and

for all $0 \leq i, l_i < l_{i+1}$ and $(\mathcal{M}_{\sigma_{l_i} \ldots \sigma_{l_{i+1}}}[\![f]\!] = tt)$)

3.3.3 Derived Construct

The following ITL derived constructs will be used for simplicity.

- The predicates true and false: $true \ \widehat{=}\ 0 = 0$ and $false \ \widehat{=}\ \neg true$.

- The logical disjunction: $f_1 \vee f_2 \;\widehat{=}\; \neg(\neg f_1 \wedge \neg f_2)$.
- The logical implication: $f_1 \supset f_2 \;\widehat{=}\; \neg f_1 \vee f_2$.
- The equivalence: $f_1 \equiv f_2 \;\widehat{=}\; (f_1 \supset f_2) \wedge (f_2 \supset f_1)$.
- The existential quantifier: $\exists v.f \;\widehat{=}\; \neg\forall v.\neg f$.
- The infinite and finite interval: $inf \;\widehat{=}\; true;\ false$ and $finite \;\widehat{=}\; \neg inf$.
- The next: $\bigcirc f \;\widehat{=}\; skip;\ f$.
- The more and empty: $more \;\widehat{=}\; \bigcirc true$ and $empty \;\widehat{=}\; \neg more$.
- Some useful operators in the following table:

TABLE 3.2: ITL derived constructs

$\Diamond f$	$\widehat{=}$	$finite;\ f$	Eventually.
$\Box f$	$\widehat{=}$	$\neg\Diamond\neg f$	Henceforth.
$\diamondsuit f$	$\widehat{=}$	$\Diamond(f;\ true)$	Some subinterval.
$\boxed{\text{a}} f$	$\widehat{=}$	$\neg\diamondsuit\neg f$	All subintervals.
$\diamondsuit f$	$\widehat{=}$	$\Diamond(more \wedge f)$	Some nonempty subinterval.
$\boxed{\text{m}} f$	$\widehat{=}$	$\Box(more \supset f)$	All nonempty subintervals.
$\diamondsuit f$	$\widehat{=}$	$(f \wedge finite);\ true$	Some finite prefix.
$\boxed{\text{f}} f$	$\widehat{=}$	$\neg\diamondsuit\neg f$	All finite prefix.
$fin\ f$	$\widehat{=}$	$\Box(empty \supset f)$	Final state.
$halt\ f$	$\widehat{=}$	$\Box(empty \equiv f)$	Exactly in the final state.

3.3.4 Applications

Interval Temporal Logic and its executable subset Tempura have been applied to specify and verify behavioural conditions of diverse kinds of systems. Example include a bomb disposal control robot system, hardware/software co-design, a large scale hardware system, security and trust policies [18, 57–59]. Some of these applications and others related to this research will be presented in this section.

The term *reactive systems* refers to a variety of types of concurrent and real time systems, which do not necessarily terminate and usually contain a number of parallel actions. ITL is suitable for the specification of reactive systems. Cau et al. [60] present a compositional formal framework for modelling general systems and its suitability to Information Systems. They use ITL to compositionally model an information system, which can be regarded as a reactive system.

One of the primary challenges *chip design* faces today is the validation and simulation of increasingly complex systems. Throughout the 1990s, formal specification and verification has become as a promising complement to conventional simulation. Although there are many formal methods for the specification and the verification of hardware, ITL is developed for hardware verification. Coleman et al. [58] describe some benefits of ITL and Tempura in an application that is relevant to specifying, verifying and designing a large scale hardware. They develop an ITL specification and simulation of a general-propose multithreaded dataflow computer known as EP/3. In addition, they suggest some solutions for problems encountered during the specification of the EP/3 such as a missing data structure and the way to represent inter processor communication in ITL.

Mixed hardware/software systems (heterogeneous systems) are fast gaining popularity because of the benefits to performance, cost, power consumption and size. The big challenges of this system are the design and analysis. Hybrid hardware/software transactional memory is an example of this approach. Zedan and Cau [61] propose a single logic framework with a supporting tool, AnaTempura, for hardware/software co-design. ITL and its executable subset Tempura are used in this approach to perform validation and analysis system's behaviours of interest compositionally within a single logical framework. The ability to capture and validate subsystem properties, which is the main topic of the work, is performed by inserting assertion points at suitably chosen places in the code to divide it into several code-chunks. Then the properties of interests are validated over this behaviour. Voice over IP is presented as an application for this framework.

3.3.5 Justification of ITL for TM

As we already noted, our framework for TM is based on ITL. Our selection of ITL is justified as follows:

- Transactions can be regarded as a set of intervals each with a specific beginning and ending state. A transaction in the history of concurrent transactions can be expressed and reasoned about in a compositional way using ITL's operators for subinterval such as $\diamondsuit$ and $\boxed{a}$.

- The order of transactions with their relevant operations is significant for ensuring the conflict-free property (see Section 4.3). The $skip$ and $chop$ operators are well suited for formalising this, thus demonstrating that ITL can easily handle transactions ordering.

- The specification of the end time of each committed transaction is also important for verifying the strict serialisable safety condition (see Section 4.3). The ITL's fin and $chop$ operators can be used for this purpose.

3.4 Tempura and Refinement

3.4.1 Tempura and AnaTempura

Tempura is developed by Ben Moszkowski as a programming language based on temporal logic [17]. Tempura provides an executable framework for suitable ITL specifications of digital circuits, parallel programs and other dynamic systems. One of the main features of Tempura is its similarity with conventional imperative programming languages. For example, Tempura has as in-place assignment. Moreover, it contains iteration constructs such as a *While* statement. However, there are some differences that let the dealing with Tempura is travail such as the length of a formula interval and the values of the variables (in the formula and throughout the interval) should be specified before executing the formula.

AnaTempura is developed as an integrated workbench for ITL that offers specification, validation and run-time verification in the form of simulation. In addition, it provides a powerful

visualisation function to enhance the ease of using the tool. Moreover, it supports animation of the system execution's behaviour as well as draws timing diagrams during run time which are helpful into analysis of the system behaviour. AnaTempura consists of two main parts which are the Tempura Interpreter and Monitor. The Tempura Interpreter is used to execute Tempura files. The Monitor allows users to analyse the program at run-time with respect to a specification [62].

3.4.2 Refinement of ITL into Tempura

To transform an ITL abstract system specification into Tempura code, we use a set of sound refinement laws that have been derived in [18, 54]. The refinement relation $\sqsubseteq$ is defined on a system: A system X is refined by the system Y, denoted $X \sqsubseteq Y$, if and only if $Y \supset X$. The following are some useful example of refinement rules:

- The conditional *If-Then-Else* is introduced with the following rule.

$$\text{(if-1)} \quad (f_0 \wedge f_1) \vee (\neg f_0 \wedge f_2) \quad \sqsubseteq \quad \text{if } f_0 \text{ then } f_1 \text{ else } f_2$$

- The characteristics of the chop construct (;) rule are described as follows:

$$\begin{aligned}
&\text{(;-1)} \quad empty;\ f && \equiv \quad f \equiv f;\ empty \\
&\text{(;-2)} \quad (f_0;\ f_1);\ f_2 && \equiv \quad f_0;\ (f_1;\ f_2) \\
&\text{(;-3)} \quad f_0;\ (f_1 \vee f_2);\ f_3 && \equiv \quad (f_0;\ f_1;\ f_3) \vee (f_0;\ f_2;\ f_3)
\end{aligned}$$

- The *While* formula and the non-terminating loop rules are introduced as follows:

$$\begin{array}{lllll} \text{(while-1)} & (f_0 \wedge f_1)^* \wedge fin(\neg f_0) & \sqsubseteq & \text{while } f_0 \text{ do } f_1 \\ \text{(while-2)} & {f_0}^* \sqsubseteq {f_0}^* \wedge inf & \equiv & \text{while } true \text{ do } f_0 \end{array}$$

- Some other formulae rules are introduced as follows:

$$\begin{array}{lllll} \text{(repeat)} & f_0; (\neg f_1 \wedge f_0)^* \wedge fin(f_1) & \equiv & \text{repeat } f_0 \text{ until } f_1 \\ \text{(assignment)} & \bigcirc A = exp & \equiv & A := exp \end{array}$$

3.4.3 Refinement Mapping Technique

To prove that a TM system satisfies specification of an abstract one, we use the abstract mapping method of Abadi and Lamport [63] which is described as follows:

Let C and A be two systems that denote respectively $concrete$ and $abstract$ systems. According to the refinement mapping technique, a specification of system C implements a specification of system A (equivalently C refines A), denoted $C \sqsubseteq_F A$, iff every observable behaviour allowed by C is also allowed by A (possibly with stuttering which means that C may require several steps to match A). In other words, to verify that $C \sqsubseteq_F A$, it suffices to prove that if C allows the behaviour $((e_0, x_0);\ (e_1, x_1);\ \dots)$ and A allows the behaviour $((e_0, y_0);\ (e_1, y_1);\ \dots)$ (where e is a state of the externally visible component, while x and y are internal states) then there exists a function F such that $F(e_i, x_i) = (e_i, y_i)$ preserves the state machine behaviour and liveness.

Let us consider the specification of $C = (S_c, I_c, N_c, L_c)$ and $A = (S_a, I_a, N_a, L_a)$ where S, I, N and L denote the following:

- S: A state space.
- I: The set of all initial states.
- N: The next-state relation (a transition relation).
- L: A supplementary property of the specification which represents the fairness constraints on the abstract model such as asserting that certain actions must eventually occur, ensuring the liveness property that operations that should complete eventually do complete.

Then $C \sqsubseteq_F A$ can be established iff we can reason about the following refinement mapping:

- R_1: $\forall\ s \in S_c$: $F_e(s) = s$, F_e preserves the externally visible state component.
- R_2: $F(I_c) \subseteq I_a$, F takes concrete initial states into abstract initial states.
- R_3: if $(s_0;\ s_1) \in N_c$ then $(F(s_0);\ F(s_1)) \in N_a$, A state transition allowed by C is mapped by F into a [possibly stuttering] transition allowed by A.
- R_4: $F(L_c) \subseteq L_a$, where P_c is the liveness property defined by C, F maps the states behaviour allowed by C into the states behaviour that satisfy A's supplementary property.

This lead to the following theorem:

Theorem 3.1. *If there exists a refinement mapping from C to A, then $C \sqsubseteq_F A$.*

3.5 Summary

In this chapter, we presented an overview for the proposed TM formal framework. The foundation of the uniform formal TM framework is ITL. Its formal syntax and semantics are given. Our reasons for using ITL and its executable subset AnaTempura are discussed as the relationship between the ITL and transactional memory. We concluded the chapter with some of the refinement rules of ITL into Tempura and overview a refinement mapping technique.

Chapter 4

Abstract Model of Transactional Memory

4.1 Introduction

Transactional memory designs have varying criteria and aspects that may define the programming model and performance of a given TM system such as its conflict detection and resolution policies. In addition, the safety properties in proposed transactional memory systems are quite diverse and differ. So the formal verification of different TM systems needs as a reference a general, flexible and provable abstract TM model which can support most of these features and properties.

As shown in the framework's main parts in Fig. 4.1, this chapter focuses on development of a provable abstract TM model and TM safety properties. We propose a computational model for

an abstract TM and a formalisation for the standard safety properties discussed by the TM community using ITL and its executable subset AnaTempura. Since altogether, ITL and AnaTempura provide a powerful framework supporting logical reasoning about time intervals as well as programming and simulation.

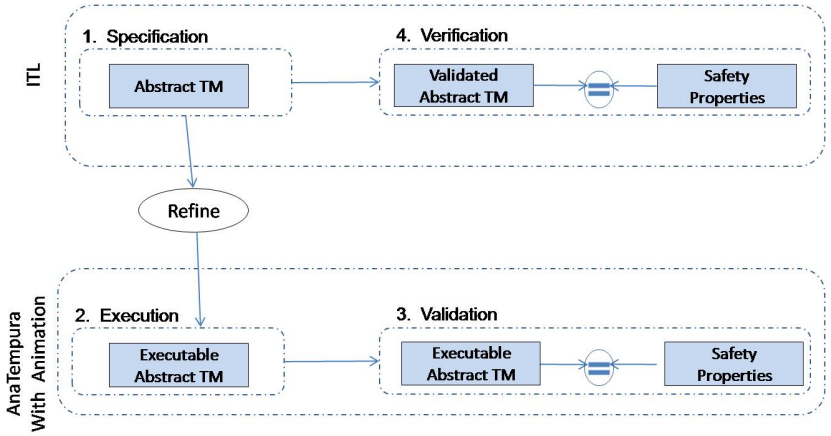

FIGURE 4.1: Framework's main part.

We first give a high level specification of a general TM abstract model and its safety conditions which are based on well-known published papers on generalising the safety of TM such as [39, 41, 42]. The generality of the proposed TM model gives us the capability to use it as a standard and match it to different TM systems.

One advantage of AnaTempura that it can be used to validate our proposed TM specification model before we prove properties about the model. In fact, this helps us to get the right specification. We then verify that the proposed TM specification model satisfies these safety conditions using a mathematical proof.

4.2 Computational Model for TM

In this section, we present an abstract model to specify TM similar to [39, 41, 42]. 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 just assumed in work by others [42].

Table 4.1 explains some terms used in the following sections which have not been introduced yet.

TABLE 4.1: Glossary Table

Term	Definition
Commit	A transaction successfully completes and all the temporary updates by this transaction are made visible to other transactions.
Abort	A transaction fails and discards any updates.
Conflict	There are two concurrent transactions accessing (and at least one modifies) the same object(s) and one of them needs to abort.
Doomed Transaction	A transaction has a conflict with another transaction. It may continue to execute new read and write events, but it must eventually abort.
Doomed Consistency	A TM safety property used to ensure that even the doomed transactions do not observe an inconsistent state.
Inconsistent read state	A state when a read operation responds with i.e., a value that may cause illegal actions such as an infinite loop or divided by zero.

Processes and Transactions

An interval σ is a finite or infinite sequence of one or more states $s_0, s_1, s_2, ...$ Each state has concurrent observable events E. Each such event E_p^t belongs to process p and transaction t. A sequence of events forms a transaction Tr that is issued sequentially by a process. Process p

cannot invoke a new transaction Tr_p^1 until the preceding transaction Tr_p^0 terminates. Also, a transaction Tr_p^t, which has a unique identifier (t, p) (helps in capture properties of each transaction invoked by the same process), cannot invoke the next operation $(\bigcirc E_p^t)$ until the previous operation E_p^t gets a response and cannot invoke an operation after it gets a commit or abort response. If a transaction aborts and requests again it is modelled as a new transaction. Here are the ITL formulae for representing the proposed abstract TM tm_{spec} as a group of processes, the transactions of an individual process and the events of an individual transaction:

$$\begin{aligned} tm_{spec} &\mathrel{\widehat{=}} \textstyle\bigwedge_{p=0}^{n} Process_p \\ Process_p &\mathrel{\widehat{=}} Tr_p^0;\ Tr_p^1;\ ...;\ Tr_p^t \\ Tr_p^t &\mathrel{\widehat{=}} E_{p,0}^t;\ E_{p,1}^t;\ ...;\ E_{p,m}^t \end{aligned} \tag{4.1}$$

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 $val \in \mathbb{N}$ of these objects are uninitialized and hence equal to $\perp$, so $obj \rightarrow val \cup \{\perp\}$. An event E is either an invocation by a transaction or a response as follows: Let $p, q \leq |Processes|$; $s, t \leq |Tr|$; $x, y \leq |Locations|$; $u, u', v \in \mathbb{N}$. Where $|L|$ represents length of the list L minus one.

- $R_p^t(x)$: a read operation by transaction t in process p. The response gives the current value u of object x and has the form $\widehat{R}_p^t(x, u)$.

- $W_p^t(x, u')$: a write value u' operation to object x by transaction t in process p. The response is ok. When the value written is of no importance and has no relevance, we write the above as $W_p^t(x)$ and regard both of the two forms as equivalent.

- $tryCom_p^t$: a commit request by transaction t in process p. If the attempt to commit succeeds, the response is com_p^t (or the notation $\oplus_p^t$) and it makes all the temporary updates visible by other transactions. If it fails, the response is $abort_p^t$ (or the notation $\otimes_p^t$) and it discards any update.

- $tryAbort_p^t$: an abort request by transaction t in process p, the response is $\otimes_p^t$ and it discards any update.

Note: 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 and 🄼 see Table 3.2):

$$\forall x, u \cdot \quad \boxed{\mathrm{f}} \big(fin\ \widehat{R}_p^t(x, u) \quad \equiv \Diamond(R_p^t(x) \wedge (\bigcirc\, \boxed{\mathrm{m}}\, no_{ev_p^t}) \wedge \widehat{R}_p^t(x, u)) \big)$$

$$\forall y, u' \cdot \quad \boxed{\mathrm{f}} \big(fin\ W_p^t(y, u') \quad \equiv \Diamond(W_p^t(y, u') \wedge (\bigcirc\, \boxed{\mathrm{m}}\, no_{ev_p^t}) \wedge ok_p^t) \big)$$

Main Components

As shown in Fig. 4.2, the proposed TM abstract model tm_{spec} has four main state variables which are:

- $Mem[obj]$: Persistent memory $(0 \leq obj < |Locations|)$; initially $\perp$.

- P_p : Process status $\in \{free$: *It does not have a transaction in progress*, $busy$: *It has an active or doomed transaction* $\}$; where $(0 \leq p < |Processes|)$; initially $free$.

- T_p^t : Transaction status $\in \{idle$: *It has not been issued*, $active$: *It is in progress*, $doomed$: *It is in progress but it has a conflict with another transaction and cannot commit*, $finished$: *It is committed or aborted*$\}$; where $(0 \leq t < |Tr|)$; initially $idle$.

- $E_{p,i}^t$: Event type $\in \{no_{ev}, r, w, ok, tryCom, tryAbort, \oplus, \otimes\}$; where $(0 \leq i < |E|)$; initially no_{ev}.

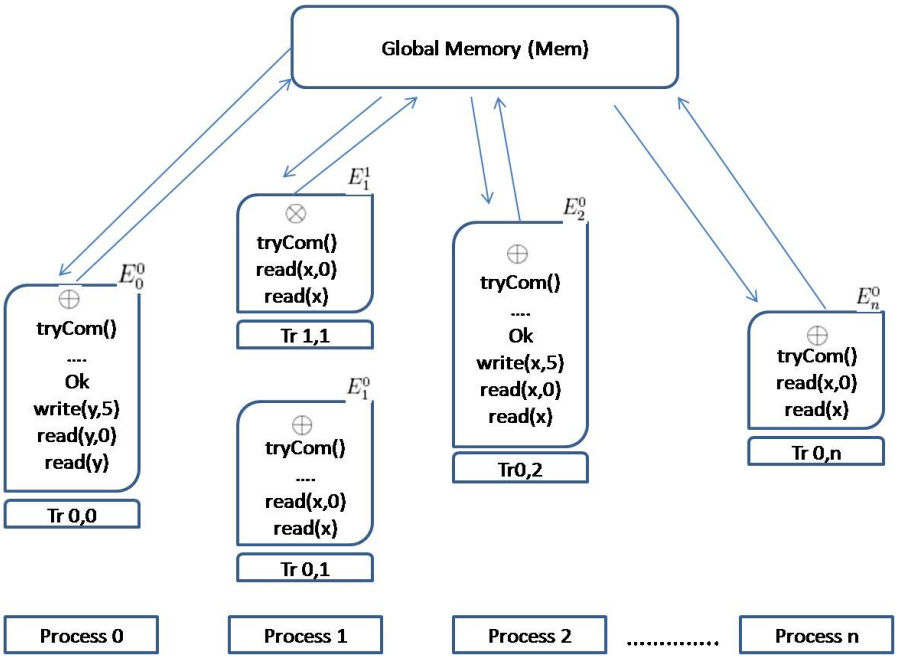

FIGURE 4.2: The proposed TM abstract model.

Transition Behaviour

The states transition of the tm_{spec} are, for better readability, partitioned into two tables: Table 4.2 lists all possible invocation states of the tm_{spec}, while Table 4.3 lists all possible responses for invocation of transactional operations states. Both tables describe the preconditions under which each transition can be taken and describe the effects of the transition on other variables. The formal description of the two tables and their relation are illustrated later in this section and in Appendix A. The formula $ConflictDetRes()$ in Table 4.2, which is defined later on in this section(see Equation 4.2), concerns the conflict detection and resolution mechanisms. While ε and ε_r in this formula refer to the mechanism type of conflict detection and resolution that is used by a TM system (their definitions and more details in Subsection 4.3.2, see Equations 4.9 to 4.15). The $skip$ formula, in the precondition column of both tables, describes that we use an interval of two states.

TABLE 4.2: The invocation actions of the tm_{spec}'s transactional operations

Case*		Preconditions	Actions	Event out
$R_p^t(x)$	s_0	P_p=$free \wedge T_p^t$=$idle$	$skip \wedge \bigcirc P_p$=$busy \wedge ConflictDetRes(p,t,\varepsilon,\varepsilon_r)$	$R_p^t(x)$
	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	$Otherwise$	$skip \wedge AbortTran(p,t)$	$\otimes$
$W_p^t(x,u')$	s_4	same as *read* case	same as *read* case	$W_p^t(x,u')$
$tryCom_p^t$	s_5	T_p^t=$active$	$skip \wedge ConflictDetRes(p,t,\varepsilon,\varepsilon_r)$	$tryCom_p^t$
	s_6	$\neg T_p^t$=$active$	$skip \wedge \bigcirc T_p^t$=$doomed$	
$tryAbort_p^t$	s_7	-	$skip$	$tryAbort_p^t$

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

In Table 4.2, the transitions $s_0 - s_4$ 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 as long as its status does not equal *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 and cannot commit if a conflict with other transaction is detected, 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 following definition for doomed transaction:

Definition 1 (Doomed Transaction). 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 conflict detection mechanism. Transaction T_p^t which has a $doomed$ status must eventually be aborted.

The concurrency control mechanism, i.e. conflict detection and resolution, is essential part in the TM systems implementation in order to detect and resolve conflicts between concurrent transactions accessing (and at least one modifies) the same object(s). There are many approaches to maintain conflict detection and resolution such as lazy (at commit time) and eager (at object access time). In our proposed abstract TM model, we firstly use lazy approach and then involve other conflict detection and resolution approaches. We now show the ITL formula of the conflict detection and resolution mechanism in tm_{spec} denoted *ConflictDetRes()*:

Let $p, q \leq |Processes|$ and $p \neq q$; $s, t \leq |Tr|$; $x, y \leq |Locations|$; $u \in \mathbb{N}$

$$ConflictDetRes(p, t, \varepsilon, \varepsilon_r) \mathrel{\widehat{=}} \boxed{a}\,(ConflictDet(p, t, \varepsilon) \supset ConflictRes(p, t, q, s, \varepsilon_r)) \quad (4.2)$$

The formula $ConflictDetRes(p, t, \varepsilon, \varepsilon_r)$ captures a conflict and resolves it by using two sub-formulas which are:

- $ConflictDet(p, t, \varepsilon)$ to check whether transaction t in process p conflicts with the concurrent transactions that have been issued by other processes (where ε specifies the type of detection). The following formula represents the lazy approach (ε_l) that detects a conflict when transaction t tries to commit ($tryCom_p^t$) before transaction s ($T_q^s = active$) and there exist, in the previous states, operations write an object in t and read the same object in s.

$$ConflictDet(p, t, \varepsilon_l) \mathrel{\widehat{=}} (\Diamond\, W_p^t(y, u') \wedge \Diamond\, R_q^s(y));\ (T_q^s = active \wedge tryCom_p^t \wedge empty) \quad (4.3)$$

- $ConflictRes(p, t, q, s, \varepsilon_r)$ to resolve a conflict between two concurrent transactions by aborting one of them (where ε_r specifies the resolution approach). The following formula changes the status of a transaction, which has conflict and still active, to *doomed* (can't commit).

$$ConflictRes(p, t, q, s, \varepsilon_{rl}) \mathrel{\widehat{=}} (\Diamond\, tryCom_p^t \wedge \neg \Diamond\, tryCom_q^s) \wedge \bigcirc\, T_q^s = doomed$$

The possible approaches of conflict detection such as *lazy*, *eager* and *mixed*, and approaches of conflict resolution such as *eager* and *lazy* arbitration are explained in detail in the next section (see Section 4.3). As shown in Table 4.2, we replicate instances $ConflictDetRes()$ to preserve the generality of this TM model. For example, if $\varepsilon = \varepsilon_l$ (Lazy), then $ConflictDetRes()$ will be activated at commit time ($tryCom$), while the others will be neglected.

TABLE 4.3: The response actions of tm_{spec}'s transactional operations

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

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

Transitions $\widehat{s}_0 - \widehat{s}_2$, in Table 4.3 list the three possible actions for responding to a transactional *read* operation $R_p^t(x)$. Transition $\widehat{s}_0$ 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. Transition $\widehat{s}_1$ 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 $InconsRead()$. This condition is used to prevent the doomed transaction in the proposed abstract TM accessing an inconsistent state and then causing illegal actions such as divided by zero (more details see Subsection 4.3.1).

Here is the ITL formula of *InconsRead()*:

$$\begin{aligned} InconsRead(p,t) \quad \widehat{=} \; & \Diamond(\widehat{R}_p^t(y,v) \wedge v = v' \,) \wedge \bigcirc \Box(\neg W_p^t(y)) \\ & \wedge fin\left(\widehat{R}_p^t(x,u) \wedge \neg(v = v')\right) \end{aligned} \tag{4.4}$$

The combination of $InconsRead()$ 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.

Transition $\widehat{s}_2$ returns a value of x from memory if there is not a previous $W_p^t(x,u')$ and there isn't an inconsistent read access.

We formalise these three possible actions in ITL formula denoted $ValidRead()$ as follows:

$$\begin{aligned} & ValidRead() \\ \widehat{=} \; & \boxed{a}\Big(\big(\big(\, fin(\widehat{R}_p^t(x,u)) \wedge f_0 \,\big) \supset u = u' \,\big) \\ & \wedge \big(\, (fin(\widehat{R}_p^t(x,u)) \wedge \big(\neg f_0 \wedge \Box(\neg W_p^t(x))\big) \wedge f_1) \supset (u = \bot \wedge fin{\otimes}_p^t) \,\big) \\ & \wedge \big(\big(\, fin(\widehat{R}_p^t(x,u)) \wedge \big(\neg f_0 \wedge \Box(\neg W_p^t(x))\big) \wedge \neg f_1) \supset u = Mem[x]\big)\big) \end{aligned} \tag{4.5}$$

where f_0 and f_1 are defined as follows:

$$\begin{aligned} f_0 \quad & \widehat{=} \quad W_p^t(x,u') \wedge \bigcirc \Box(\neg W_p^t(x)) \\ f_1 \quad & \widehat{=} \quad fin(\, T_p^t = doomed \wedge InconsRead(p,t)) \end{aligned}$$

The formula $ValidRead()$ 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 $(W_p^t(x) \wedge finR_p^t(x))$, and 3) no $W_p^t(x)$ in between. 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 the order satisfies $(R_p^t(y) \wedge finR_p^t(x))$, and 3) a conflict is detected because of that the value of y has been updated by a concurrent committed transaction. Finally, it equals to u'' if there is 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.

Transitions $\widehat{s}_4 - \widehat{s}_6$ respond to $tryCom$ and $tryAbort$ instructions. The actions are either committing a transaction (making all the temporary updates in the event list E permanent by transferring the updated object's value to the corresponding memory location) using $CommitTran()$ formula or aborting a transaction (undoing any update) using $AbortTran()$ formula, as follows: (note: more details for the following formula in Appendix A)

$$\begin{aligned} CommitTran(p,t) \mathrel{\widehat{=}} \quad & \bigcirc\, T_p^t = \textit{finished} \wedge \bigcirc\, E_p^t = \oplus \wedge \bigcirc\, P_p = \textit{free} \\ & \wedge\, UpdateMemory() \end{aligned} \tag{4.6}$$

$$AbortTran(p,t) \mathrel{\widehat{=}} \quad \bigcirc\, T_p^t = \textit{finished} \wedge \bigcirc\, E_p^t = \otimes \wedge \bigcirc\, P_p = \textit{free} \tag{4.7}$$

Definition 2 (Memory Update). In the tm_{spec} model, the value of a memory location such as $Mem[x]$ is permanently updated with a value such as u'' by $CommitTran()$ formula only iff β

is satisfied:

$$\beta \mathrel{\widehat{=}} \Big(\big(\big(W_q^s(x,u'') \wedge \bigcirc\Box(\neg W_q^s(x))\big) \wedge fin{\oplus}_q^s\big) \wedge \bigcirc\Box(\neg W_j^i(x)) \wedge fin{\oplus}_j^i\Big)$$

The formula β states that there is a finite interval that ends with a commit event in its last state and a previous write event state for an object such as x with a value such as u'' by the process's q transaction s. Also, there is not another finite interval that has a sequence of both a write event for the same object x and a commit event at the end.

To guarantee that each invocation event is followed by a response and each active transaction eventually finishes, we categorise the events described in Tables 4.2 and 4.3 into two parts as follows: (for their definitions, see Fig. A.1 in Appendix A)

- $TranInvOp()$ for *read* and *write* invocation of transactional operations, and the formula $TranResOp()$ for its response.
- $TranInvEnd()$ for *tryComit* and *tryAbort* invocation of ending a transaction, and the formula $TranResEnd()$ for its response.

According to these categories, the sequence of invocation and response events that form transaction Tr_p^t and described in 4.1 can be modelled in the following ITL formula:

$$\begin{aligned} Tr_p^t \mathrel{\widehat{=}} \quad & ((TranInvOp(p,t,op);\ TranResOp(p,t))^*; \\ & TranInvEnd(p,t,op);\ TranResEnd(p,t)) \end{aligned}$$

In addition, the complete specification of the abstract TM model tm_{spec} that described in 4.1 can be modelled with the initial values in the following ITL formula:

$$\begin{aligned} tm_{spec} \mathrel{\widehat{=}} & \bigwedge_{p=0}^{n} Process_p \\ Process_p \mathrel{\widehat{=}} & P_p = free \wedge T_p^t = idle \\ & \wedge \big((TranInvOp(p,t,op);\ TranResOp(p,t))^*; \\ & \quad TranInvEnd(p,t,op);\ TranResEnd(p,t) \big)^* \end{aligned} \tag{4.8}$$

To ensure the validity of our proposed TM abstract model tm_{spec}, we build an executable specification for tm_{spec} (see Fig. A.1 in Appendix A) by refining the high-level TM abstract specification written in ITL into a set of Tempura modules using the refinement rules in [18, 54]. Then we simulate and analyse this model using AnaTempura (see Appendix A).

4.3 Formalisation of TM Safety Properties

Many TM correctness conditions have been proposed in the literature with varying degrees of precision and rigour. However, the basic correctness property for concurrent transactions is *serialisability* [7, 64]. A *transactional history* (TH) 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 *sequential transactional history* (STH) 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. We will consider all subintervals in finite time.

4.3.1 Read Consistency

A TM system shows only a single memory image to the concurrent transactions. Each transaction is allowed to read from and write into each memory location. Also, more than one transaction may read the same memory location at the same time. However, transactions executing at the same time need to access a consistent state, i.e., a state produced by a sequence of previously committed transactions [40]. One of the inconsistent read states arises when the value returned by each read operation is not the value written by the last committed write operation in that location. The read consistency property can deal with such cases and go on to ensure that transactions run in such a way that they appear to be executed one at a time, or serially, rather than concurrently. In this section, we show three conditions which are needed to preserve the read consistency.

Local Read Consistency

The order of read and write operations within a transaction should be preserved in the same way that appear in the program. This can be violated if the read operation returns the last value that is written by the last committed write of a different transaction regardless of the last write

by the same transaction to the same location. In this case the later read will appear as executed before the early write operation. The local read consistency property guarantees that each read operation returns the last write by the same transaction to the same location.

Example:

where $p \leq |Processes|$; $t \leq |Tr|$; $x \leq |Locations|$; $u, u' \in \mathbb{N}$

$$Processes_p \mathrel{\widehat{=}} W_p^t(x,u);\ ok;\ R_p^t(x);\ \widehat{R}_p^t(x,u)$$

This example shows that the response comes from the process's local write and not from global memory, because the same location has been written by the same transaction. So, it satisfies the local consistency commitment (see Fig. 4.3).

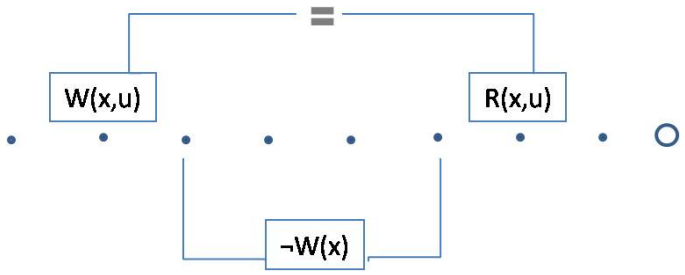

FIGURE 4.3: Local consistency.

Definition 3 (Local 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.

We now show how to express local consistency as an ITL formula denoted *Local_Cons*:

$$Local\_Cons \quad \widehat{=} \neg \Diamond\!\!\!\!-(\varphi \wedge u \neq u')$$

φ is defined as follows: $\forall x$

$$\varphi \quad \widehat{=} \quad \Big((W_p^t(x,u') \wedge skip);\ \Box(\neg W_p^t(x));\ (\widehat{R}_p^t(x,u) \wedge empty)\Big)$$

The local formula states that if a finite subinterval has a write event state for an object such as x with a value such as u', ends with a read response event for the same object x with a value such as u and no other write event for the same object occurring in between, then u' should equal u.

Remark 1. My supervisor Dr. Ben Moszkowski has suggested the following alternative ITL formula for representing the local read consistency property:

$$Local\_Cons \quad \widehat{=} \forall x \cdot \boxed{\mathrm{f}} \Big((fin\ \widehat{R}_p^t(x,u)\ ) \supset \Diamond \big((W_p^t(x,u) \wedge skip);\ \Box(\neg W_p^t(x))\big)\ \Big)$$

This kind of formula is further discussed in his interesting paper [55]. This formula states that if a finite prefix subinterval ends with a read value, then the same transaction previously wrote this value. No other transaction occurred in between. In actual fact, this formal representation is simpler than the first one. It uses fewer terms and precisely describes the local read property. Because of the lack of time, it will be considered in the future work.

Doomed Read Consistency

Guerraoui and Kapalka [41] 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, called *doomed consistency*, as one of the safety conditions that can lead finally to strict serialisability

(where defined later in Subsection 4.3.3) with the property that even doomed transactions do not observe an inconsistent state (see Fig. 4.4).

Here is an example will initially y=4, x=2.

$p \neq q \leq |Processes|\ ;\ s,t \leq |Tr|\ ;\ x,y \leq |Locations|\ ;\ u \in \mathbb{N}$

$Processes_p \mathrel{\widehat{=}} R_p^t(y);\ \widehat{R}_p^t(y,4);\ R_p^t(x);\ \widehat{R}(x,4);\ W_p^t(z,1/(y-x))$

$Processes_q \mathrel{\widehat{=}} W_q^s(y,6);\ ok;\ W_q^s(x,4), ok, tryCom_q^s, \oplus_q^s$

A case with divided-by-zero clearly occurs in this example when the value of x is changed by transaction s, where x-y=0 and z=1/0.

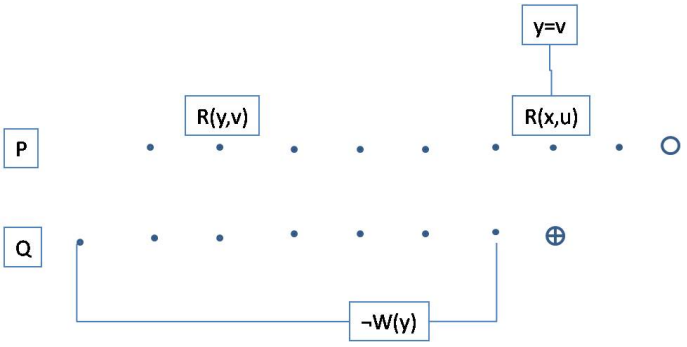

FIGURE 4.4: Doomed consistency.

Definition 4 (Doomed Consistency). Each transaction in tm_{spec} satisfies 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.

We now show how to express doomed consistency as an ITL formula denoted by *Doomed_Cons*:

$$Doomed\_Cons \quad \widehat{=} \neg \diamondsuit(\psi \wedge \neg(u = \perp \wedge \otimes_p^t))$$

where ψ is defined as follows: $\forall\, x, y$ and $p \neq q$

$$\psi \quad \widehat{=} \ \Box(\neg W_p^t(x))$$
$$\wedge \Big(\big(\big(R_p^t(y) \wedge empty;\ \Diamond W_q^s(y)\big) \vee \big(W_q^s(y) \wedge empty;\ \Diamond R_p^t(y)\big) \big)$$
$$\wedge fin(\oplus_q^s \wedge T_p^t = active) \Big);\ fin(\widehat{R}_p^t(x, u))$$

Global Read Consistency

Definition 5 (Global Consistency). A transaction in tm_{spec} satisfies the *global consistency* condition iff each $R(x, u)$ in the successful transaction (no conflict or not doomed) returns the most recent $W(x, u'')$ in any committed transaction.

We now show how to express global consistency as an ITL formula denoted by *Global_Cons*:

$$Global\_Cons \quad \widehat{=} \neg \diamondsuit(\alpha \wedge u \neq u'')$$

where α is defined as follows: $\forall\, x, y$ and $p \neq q \neq j$

$$\alpha \quad \widehat{=} \ \Box(\neg W_p^t(x)) \wedge \Big(\big(\big(W_q^s(x, u'') \wedge skip;\ \Box(\neg W_q^s(x))\big)$$
$$\wedge fin\oplus_q^s) \wedge \bigcirc(\neg \diamondsuit(\Diamond W_j^i(x) \wedge fin\oplus_j^i))\big)$$
$$;\ fin\big(\widehat{R}_p^t(x, u) \wedge \neg(\, T_p^t = doomed \wedge InconsRead(p, t))\big)$$

The global formula states that if a finite subinterval has the following sequences: Firstly, a finite subinterval that has a commit event in its last state $(fin \oplus)$ and a previous write event state for an object such as x with a value such as u'' by the process's q transaction s. Secondly, there is no other finite subinterval that has a sequence of write event for the same object x and commit event at the end. Finally, if there is a read response event for the same object x with a value such

as u and there are no a local write for the same object and a conflict detection , then u should equal u''.

4.3.2 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 [39] presents practical policies for detecting conflicts to describe the STH's characteristic of different classes of TM systems. Although these conflict policies are meant to serve as strong conditions [41], he does not include the case of $write$ for the same object by two concurrent transactions (as shown in this section). Also, Scott introduces arbitration functions to ensure progress by specifying which of the two conflicting transactions will fail. We augment Scott's policies for detecting and solving conflicts by adding a case for exclusive read which is used in shared memory systems.

Conflict Detection

There are two main methods to handle conflict detection: lazy (sometimes referred to as late or optimistic) and eager (sometimes referred to as early or pessimistic). Lazy conflict detection is based on the concept that the TM system delays the detection until a transaction requests to commit, while the concept of eager is that the TM system detects conflicts as soon as they occur.

Here we formalise the classes of conflict detection 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 write W an object in s and read R the same object in t such that s commits before the end of t (see Fig. 4.5).

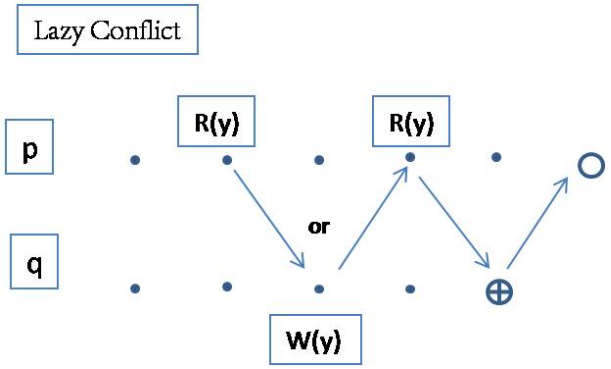

FIGURE 4.5: Lazy conflict.

$$\varepsilon_l \quad \widehat{=} fin(tryCom_q^s) \wedge (\Diamond W_q^s(y,u') \wedge \Diamond R_p^t(y) \wedge fin(T_p^t = active)) \tag{4.9}$$

- 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 read R an object in s and write W the same object in t such that W precedes R or vice versa, but neither transaction has ended (see Fig. 4.6).

$$\begin{aligned} \varepsilon_e \quad \widehat{=} \varepsilon_l \vee \big(& \big(fin(R_q^s(y)) \wedge (\Diamond W_p^t(y) \wedge fin(T_p^t = active))\big) \\ & \vee \big(fin(W_p^t(y)) \wedge (\Diamond R_q^s(y) \wedge fin(T_q^s = active))\big) \big) \end{aligned} \tag{4.10}$$

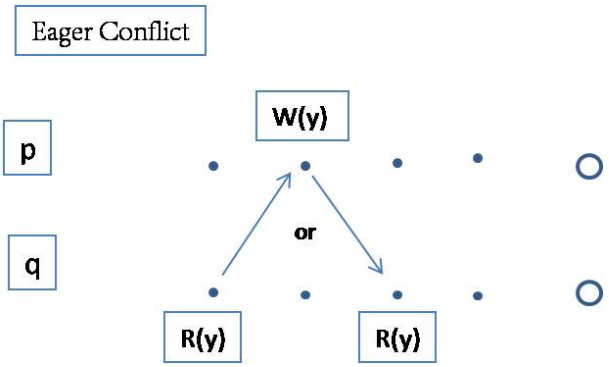

FIGURE 4.6: Eager conflict.

- Strong Eager Conflict (ε_{e^s}): Some systems use exclusive read operation to prepare for writing. This formula can detect a conflict for such instructions. The exclusive read operation is treated like a write in ε_e as follows: transactions t belongs to process p and transaction s belongs to process q conflict if t and s have a lazy conflict or if there exist operations read R an object in s and write W or R the same object in t such that W or R precedes R or vice versa, but neither transaction has ended.

$$\begin{aligned}\varepsilon_{e^s} \quad \widehat{=} \varepsilon_l \vee \big(\big(fin(R_q^s(y)) \wedge (\Diamond(W_p^t(y) \vee R_p^t(y)) \wedge fin(T_p^t = active)) \big) \\ \vee \big(fin(W_p^t(y) \vee R_p^t(y))) \wedge (\Diamond R_q^s(y) \wedge fin(T_q^s = active))) \big)\end{aligned} \tag{4.11}$$

- 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 write W an object in t, read R and write W the same object in s such that R precedes the two instances of W, but neither transaction has ended (see Fig. 4.7).

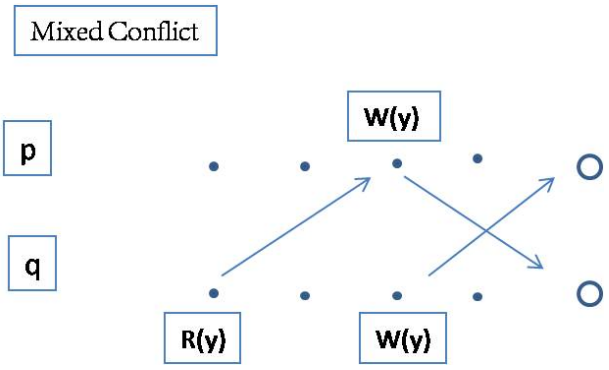

FIGURE 4.7: Mixed conflict.

$$\begin{aligned}\varepsilon_m \quad \widehat{=} \varepsilon_l \vee \big(\big(fin(W_p^t(y)) \wedge (\Diamond(R_q^s(y) \wedge \bigcirc \Diamond W_q^s(y)) \wedge fin(T_q^s = active)) \big) \\ \vee \big(fin(W_q^s(y)) \wedge (\Diamond(R_q^s(y) \wedge \bigcirc \Diamond W_p^t(y)) \wedge fin(T_p^t = active)) \big) \big)\end{aligned} \tag{4.12}$$

Conflict Resolution

Transactional memory systems have a contention management policy (arbitration) to resolve a conflict between two transactions by aborting one of them. Scott [39] suggests three arbitration functions:

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

$$\varepsilon_{re} \quad \widehat{=} fin \otimes_p^t \wedge (\Diamond T_p^t = active;\ \Diamond T_q^s = idle) \tag{4.13}$$

- Eagerly own arbitration (ε_{ro}): Whoever owns the conflict object first wins.

$$\varepsilon_{ro} \quad \widehat{=} fin(R_p^t(y) \vee W_p^t(y,u));\ fin \otimes_p^t \wedge T_q^s = active \tag{4.14}$$

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

$$\varepsilon_{rl} \quad \widehat{=} \, fin \otimes_p^t \wedge (\neg \Diamond tryCom_p^t \wedge \Diamond tryCom_q^s) \tag{4.15}$$

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

$$\begin{aligned} ConflictFree(\varepsilon, \varepsilon_r) &\quad \widehat{=} \neg \, \diamondsuit (\varepsilon \wedge \neg \varepsilon_r) \\ \varepsilon &\quad \widehat{=} \, \varepsilon_l \vee \varepsilon_e \vee \varepsilon_{e^s} \vee \varepsilon_m \\ \varepsilon_r &\quad \widehat{=} \, \varepsilon_{re} \vee \varepsilon_{rl} \vee \varepsilon_{ro} \end{aligned}$$

To help the reader follow the formulae of all previous safety properties, we collect them in one place, as shown in Table 4.4.

4.3.3 Strict Serialisability

Papadimitriou [65] augments the strength of serialisability by adding the requirement of real time ordering of the committed transactions. A TM system satisfies this property if:

- Each read in every successful transaction satisfies the read consistency conditions.
- The committed transactions can be ordered serially according to the order of their committed operations.
- Every read and write operation in serially ordered committed transactions appear serially according to their transactions' ordering.

TABLE 4.4: Formal TM safety properties

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

We formalise this property as follows: Let $\sigma : s_0, s_1, s_2, ...$ be a finite or infinite sequence of states. Each state is a mapping from variable to value and because an event E is represented by a variable you can represent the concurrent observable events by a set of variables with Boolean indications whether they occurred in the state or not. Each sequence of events formed a transaction Tr that is issued sequentially. The sequence σ is called transactions history TH.

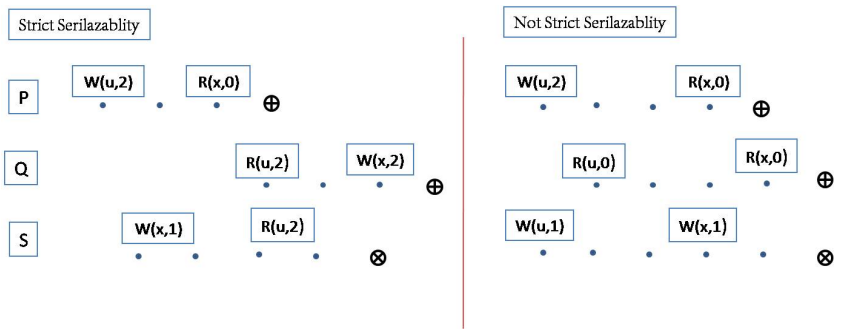

FIGURE 4.8: Strict and Non-Strict Serialisability.

Moreover, 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 proposed model tm_{spec}, 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 6 (Strict Serialisability). The TH can be strictly serialised, if we can obtain σ' from σ with respect to $Ser(TH)$ as follows:

$$
\begin{aligned}
Ser(TH) \quad &\widehat{=} (Tr_p^t;\ Tr_q^s) \\
&\equiv Tr_p^t \wedge Tr_q^s \\
&\quad \wedge \{p \neq q\} \\
&\quad \wedge \{\textit{The order of transactions over } \sigma' \textit{ is the} \\
&\qquad \textit{order of the committing events for the} \\
&\qquad \textit{same transactions } (fin\oplus_p^t;\ fin\oplus_q^s) \textit{ over } \sigma\} \\
&\quad \wedge \{\textit{all } R_p^t \textit{ and } R_q^s \textit{ over } \sigma \textit{ respects Read-Consistency property}\} \\
&\quad \wedge \{\textit{all } \oplus_p^t \textit{ and } \oplus_q^s \textit{ over } \sigma \textit{ respects Conflict-Free property}\}
\end{aligned}
$$

4.4 Verification of Abstract TM Model

There have been several recent approaches proposed for verifying the correctness of TM systems, which use a model checker based on algorithms [43, 64]. However, this method has limitations because of state explosion for large scale concurrent systems, namely the exponential growth of the global state space in the number of components [66]. Instead, we use a compositional technique based on the mathematical verification method. This method could perhaps allow us in the future to do a mechanical verification using special-purpose theorem provers such as PVS or KIV [48, 66, 67].

We propose a simplification of the 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 studied by the TM community is strict serialisability with respect to doomed consistency. However, the verification of this property depends on two other properties, which are read-consistency (*Local_Cons*, *Doomed_Cons* and *Global_Cons*) and conflict-free properties, see Subsection 4.3.1.

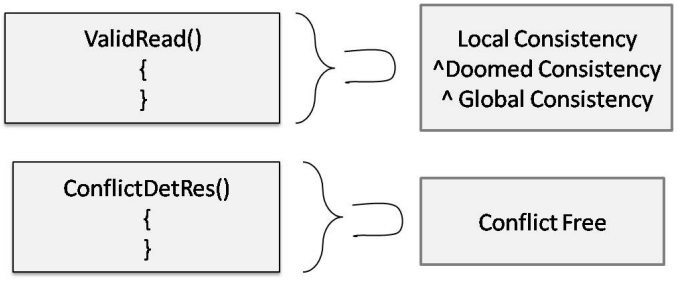

FIGURE 4.9: Safety proof.

In the tm_{spec} case, the two main components that can affect the read consistency, detection conflict and correct resolution of conflict are $ValidRead()$ and $ConflictDetRes()$. Therefore we must firstly verify that each of these components satisfies the corresponding property as follows:

Lemma 4.1. $ValidRead() \supset Local\_Cons$

Lemma 4.2. $ValidRead() \supset Doomed\_Cons$

Lemma 4.3. $ValidRead() \supset Global\_Cons$

Lemma 4.4. $ConflictDetRes(\varepsilon, \varepsilon_r) \supset ConflictFree(\varepsilon, \varepsilon_r)$

To prove these lemmas, we simplify the lhs of each part using an assumption to reduce the big formula and then we mathematically prove that new formula satisfies the rhs by using propositional reasoning, the ITL (semantic) inference rules and the definition of ITL operators.

Moreover, some definitions are used in this verification approach to cover the relationship between the model's components. For example, the definitions of doomed transaction (Definition 1) and memory update (Definition 2) show the relationship between the $CommitTran()$ and the $ValidRead()$.

Before starting the proof of these lemmas, we will remind the readers about the definition 4.5 of formula $ValidRead()$ in Section 4.2:

$$
\begin{array}{l}
ValidRead() \\
\widehat{=} \ \boxed{a} \Big(\big(\big(fin(\widehat{R}_p^t(x,u)) \wedge f_0 \big) \supset u = u' \big) \\
\quad \wedge \Big(\big(fin(\widehat{R}_p^t(x,u)) \wedge \big(\neg f_0 \wedge \Box(\neg W_p^t(x)) \big) \wedge f_1 \big) \supset (u = \bot \wedge fin{\otimes}_p^t) \Big) \\
\quad \wedge \Big(\big(fin(\widehat{R}_p^t(x,u)) \wedge \big(\neg f_0 \wedge \Box(\neg W_p^t(x)) \big) \wedge \neg f_1 \big) \supset u = Mem[x] \big) \Big)
\end{array}
\tag{4.16}
$$

where f_0 and f_1 are defined as follows:

$$
\begin{array}{lcl}
f_0 & \widehat{=} & W_p^t(x,u') \wedge \bigcirc \Box(\neg W_p^t(x)) \\
f_1 & \widehat{=} & fin(T_p^t = doomed \wedge InconsRead(p,t))
\end{array}
$$

Proof: [Lemma 4.1]

We start by assuming that process p's transaction t has a local write $W_p^t(x)$. According to that assumption the $ValidRead()$ formula can be simplified as follows:

$ValidRead()$

$\equiv$

$\{$ *assuming* $W_p^t(x)$, *then* $\Box(\neg W_p^t(x)) = False$ $\}$

1. $\boxed{a} \Big(\big(\big(fin(\widehat{R}_p^t(x,u)) \wedge f_0 \big) \supset u = u' \big)$
   $\wedge \Big(\big(fin(\widehat{R}_p^t(x,u)) \wedge (\neg f_0 \wedge False) \wedge f_1 \big) \supset (u = \bot \wedge fin{\otimes}_p^t) \Big)$
   $\wedge \Big(\big(fin(\widehat{R}_p^t(x,u)) \wedge (\neg f_0 \wedge False) \wedge \neg f_1 \big) \supset u = Mem[x] \big) \Big)$

{ *1, propositional reasoning* }

2. $\boxed{a}\Big(\big(\big(\ fin(\widehat{R}_p^t(x,u)) \wedge f_0\ \big) \supset u = u'\ \big) \wedge \text{True} \wedge \text{True}\Big)$

{ *2, propositional reasoning* }

3. $\boxed{a}\Big(\neg\big(\big(\ fin(\widehat{R}_p^t(x,u)) \wedge f_0\ \big) \wedge \neg(u = u')\ \big)\Big)$

{*3, definition of* $\boxed{a}$ }

4. $\neg \diamondsuit \neg\Big(\neg\big(\big(\ fin(\widehat{R}_p^t(x,u)) \wedge f_0\ \big) \wedge \neg(u = u')\ \big)\Big)$

{*4, propositional reasoning* }

5. $\neg \diamondsuit \big(\big(\ fin(\widehat{R}_p^t(x,u)) \wedge f_0\ \big) \wedge u \neq u'\ \big)$

$f_0 \equiv$ {*definition of* f_0 (4.16)}

5.1. $W_p^t(x,u') \wedge \bigcirc \Box(\neg W_p^t(x))$

{*5.1, definition of ITL operators* $\bigcirc$ *and ITL inference rules*}

5.2. $W_p^t(x,u') \wedge (skip;\ \Box(\neg W_p^t(x)))$

{*5.2, ITL inference rules*}

5.3. $(W_p^t(x,u') \wedge skip);\ \Box(\neg W_p^t(x))$

{*5.1,5.3, substituting* f_0*(5.3) in 5* }

6. $\neg \diamondsuit \big(\big(\ fin(\widehat{R}_p^t(x,u)) \wedge (W_p^t(x,u') \wedge skip);\ \Box(\neg W_p^t(x))\ \big) \wedge u \neq u'\ \big)$

{*6, definition of ITL operators* fin}

7. $\neg \diamondsuit\big(\ (\ W_p^t(x,u') \wedge skip;\ \Box(\neg W_p^t(x));\ \widehat{R}_p^t(x,u) \wedge empty) \wedge u \neq u'\ \big)$

{*1-7, where 7 is equivalent to Local_Cons (see Table 4.4, page 69)* }

$ValidRead() \supset$ Local_Cons

Proof: [Lemma 4.2]

We start by assuming that process p's transaction t doesn't have a local write $\Box(\neg W_p^t(x))$ and transaction t is doomed $fin(\,T_p^t = doomed)$. According to these assumptions the $ValidRead()$ formula can be simplified as follows:

$ValidRead()$

$\equiv$

1. $\boxed{a}\Big(\big(\big(\,fin(\widehat{R}_p^t(x,u)) \wedge False\,\big) \supset u = u'\,\big)$

   $\wedge\Big(\big((fin(\widehat{R}_p^t(x,u)) \wedge \big(\neg f_0 \wedge \Box(\neg W_p^t(x))\big) \wedge f_1\big) \supset (u = \perp \wedge fin\otimes_p^t)\,\big)$

   $\wedge\Big(\big(\,fin(\widehat{R}_p^t(x,u)) \wedge \big(\neg f_0 \wedge \Box(\neg W_p^t(x))\big) \wedge False\big) \supset u = Mem[x]\big)\Big)$

   { *1, propositional reasoning* }

2. $\boxed{a}\Big(True \wedge \big(\big(\,fin(\widehat{R}_p^t(x,u)) \wedge \big(\neg f_0 \wedge \Box(\neg W_p^t(x))\big) \wedge f_1\big)$

   $\supset (u = \perp \wedge fin\otimes_p^t)\,\big) \wedge \text{True}\Big)$

   {*2, let* $\epsilon \equiv \neg f_0 \wedge \Box(\neg W_p^t(x))$ }

3. $\boxed{a}\Big(\big(\big(\,fin(\widehat{R}_p^t(x,u)) \wedge \epsilon \wedge f_1\big) \supset (u = \perp \wedge fin\otimes_p^t)\,\big)\Big)$

$\epsilon \equiv$ { *simplifying* ϵ*, where W(x) and W(x,u') are regarded as equivalent* }

3.1. $\neg f_0 \wedge \Box(\neg W_p^t(x))$

{*3.1, definition of* f_0 (4.16)}

3.2. $\big(\neg W_p^t(x,u') \vee \neg(skip;\ \Box(\neg W_p^t(x)))\big) \wedge \Box(\neg W_p^t(x))$

{*3.2, propositional reasoning*}

3.3. $\big(\neg W_p^t(x,u') \wedge \Box(\neg W_p^t(x))\big) \vee \big(\neg(skip;\ \Box(\neg W_p^t(x))) \wedge \Box(\neg W_p^t(x))\big)$

{*3.3, propositional ITL*}

3.4. $\Box(\neg W_p^t(x)) \vee (empty \wedge \neg W_p^t(x))$

{*3.4, propositional ITL*}

3.5. $\Box(\neg W_p^t(x))$

{*3.1,3.5, replacement of ϵ by 3.5 in 3*}

4. $\boxed{a}\Big(\big(\big(\, fin(\widehat{R}_p^t(x,u)) \wedge \Box(\neg W_p^t(x)) \wedge f_1\big) \supset (u = \perp \wedge fin{\otimes}_p^t)\,\big)\Big)$

{*4, propositional ITL*}

5. $\neg \diamondsuit \neg \Big(\neg\big(\big(\, fin(\widehat{R}_p^t(x,u)) \wedge \Box(\neg W_p^t(x)) \wedge f_1\big) \wedge \neg(u = \perp \wedge fin{\otimes}_p^t)\,\big)\Big)$

{*5, propositional reasoning*}

6. $\neg \diamondsuit \Big(\big(\, fin(\widehat{R}_p^t(x,u)) \wedge \Box(\neg W_p^t(x)) \wedge f_1\big) \wedge \neg(u = \perp \wedge fin{\otimes}_p^t)\,\Big)$

$f_1 \equiv$ {*definition of f_1 (4.16)*}

6.1. $fin(\, T_p^t = doomed \wedge InconsRead())$

{*Definition 1 of doomed transaction, we assume*

a weakest conflict detection type (lazy) is used (4.3), (4.4)}

6.2. $fin \oplus_q^s \wedge (\Diamond\, W_q^s(y) \wedge \Diamond\, R_p^t(y) \wedge fin(T_p^t = active))$

{*6.2, propositional ITL*}

6.3. $\Big(\big(\big(R_p^t(y) \wedge empty;\ \Diamond\, W_q^s(y)\big) \vee \big(W_q^s(y) \wedge empty;\ \Diamond\, R_p^t(y)\big)\big)$
$\wedge\, fin(\oplus_q^s \wedge T_p^t = active)\Big)$

6.4. $\equiv f_1'$

{*6.1,6.4, substitution of* f_1 *in 6 by 6.4 and definition of fin* }

7. $\neg \diamondsuit \big(\big(\Box(\neg W_p^t(x)) \wedge f_1';\ fin(\widehat{R}_p^t(x,u)) \big) \wedge \neg(u = \bot \wedge fin\otimes_p^t) \big)$

{*1-7, 7 is equivalent to Doomed_Cons (see Table 4.4, page 69)* }

$ValidRead()$ $\supset$ Doomed_Cons

Proof: [Lemma 4.3]

We start by assuming that process p's transaction t doesn't have a local write $\Box(\neg W_p^t(x))$ and transaction t isn't doomed $fin(T_p^t \neq doomed)$. According to these assumptions the $ValidRead()$ formula can be simplified as follows:

$ValidRead()$

$\equiv$

1. $\boxdot \big(\big(\big(fin(\widehat{R}_p^t(x,u)) \wedge False \big) \supset u = u' \big)$
$\wedge \big(\big(fin(\widehat{R}_p^t(x,u)) \wedge \big(\neg f_0 \wedge \Box(\neg W_p^t(x))\big) \wedge False \big) \supset (u = \bot \wedge fin\otimes_p^t) \big)$
$\wedge \big(\big(fin(\widehat{R}_p^t(x,u)) \wedge \big(\neg f_0 \wedge \Box(\neg W_p^t(x))\big) \wedge \neg f_1 \big) \supset u = Mem[x] \big) \big)$

{ *1, propositional reasoning* }

2. $\boxdot \big(True \wedge True \big(\big(fin(\widehat{R}_p^t(x,u)) \wedge \big(\neg f_0 \wedge \Box(\neg W_p^t(x))\big) \wedge \neg f_1 \big)$
$\supset u = Mem[x] \big) \big)$

{*2, simplification of* $\neg f_0 \wedge (\Box(\neg W_p^t(x))$*, see 3.5 in proofs of* (4.2) }

3. $\boxdot \big(\big(fin(\widehat{R}_p^t(x,u)) \wedge (\Box(\neg W_p^t(x)) \wedge \neg f_2 \big)$
$\supset u = Mem[x] \big)$

{*Assuming that there is a previous write by a committed transaction*

in location x *with value* u''*. The* $u = Mem[x]$ *term is rewritten as follows:*}

3.1. $Mem[x] = u'' \supset u = u''$

{ *3.1, Definition 2 of updating a memory location*}

3.2. $\Big(\big(\big(W_q^s(x, u'') \wedge \bigcirc \Box(\neg W_q^s(x)) \big) \wedge fin{\oplus}_q^s \big)$
$\wedge \bigcirc(\neg \diamondsuit_{\Phi}(\Diamond W_j^i(x) \wedge fin{\oplus}_j^i)) \Big) \supset u = u''$

{ *3.2, propositional ITL* }

3.3. $\Big(\big(\big(W_q^s(x, u'') \wedge skip;\ \Box(\neg W_q^s(x)) \big) \wedge fin{\oplus}_q^s \big)$
$\wedge \bigcirc(\neg \diamondsuit_{\Phi}(\Diamond W_j^i(x) \wedge fin{\oplus}_j^i)) \Big) \supset u = u''$

{*3.1,3.3, simplification* }

3.4. $\equiv \gamma \supset u = u''$

{ *substitution* $u = Mem[x]$ *in 3 by 3.4* }

4. $\boxed{a} \big(\big(fin(\widehat{R}_p^t(x, u)) \wedge (\Box(\neg W_p^t(x)) \wedge \neg f_1) \supset \gamma \supset u = u'' \big)$

{ *4, propositional reasoning* }

5. $\boxed{a} \big(\big(fin(\widehat{R}_p^t(x, u)) \wedge (\Box(\neg W_p^t(x)) \wedge \gamma \wedge \neg f_1) \supset u = u'' \big)$

{*5,propositional reasoning and definition of ITL operator* $\boxed{a}$ }

6. $\neg \diamondsuit_{\Phi} \big(\big(fin(\widehat{R}_p^t(x, u)) \wedge (\Box(\neg W_p^t(x)) \wedge \gamma \wedge \neg f_1) \wedge u \neq u'' \big)$

{*6, definition of ITL operators fin and definition of* f_1 (4.16)}

7. $\neg \diamondsuit \big(\big((\Box(\neg W_p^t(x)) \wedge \gamma;\ fin(\widehat{R}_p^t(x,u) \wedge \neg(T_p^t = doomed \wedge InconsRead(p,t))) \big) \wedge u \neq u'' \big)$

{*1-7, 7 is equivalent to Global_Cons (see Table 4.4, page 69)* }

$ValidRead() \supset$ Global_Cons

Proof: [Lemma 4.4]

Let us simplify the formula $ConflictDetRes()$ (see Section 4.2) of (4.4):

$ConflictDetRes(\varepsilon, \varepsilon_r)$

$\equiv \quad \boxed{a}\,(ConflictDet(p,t,\varepsilon) \supset ConflictRes(p,t,q,s,\varepsilon_r))$

{*propositional reasoning and the definition of ITL operator* $\boxed{a}$ }

$\neg \diamondsuit \neg\,(\neg(ConflictDet(p,t,\varepsilon) \wedge \neg ConflictRes(p,t,q,s,\varepsilon_r)))$

Then,

$\neg \diamondsuit\,(ConflictDet(p,t,\varepsilon) \wedge \neg ConflictRes(p,t,q,s,\varepsilon_r))$

{ *The* ε *and* ε_r *are equivalent to* ConflictFree() *(see Table 4.4, page 69)* }

$ConflictDetRes(\varepsilon, \varepsilon_r) \supset ConflictFree(p,t,\varepsilon,\varepsilon_r)$

Strict Serialisability

As we mentioned previously in Section 4.3, strict serialisability with respect to doomed consistency is considered the primary standard safety condition. According to its definition (see Subsection 4.3.3), it depends on the correctness of the two main components that are responsible for violating the read consistency and conflict free safety conditions which are $ConflictDetRes()$ and $ValidRead()$. Since we have already established the correctness of these components (see Section 4.4), strict serialisability can be simply proved.

Overlap between two transactions such as Tr_p^t and Tr_q^s means that the two transactions execute concurrently at the same states. Here is the formal definition of overlap: $p \neq q$

$$\begin{aligned} overlap(Tr_p^t, Tr_q^s) &\;\widehat{=}\; Tr_p^t \wedge Tr_q^s \\ &\;\widehat{=}\; (Tr_p^t;\ Tr_q^s) \vee (Tr_q^s;\ Tr_p^t) \end{aligned}$$

Theorem 4.5. *The Transaction History TH σ can be serialised with respect to $Ser(TH)$ if there is no overlap and conflict over σ.*

Proof: [Theorem 4.5] (by contradiction)

Let p and q be processes, t and s be transactions, u and $u' \in \mathbb{N}$ and x be a memory location, such that t and s have an 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 assume that σ can be serialised with respect to $Ser(TH)$ and obtain a contradiction. According to Definition 6, σ' can be obtained from σ and satisfies that each commit respects the $ConflictDetRes()$ formula (that satisfies $ConflictFree()$) which is violated 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 cannot be serialised.

4.5 Summary

In this chapter a general and flexible TM formal abstract model is presented. In addition, standard safety properties studied by the TM community are specified such as conflict free, read consistency and strict serialisability . Some additional conflict detection and resolution policies are specified to make the proposed TM model more general. For example, the strong-eager-conflict policy is specified in order to detect a conflict in TM systems that use a read exclusive instruction. Also, the eager-own-arbitration is specified in order to solve a conflict in the TM systems that use a cache coherency protocol. ITL and its executable subset AnaTempura are used to specify and validate the specification of the abstract TM. The major benefit of the validation process is ensuring the specification correctness of the proposed abstract TM model. Moreover, a mathematical verification method to prove that the proposed abstract TM satisfies the standard TM conditions is used.

Chapter 5

Validation and Refinement of a TM System

5.1 Introduction

Although a variety of methodologies for verifying the correctness of transactional memory systems have been developed, most of them only capture the main algorithmic aspects of those systems. In addition, the specification methods for TM systems are more abstract than real designs and may overlook many details as a result.

As shown in the framework's Fig. 5.1, this chapter focuses on the steps inside the circle. We have the objective to eventually transform a provably correct TM system into a hardware design. Therefore in this chapter we show a concrete specification, close to reality and more than just an abstract concept, for the chosen TM system. Then, for the sake of reassurance and understandability, we validate the ITL specification of this TM system with animation using AnaTempura

and by applying a concurrent data structure example. Moreover, we show how this stage in our framework helps to note a possible violation for doomed consistency within the chosen TM system. Finally, the correctness of the concrete specification model is demonstrated using refinement.

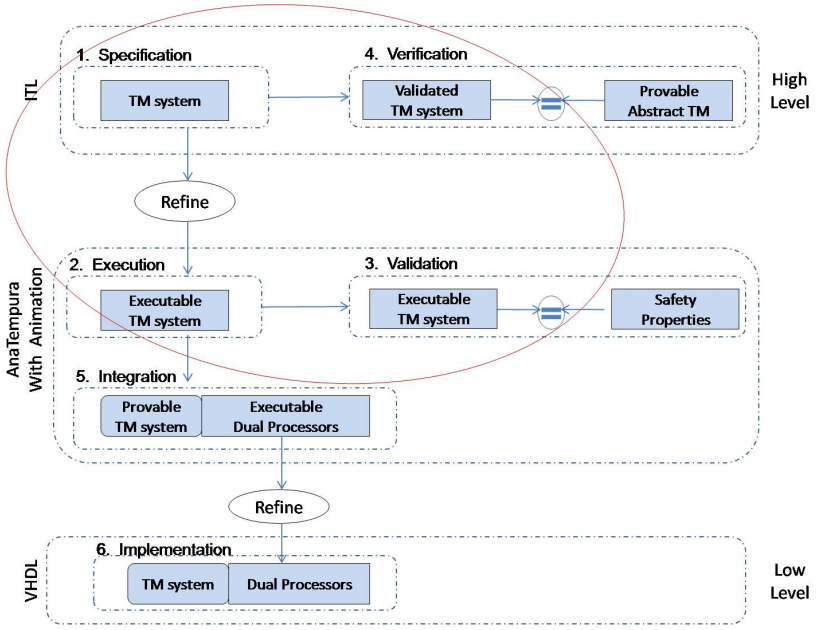

FIGURE 5.1: Proposed framework.

In this chapter, the provable abstract TM model described in the previous chapter is used as a basis for verifying the correctness of the chosen TM system.

5.2 Example of TM System

We provide, as an example, a TM system tm_{imp} which is firmly based on the original hardware transactional memory proposal by Herlihy and Moss [6]. In this section, important features of

the system tm_{imp} are briefly described. Then, an ITL specification for tm_{imp} is presented.

The system tm_{imp} detects a conflict between concurrent transactions using a coherency protocol and resolves conflicts by refusing to load data that already belongs to another active transaction. There is also a transactional cache in tm_{imp} for local transactional operations. Cache lines in the transactional cache have, in addition to a regular tag, a transaction tag which can hold one of the following values: x_{commit}, x_{abort}, $normal$ or $empty$. A line marked as x_{commit} indicates that the line contains data that was valid before entering the transactional mode, and may be used to recover from an aborted transaction. An x_{abort} tag indicates that the line contains data that was modified within the transaction; it is not visible to other processors (more details in the next section).

In addition, Herlihy and Moss [6] propose three instructions for accessing memory transitionally (they actually only access the transactional cache): Load-Transaction (lt): reads the value of a memory location into a private cache and then into a private register, Load-Transaction-Exclusive (ltx): reads the value of a memory location into a private cache and then into a private register which will be updated shortly, and Store-Transaction (st): tentatively writes a value into a private cache.

To make the transaction's tentative changes permanent and to simplify the writing of correct transactions, two more instructions were added: $commit$ and $validate$. These two instructions are maintained by adding two flags for each process: transaction active $Active$ and transaction status $Status$. The $Active$ flag indicates whether a transaction is in progress, while the $Status$ flag indicates whether that transaction is active (true) or aborted (false). The $Active$ is implicitly

set after the first lt or ltx instruction is executed and the object is read. The $Status$ flag is initially equal to true and resets when a conflict is detected by another concurrent transaction. While the $Active$ only resets when a transaction commits or a transaction aborts itself. The $commit$ and $validate$ instructions return an indicator of $Status$ flag; when this flag equals false , it discards all changes to the write set; when it equals true, $commit$ changes the write set to become permanent.

5.2.1 Cache and Coherency Specification

As we mentioned in the previous chapter (Section 4.2), the conflict detection mechanism is considered one of the most likely parts which may violate the correctness of a TM system. The tm_{imp} example depends on a transactional cache and a coherency protocol to detect a conflict between concurrent transactions. Thus, we present in this section a specification description for a full associative cache structure and a Modified-Exclusive-Shared-Invalid (MESI) coherency protocol [4] that will be used in the tm_{imp} specification.

Cache Structure

A full associative cache is a type of cache structure that allows a block from the main memory or other cache to be placed in any location in the cache. This is called fully associative because a block in the main memory may be associated with any entry in the cache and the entire cache must be searched every time data is accessed.

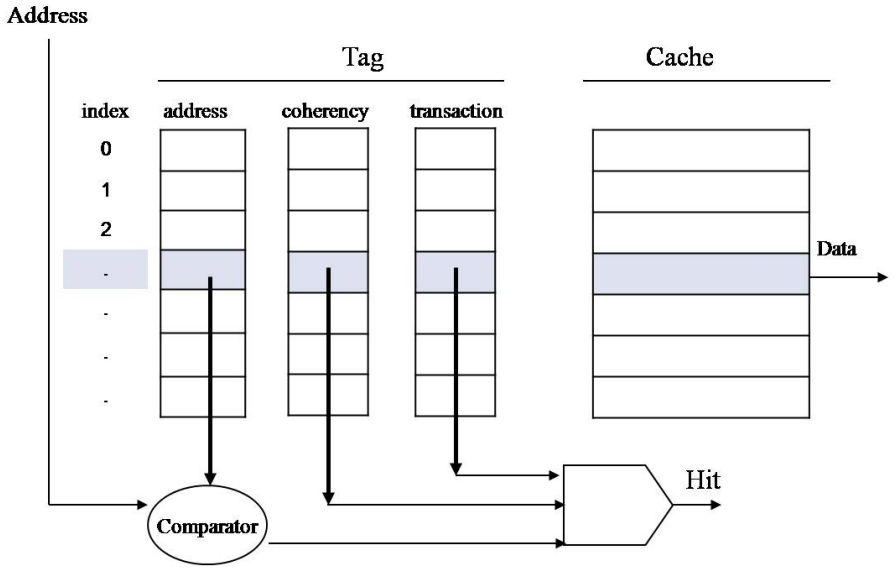

FIGURE 5.2: Cache and Tag Blocks

As shown in Fig. 5.2 the main state variables for the transactional cache are the data block *Cache* and its assistance components tag block *Tag* and flag *Hit*. The tag block is divided into the address, coherency status and transactional status of each data block. The coherency part for preserving the data consistency between each cache block and the main memory will be described in detail later in this section. Four states for a tag's transactional part are added to the regular coherency states to expand the task of preserving data consistency from a single cache block to a group of cache blocks. As shown in Fig. 5.3 the four transactional part states are:

- $empty$: The cache line does not contain a valid entry. It goes from $empty$ to x_{abort} or x_{commit}, if its location is chosen by a transaction to receive the first or second copy of a transactional data.

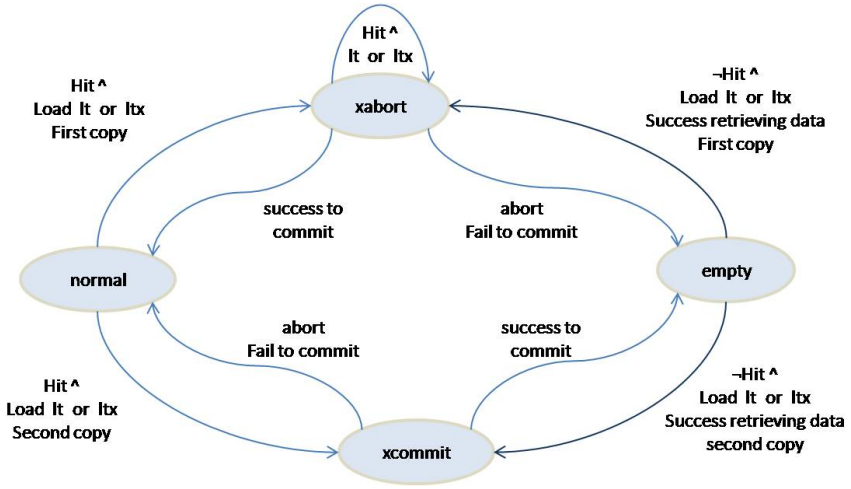

FIGURE 5.3: Transactional states diagram

- $normal$: The cache line contains a valid entry, which is not in a transactional mode and can be used by a local or external process. Also, it goes from $normal$ to x_{abort} or x_{commit}, if its location is chosen by a transaction to receive the first or second copy of transactional data.

- x_{commit}: The cache line contains data that is in a transactional mode. It can be used locally to recover from an aborted transaction. It goes from x_{commit} to $empty$ if the transaction is successfully committed, otherwise it goes to $normal$.

- x_{abort}: The cache line contains data that is in a transactional mode. It can be modified by the transaction. It goes from x_{abort} to $empty$ if the transaction is unsuccessfully committed, otherwise it goes to $normal$.

The Hit_p flag indicates the absence of the requested address in process p's local cache, where it is set when the requested address is found in the Tag_p's address part, the coherency for the

same address is valid and the transactional status is not equal to *empty*. For example, if a transactional operation is invoked by process p and the requested address equals *x* then the Hit_p flag for process p equals true if there exists an index i such that the following hold:

1. $Tag_p[i][address] = x$. The tag address of location i in cache p equals x.
2. $\neg(Tag_p[i][coherency] = invalid)$. The tag coherency status of location i does not equal invalid.
3. $\neg(Tag_p[i][transactional] = empty)$. The tag transactional status of location i does not equal empty.

In the other case where there is missing data in the local cache, the Hit_p flag is reset. The cache miss problem is handled as follows: As soon as missing data is detected and in the same state, a request is issued to retrieve the block containing the requested data from the main memory or other cache and the Hit_p flag is reset. In the next state and before invocation a new operation, the Hit_p flag is checked. Since it is found false, the same operation will be re-invoked to access the data that is retrieved in the previous state and then the Hit_p flag is set. For example, if a transactional operation op is invoked by process p then the following specification should be satisfied to handle cache miss state:

$p \mathrel{\widehat{=}} op;\ \big((\neg Hit_p \wedge op)\,{}^*\!\wedge\, fin\ Hit_p\big)$

MESI Coherency Protocol

The tag block in a cache data structure has a coherency status part for each data block to retain all copies of a main memory location in multiple caches consistent when the contents of that memory location are modified. Herlihy and Moss [6] proposed that any protocol capable of detecting conflicts can also detect transaction conflicts at no extra cost. Many protocols for cache coherency have been proposed [4, 68]. Here we use one of these coherency protocols that is close to Goodman's popular snoopy protocol for a shared bus [4].

Modified-Exclusive-Shared-Invalid (MESI) is a write-invalidate snoopy protocol used in many current systems. Most coherence protocols, including MESI, incorporate a write-invalidate strategy with write-back policy. The choice of this protocol was due to the efficiency and easy support for memory synchronization.

The following description is adapted from the one by Culler and Singh [4]. The protocol uses four states to encode the state of a cache block that resides in a processor's cache. These four states are:

- $invalid$: The cache line does not contain a valid entry.
- $shared$: The cache line contains a valid entry, which may be shared with other processors. The cache line is unmodified, i.e. it contains the same data as the corresponding memory location.

- *modified*: The cache line contains data that has been written to by the processor. The corresponding memory location has not been updated yet and therefore does not hold the actual value. The cache line is held exclusively by this cache, therefore read and write operations can be performed on this line without notification of the other processors.

- *exclusive*: The cache line is held exclusively by the respective cache, but is not modified. The purpose of this enhancement is to reduce snooping traffic in shared-memory systems by avoiding snooping messages for non-shared data. Systems running applications that do not or rarely communicate with each other profit especially from this enhancement.

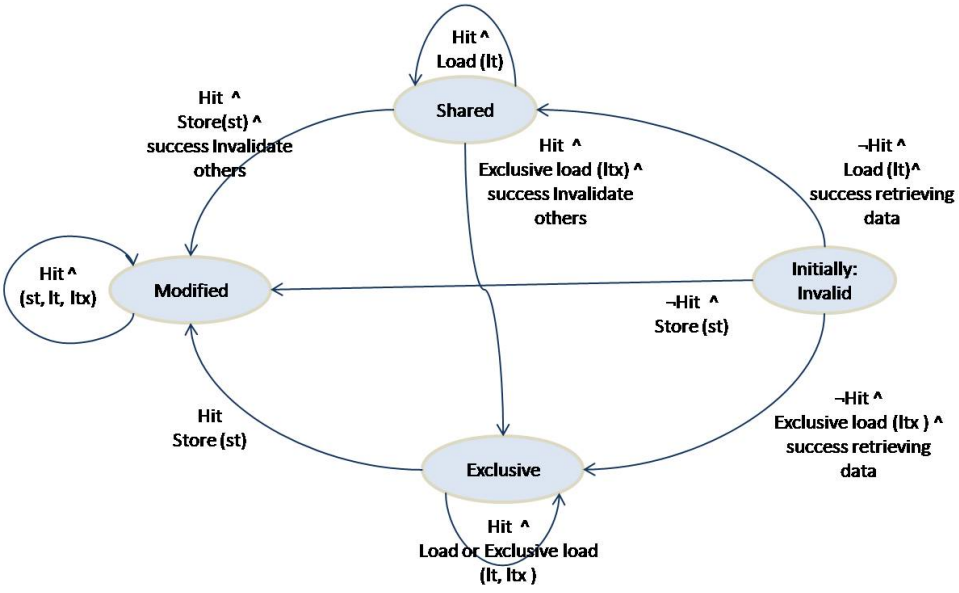

FIGURE 5.4: State diagram for the Request-Cache in the MESI protocol

The state diagram for the MESI protocol is divided into two diagrams: Fig. 5.4 shows the state diagram for the request operations, denoted by *Request-Cache*, which means a cache issues a request for either retrieving a data or invalidating other copies, while Fig. 5.5 shows the

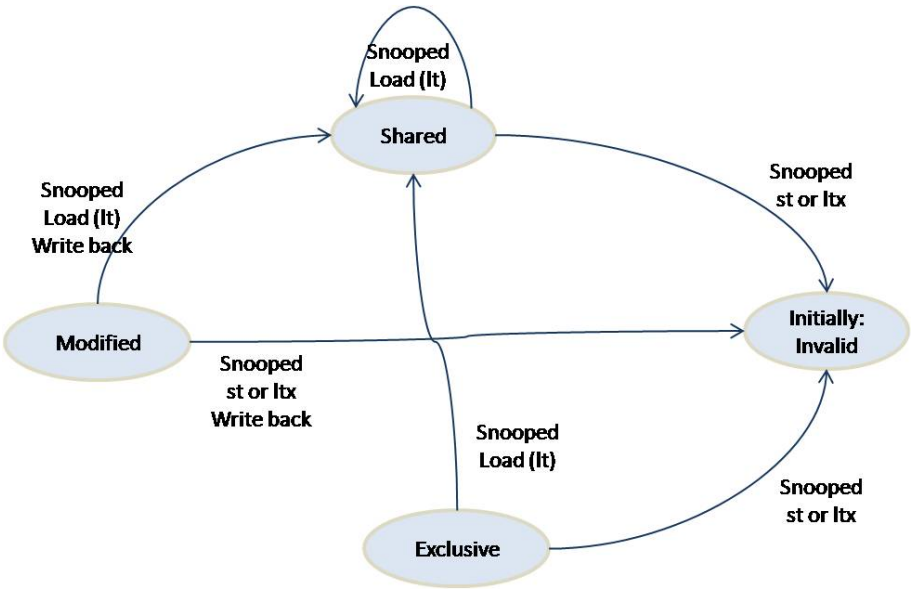

FIGURE 5.5: State diagram for the Snooped-Cache in the MESI protocol

state diagram for the snooping operations, denoted by *Snooped-Cache*, which means a cache snoops and handles demands of other caches. For example, if process p invokes $ltx(x)$ operation (*Request-Cache*) and process q's cache snoops others (*Snooped-Caches*) then before p becomes owner of the x line and changes its state to $exclusive$, all other caches holding this line such as q should convert x's coherency status to $invalid$.

The system tm_{imp} combines a tag's transactional status part with tag's coherency status part for detecting a conflict. In example just considered, where process q has the requested line x and this line is in transactional mode (transactional status equals x_{abort}), which means that the line x is still being used by an unfinished transaction and can't be used by others, then before converting the x's coherency status to $invalid$, its transactional status is read and the transactional mode case indicates that there is a conflict between the request process which is p and the owner which is q.

5.2.2 Specification of the TM System

An interval σ is a finite or infinite sequence of one or more states $s_0, s_1, s_2, ..., s_n$. Each state has concurrent observable events E. An event E_p that belongs to process p is either an invoke-response for the load and store transactional operations op, denoted by ***TraInvResOp(p,op)***, or an invoke-response for ending and status validating transactional operations op_e, denoted by ***TraInvResEnd(p,op_e)***.

The invocation and response events of transactional operations ***TraInvResOp(p,op)*** are described as follows:

- $lt_p(x)$: a load-transactional-operation reads the current value of x from $Cache_p$ by process p. If $Cache_p$ doesn't have x, a request is issued and the immediate interim response is $\perp$.
- $ltx_p(x)$: a load-transactional-exclusive-operation is the same as lt but also invalidates any copies of x in other caches.
- $st_p(y, u)$: a store-transactional-operation by process p to write a value u to location y in $Cache_p$. We assume that each operation st has been preceded by ltx, so there is no data missing.

The invocation and response events of ***TraInvResEnd(p,op_e)*** are described as follows:

- $commit_p$: a commit request that is issued by process p. If the attempt to commit succeeds, it will change all x_{abort} entries of the tag $Cache_p$ to $normal$ and all the x_{commit} entries to

$empty$. If it fails, all x_{abort} entries will be changed to $empty$ and all the x_{commit} entries to $normal$.

- $abort_p$: an abort request that is issued by process p. It changes all x_{abort} entries to $empty$ and all the x_{commit} entries to $normal$.

- $validate_p$: a validate for transaction status. If the transaction is active, then no action or else the same as $abort_p$.

Components of tm_{imp}

As shown in Fig. 5.6, the main components of tm_{imp} are:

- $Mem_{imp}[obj]$: Global memory, where obj: $(0 \leq obj < |Locations|)$, initially $\perp$.

- $Cache[obj]$: Each process has its own cache, $(0 \leq obj < |Cache|)$, initially $\perp$. Also, each cache has a Hit flag and each data block in the cache has a Tag.

- $Active$: An array $[1..p]$ of booleans recording which process is in progress, initially all $false$.

- $Status$: An array $[1..p]$ of booleans recording which transaction is aborted, initially all $true$.

- E_{imp} : Event array for each process as an auxiliary history to record each event, initially no_{ev}.

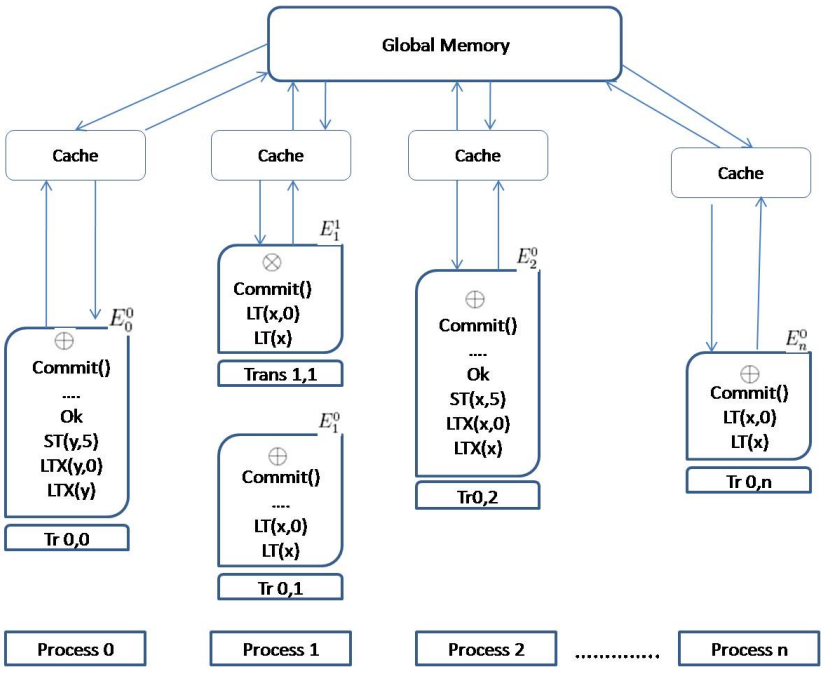

FIGURE 5.6: The proposed TM abstract model.

Transition Behaviour

The state transition for the tm_{imp} is, for better readability, partitioned into two tables: Table 5.1 lists all possible invocation states of tm_{imp} involving *TranInvResOp()* ($m_0 - m_2$) and *TranInvResEnd()* ($m_3 - m_5$), while Table 5.2 lists all possible responses for invocation of transactional operations m_0 and m_1. Both tables describe the preconditions under which the transition can be taken and describe the effects of the transition on other variables.

The output of the m_0 and m_1 is cleared in the event list column of the second table, while the output of $m_2 - m_4$ is either commit or abort. Each one of the first two invocation cases of Table 5.1 m_0 and m_1 has one possible response from the transitions list $k_0 - k_9$ in Table 5.2.

The following definitions will be considered to make conjunctions between the two tables:

$$\begin{aligned} TranInvResOp(p, op) &\;\widehat{=}\; \bigvee_{i=0}^{9} \widehat{k_i} \vee m_2 \\ \widehat{k_i} &\;\widehat{=}\; k_i \wedge \bigvee_{j=0}^{1} m_j \\ TranInvResEnd(p, op_e) &\;\widehat{=}\; \bigvee_{i=3}^{5} m_i \end{aligned} \tag{5.1}$$

TABLE 5.1: The invocation actions of tm_{imp}

Invocation Cases			
Case		Preconditions	Actions
$TranInvResOp(p, op)$	m_0	$\neg Active_p \wedge Status_p$	$skip \wedge \bigcirc Active_p = true$ $\wedge FindData()$
	m_1	$Active_p$	$skip \wedge stable(Active_p)$ $\wedge FindData()$
	m_2	$otherwise$	$skip \wedge AbortRes(p)$
$TranInvResEnd(p, op_e)$	m_3	$\neg Status_p \vee op_e = abort$	$skip \wedge AbortRes(p)$
	m_4	$op_e = commit \wedge Status_p$	$skip$ $\wedge CommitRes(p, op_e)$
	m_5	$op_e = validate \wedge Status_p$	$skip \wedge stable(Active_p)$

In Table 5.1 the transitions $m_0 - m_2$ deal with the *ltx, lt and st* transactional operations. The actions are invocations for an operation using a new transaction, invocation for an operation using the same transaction or aborting a transaction. For example, the preconditions for m_1 are an invocation of a transactional operation op and the process status flag $Active$ equals *true*. The actions of m_1 are a determination of interval length and a retrieving of requested data. The formula *FindData()* (defined later in Fig. 5.7) searches for the requested data in the local cache and responds with one of the possible cases in Table 5.2. Transitions $m_3 - m_5$ deal with *commit, abort and validate* instructions. The actions are either committing a transaction (making all the tentatively updates permanent using *CommitRes()*) or aborting a transaction

(undoing any update using *AbortRes()*). Here are the formal definitions of *CommitRes()* and *AbortRes()* (see transactional states diagram Fig. 5.3):

$$
\begin{aligned}
&CommitRes(p, op_e) \\
&\widehat{=} \quad \bigcirc Status_p = true \wedge \bigcirc Active_p = false \wedge \bigcirc E_{imp} = \oplus \\
&\qquad \wedge \forall\, i < |Cache| : \big((Tag_p[i][transaction] = x_{abort} \\
&\qquad\qquad \supset \bigcirc Tag_p[i][transaction] = normal) \\
&\qquad\qquad \wedge (Tag_p[i][transaction] = x_{commit} \\
&\qquad\qquad \supset \bigcirc Tag_p[i][transaction] = empty)\big)
\end{aligned}
$$

$$
\begin{aligned}
&AbortRes(p, op_e) \\
&\widehat{=} \quad \bigcirc Status_p = true \wedge \bigcirc Active_p = false \wedge \bigcirc E_{imp} = \otimes \\
&\qquad \wedge \forall\, i < |Cache| : \big((Tag_p[i][transaction] = x_{abort} \\
&\qquad\qquad \supset \bigcirc Tag_p[i][transaction] = empty) \\
&\qquad\qquad \wedge (Tag_p[i][transaction] = x_{commit} \\
&\qquad\qquad \supset \bigcirc Tag_p[i][transaction] = normal)\big)
\end{aligned}
$$

Table 5.2 describes all possible responses of the *FindData()* formula mentioned in Table 5.1. The list of possible response $k_0 - k_9$ are categorised into two main parts, which are the cache's data response actions for hits *HitRes()* and misses *MissRes()* (defined later in Fig. 5.8).

The following abbreviations are used in Table 5.2:

- *Hit* is a flag to indicate that an invocation is a success and a new operation can be invoked.

TABLE 5.2: The response actions of tm_{imp}'s transactional operations

Response Cases				
Case*		Preconditions	Actions	Event
$MissRes(p, ltx, x)$	k_0	$Status_p \wedge \neg ReqReady$	$\bigcirc\ Hit{=}false \wedge$ $ReqData(p, op, x)$	$ltx(x)$
	k_1	$Status_p \wedge$ $ReqReady \wedge$ $MemRes$	$\bigcirc\ Hit{=}false \wedge$ $Allocate2Loc(p) \wedge$ $Cache[ft]{=}Mem_{imp}[x] \wedge$ $Cache[sd]{=}Mem_{imp}[x] \wedge$ $UpdateTag(p, x, ft, sd)$	$ltx(x, u)$ u= Mem_{imp} [x]
	k_2	$Status_p \wedge ReqReady \wedge$ $\neg MemRes$	$\bigcirc\ Hit{=}false \wedge$ $Allocate2Loc(p) \wedge$ $Mem_{imp}[x]{=}Bus \wedge$ $Cache[ft]{=}Bus \wedge$ $Cache[sd]{=}Bus \wedge$ $UpdateTag(p, x, ft, sd)$	$ltx(x, u)$ u=Bus
	k_3	$\neg Status_p$	$\bigcirc\ Hit{=}true \wedge$ $BusyRes(p, op, x)$	$ltx(x, u)$ u= Random
$HitRes(p, ltx, x, \perp)$	k_4	$Status_p \wedge$ $Tag_p[i][c] = shared \wedge$ $\neg ReqReady$	$\bigcirc\ Hit = false \wedge$ $ReqInvOther(p, op, add)$	$ltx(x)$
	k_5	$\neg(Tag_p[i][c] = shared)$ $\vee\ (Tag_p[i][c] = shared$ $\wedge\ ReqReady$ $\wedge\ Status_p)$	$\bigcirc\ Hit = true \wedge$ $u = Cache[ft] \wedge$ $UpdateTag(p, x, ft, sd)$	$ltx(x, u)$
	k_6	$Tag_p[i][c] = shared \wedge$ $ReqReady \wedge$ $\neg Status_p$	$\bigcirc\ Hit = true \wedge$ $BusyRes(p, op, x)$	$ltx(x, u)$ u= Random
$MissRes(p, lt, x)$	k_7	The same as one of k_0-k_3	The same as one of k_0- k_3	$lt(x)$
$HitRes(p, lt, x, \perp)$	k_8	no further condition	The same actions as k_5	$lt(x, u)$
$HitRes(p, st, x, u)$	k_9	for simplicity we assume that each st has ltx before, so there is no miss	Cache[ft]=u$\wedge$ $UpdateTag(p, x, ft, sd)$	ok

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

* This table associated with Fig. 5.7 later on.

- *ReqReady* is a flag to indicate that there is a request for owning data and the response is ready to be received.

- *MemRes* is a flag to indicate that the missing data is retrieved from the main memory, or from another cache and the data is ready in a state variable called *Bus*.

- *Allocate2Loc(p)* is a replacement technique that searches for two locations in $Cache_p$ when this cache needs space for a new entry. It first searches for an $empty$ entry, then for a $normal$ entry and finally for a x_{commit} entry. If the $normal$ or the x_{commit} entry is in modifying state, it issues a request to write back the replacement data cache block, see Appendix B.

- *UpdateTag(p,x,ft,sd)* updates the tag's coherency and transactional status of the two locations ft and sd according to the *MESI* protocol and the transactional operations.

- *ReqData(p,op,x)* is a task that brings data of address x from one of the other caches or from the main memory after invalidating other copies to prevent conflict in the case of ltx and st. Moreover, *ReqData()* works as *ConflictDetRes()* in tm_{spec}, where it detects a conflict with x when it finds x in another cache with transactional status equal to x_{abort}. In addition, it solves the conflict by resetting its $Status_p$.

  Here is the formal definition of *ReqData()* formula:

  (note: a=address, c=coherency and t=transactional).

$$
\begin{aligned}
&ReqData(p, op, x)\\
&\quad \widehat{=} \quad \bigcirc ReqReady = true\\
&\qquad \wedge \Big(\big(\bigcirc MemRes = false\\
&\qquad\quad \wedge \big((z \wedge \bigcirc Status_p = false) \vee (\neg z \wedge z' \wedge stable(Status_p))\big) \big)\\
&\qquad \vee \Big(\neg z \wedge \neg z' \wedge z'' \wedge \bigcirc Bus = Mem[x] \wedge \bigcirc MemRes = true\\
&\qquad\quad \wedge stable(Status_p) \Big) \Big)\\
&z \widehat{=} \quad \exists i < |Cache|, p \neq q :\\
&\qquad Tag_q[i][a] = x \wedge Tag_q[i][t] = x_{abort}\\
&\qquad \wedge \big(Tag_q[i][c] = modified \vee (Tag_q[i][c] = shared \wedge \neg(op = lt)) \big)\\
&\qquad \wedge \bigcirc Bus = \bot\\
&z' \widehat{=} \quad \exists i < |Cache|, p \neq q :\\
&\qquad Tag_q[i][a] = x \wedge Tag_q[i][t] = normal \wedge Tag_q[i][c] = modified\\
&\qquad \wedge \Big((\neg(op = lt) \wedge \bigcirc Tag_q[i][c] = invalid)\\
&\qquad\quad \vee (op = lt \wedge \bigcirc Tag_q[i][c] = shared) \Big) \wedge Bus = Cache_q[i]\\
&z'' \widehat{=} \quad \forall q < |Processes| : \exists i < |cache| :\\
&\qquad (z''' \wedge \bigcirc Tag_q[i][c] = invalid) \vee (\neg z''' \wedge stable(Tag_q[i]))\\
&z''' \widehat{=} \quad Tag_q[i][a] = x \wedge Tag_q[i][t] = normal\\
&\qquad \wedge Tag_q[i][c] = shared \wedge \neg(op = lt)
\end{aligned}
$$

- ***ReqInvOther(p,op,x)*** is similar to the ***ReqData()***, but just invalidates other copies of x and does not transfer any data.

- ***BusyRes()*** is a formula that is activated when its transaction has a conflict ($Status$ equals false) and there is an invocation of a transactional operation op. ***BusyRes()*** keeps the

status of transactional flag $Status$, drops all x_{abort} entries and sets all x_{commit} entries to $normal$.

Here is the formal definition of ***BusyRes()***:

BusyRes(p,op,add)

$$\begin{aligned}
\widehat{=} \quad & stable(Status_p) \\
& \wedge \forall\, i < |Cache| : \big((Tag_p[i][transaction] = x_{abort} \\
& \qquad \supset \bigcirc Tag_p[i][transaction] = empty) \\
& \qquad \wedge (Tag_p[i][transaction] = x_{commit} \\
& \qquad \supset \bigcirc Tag_p[i][transaction] = normal)\big)
\end{aligned}$$

The formula ***MissRes(p,ltx,x)*** in Table 5.2 deals with missing data that has address x in local cache $Cache_p$ for the ltx operation case. Where,

- k_0 issues a request for retrieving the missing data using ***ReqData()***.
- k_1 or k_2 retrieves a block containing the missing data from the main memory or the other caches respectively. They first allocate two locations (ft and sd) in $Cache_p$ using the $Allocate2Loc(p)$ formula and then copy the retrieved data in the first and second location with transactional status x_{abort} and x_{commit}, respectively. In addition, the transition k_2 updates the main memory data location to preserve data consistency.
- k_3 keeps the status of transactional flag $Status$, undoes any tentatively update using ***BusyRes*** and returns arbitrary data when a conflict is detected by ***ReqData()*** that is issued by k_0.

The *MissRes(p,ltx,x)* transitions are activated when there is no valid copy of the requested address x in $Cache_p$ and the following formula holds:

$$\neg \big(\exists i < |Cache| : Tag_p[i][address] = x \\ \wedge \neg(Tag_p[i][coherency] = invalid \\ \vee Tag_p[i][transactional] = empty)\big) \tag{5.2}$$

Otherwise the *HitRes(p,ltx,x,⊥)* states are activated. Where,

- k_4 issues a request for invalidating other copies of x in the other caches using *ReqInvOther()*. Although, the transition of k_4 implies that there is a valid copy of x in $Cache_p$, its coherency state is $shared$ and the operation is ltx which means that $Cache_p$ should own x exclusively and other copies should be invalidated.

- k_5 reads the value of address x from $Cache_p$ when its coherency state is not $shared$ or a response for invalidating other copies without conflict detection is received.

- k_6 discards any update by its transaction using *BusyRes* and returns arbitrary data when *ReqInvOther()* that is issued by k_5 detects a conflict and resets the transactional status flag $Status$.

- k_7 deals with the *MissRes(p,lt,x)* case of the lt operation. Although, k_7 can be as one of the ltx operation's states $k_0 - k_3$, the only difference is the *ReqData()* and *UpdateTag()* tasks. In case of lt the*ReqData()* is more weaker than ltx, for example to retrieve x at

case of lt it is sufficient to regard in the other cache the x's coherency status $shared$ and neglect x's transactional status.

- k_8 reads the value of address x from $Cache_p$ as in case k_5. The activation of the formula ***HitRes(p,lt,x,⊥)*** is only the condition for this state.

- k_9 deals with the ***HitRes(p,st,x,u)*** case of st operation. For simplicity, we assume that there is a ltx operation before any st, so there are no preconditions and miss cases for a st operation.

A sequence of events of the form $((\mathit{TranInvResOp}(p, op))^*;\ \mathit{TranInvResEnd}(p, op_e))$ is issued sequentially by process p and this sequence is called a transaction.

The complete tm_{imp} system description can be modelled with the initial values in the following ITL formula:

$$
\begin{aligned}
tm_{imp} \mathrel{\widehat{=}} \quad & \textstyle\bigwedge_{p=0}^{n} Processes_p \\
Processes_p \mathrel{\widehat{=}} \quad & Active_p = \mathit{false} \wedge \mathit{Status}_p = \mathit{true} \wedge \mathit{Hit}_p = \mathit{true} \\
& \wedge \Big(\big(\ \mathit{TranInvResOp}(p, op) \\
& \qquad ; \big((\neg \mathit{Hit}_p \wedge \mathit{TranInvResOp}(p, op))^* \wedge \mathit{fin}\ \mathit{Hit}_p \ \big) \big)^* \\
& \qquad ; \ \mathit{TranInvResEnd}(p, op_e) \Big)^*
\end{aligned}
\tag{5.3}
$$

As we explained previously, the Hit_p flag indicates the success of the invocation of a transactional operation. If it equals *true*, a new operation can be invoked by the same transaction. If the Hit_p flag equals *false*, the same invocation should be repeated until Hit_p equals *true*. However,

there is a special case that sets the Hit_p flag to *true* even with invocation failure. This is when a conflict detection with other concurrent transactions occurs. The setting of the Hit_p flag in this case prevents the repetition of the same conflicted invocation.

5.3 Execution and Validation

To demonstrate and validate the correctness of the tm_{imp}'s specification and make such examinations for TM safety properties, we use refinement rules for making the ITL specification of tm_{imp} executable [17, 18]. Then we use the executable specification version of tm_{imp} to execute one of the most highly studied concurrent data structures, the lock-free FIFO queue [25, 69], using AnaTempura. In addition, some animation 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 [70] using Expect [71]. The Tempura file is accompanied by a Tcl/Tk file which defines the graphics.

In fact, many lock-free queue algorithms have been proposed based on atomic instructions such as Compare-And-Set (CAS) and Load-Linked Store-Conditional (LL-SC) [25, 67, 72]. We will use an approach based on transactional memory, with some modification such as an additional shared counter. Our approach can give initial indicators for satisfying TM safety conditions but cannot guarantee the correction satisfaction of all cases.

5.3.1 Executable Specification of the TM System

The formal specification of tm_{imp}, that was first discussed in Subsection 5.2.2, is refined into AnaTempura so that it can be executed. As shown in Fig. 5.7 the main state variables of tm_{imp} are represented as follows:

- $Active$ is a list of processes statuses with state values $\{false, true\}$.
- $Status$ is a list of transactions statuses with values $\{false, true\}$.
- E_{imp} is an auxiliary history list for each process to record each state with one of the following possible values $\{no_{ev}, lt, ltx, st, ok, commit, \oplus, abort, \otimes\}$.
- Mem_{imp} is a list of objects for representing the global memory.
- $Cache$ is a list of object for each process. Each cache has a Tag list and Hit flag.

To help the understanding, the executable specification is written in functional form and divided into two main parts which are invocation Fig. 5.7 and response Fig. 5.8 of an operation.

The checking of the transaction's start is placed in the invocation of an operation part. If process status flag $Active$ of process p is $false$ and transaction status flag $Status$ of the same process is $true$, then p can invoke a new transaction by setting the status of p and an operation can be invoked too. Otherwise the same transaction can be used to invoke an operation. This prevents other transactions from being created until the existing one terminates.

The *FindData()* is a recursion function that triggers either the *HitRes()* function when the requested address is found in the local cache, or the *MissRes()* otherwise. Fig. 5.8 shows the main actions and responses of the *HitRes()* and *MissRes()* functions that were explained in Subsection 5.2.2.

Some auxiliary functions are used in the executable model of the tm_{imp} such as the $AddEv()$ that is used to record each operation and its response in their process p event list E_{imp}. The function $FlushEvList()$ clears the event list of the process p after finishing the execution of a transaction belonging to p and before initialising a new transaction.

The complete executable specification of tm_{imp} that described in the ITL formula 5.3 can be represented in Tempura as follows:

$$\begin{aligned} tm_{imp} \mathrel{\widehat{=}} & \bigwedge_{p=0}^{n} Processes_p \\ Processes_p \mathrel{\widehat{=}} & \; Active_p = false \wedge Status_p = true \wedge Hit_p = true \\ & \wedge \Big(\big(\text{ repeat } TranInvResOp(p, op) \text{ until } Hit_p \big) * \\ & \quad ; \; TranInvResEnd(p, op_e) \Big) * \end{aligned} \tag{5.4}$$

5.3.2 Queue Example

A concurrent queue is an abstract data structure that consists of two processes. The producer process adds the element x to the rear terminal position, if the queue is not full. The consumer process retrieves the element from the front terminal position, if the queue is not empty [25].

State variables:
$Active_p$: Process flag, where p: $(0 \leq p < |Processes|)$, initially $false$.
$Status_p$: Transaction flag, where p: $(0 \leq p < |Processes|)$, initially $true$.
$Mem_{imp}[obj]$: List of object, where obj: $(0 \leq obj < |Locations|)$, initially $\perp$.
$Cache_p[obj]$: List of object for each process $(0 \leq obj < |Cache|)$, initially $\perp$.
$Tag_p[a \cup c \cup t]$: Tag for each $Cache_p[obj]$ to maintain coherency, where $a \in obj$, $c \in \{invalid, shared, modified\}$, $t \in \{normal, x_{abort}, x_{commit}, empty\}$, initially $[\perp \cup invalid \cup empty]$.
Hit_p: Hit flag, where p: $(0 \leq p < |processes|)$, initially $true$.
E_{imp} : List of events for each process $\in \{no_{ev}, lt, ltx, st, validate, ok, commit, \oplus, abort, \otimes\}$, initially no_{ev}.

Transaction operations:

$TranInvResOp(p, op, add, val) \mathrel{\widehat{=}}$
$\{skip \wedge$
if $((\neg Active_p) \wedge (Status_p))$
then $\{MakeProBusy(p) \wedge FindData(p, op, add, val, 0)\}$
else if $(Active_p)$
then $\{stable(Active_p) \wedge FindData(p, op, add, val, 0)\}$
else $AbortRes(p, op)$
$\}$

$TranInvResEnd(p, op_e) \mathrel{\widehat{=}}$
$\{skip \wedge$
if $(Status_p)$
then if $(op_e = commit)$
then $CommitRes(p, op_e)$
else $\{stable(Status_p) \wedge stable(Active_p)\}$
else $AbortRes(p, op_e)$
$\}$

$FindDatat(p, op, add, val, i) \mathrel{\widehat{=}}$
$\{$if $i = |Locations|$
then if $(\neg ReqReady \wedge Status_p)$
then $\{ReqData(p, op, add) \wedge Hit := false \wedge AddEv(p, op, add, \perp)\}$
else $MissRes(p, op, add)$
else if $(Tag_p[i][a] = add \wedge \neg(Tag_p[i][c] = invalid \vee Tag_p[i][t] = empty))$
then if $(op = ltx \wedge Tag_p[i][c] = shared \wedge \neg ReqReady \wedge Status_p)$
then $\{ReqInvOther(p, op, add) \wedge Hit := false \wedge AddEv(p, op, add, val)\}$
else $\{HitRes(p, op, add, val, i) \wedge Hit := true\}$
else $FindData(p, op, add, val, i + 1)$
$\}$

$MakeProBusy(p) \mathrel{\widehat{=}} \{Active_p := true\}$

FIGURE 5.7: First core part of tm_{imp} executable specification

$HitRes(p, op, add, val, i) \mathrel{\widehat{=}}$
$\{\exists ft, sd, allocate : \{$
if $Tag_p[i][t] = normal$
  then $(ft = i \wedge sd = AllocateXcom(p, empty1, i, 0) \wedge allocate = 1)$
  else if $Tag_p[i][t] = x_{abort}$
    then $(ft = i \wedge sd = FindSDcopy(p, x_{commit}, add, 0) \wedge allocate = 0)$
    else $(ft = FindSDcopy(p, x_{abort}, add, 0) \wedge sd = i \wedge allocate = 0)$

$\wedge$if $op = st$
  then $\{Cache(p, 0, write, val, ft, sd, allocate, 0, 0, 0) \wedge stable(Status_p)$
    $\wedge\, UpdateTag(p, op, add, ft, sd, allocate)$
    $\wedge\, AddEv(p, op, add, val)\}$
  else if $(op = lt \vee (op = ltx \wedge (Tag_p[ft][c] \neq shared \vee (Tag_p[ft][c] = shared \wedge Status_p))))$
    then $\{Cache(p, read, 0, \perp, ft, sd, allocate, 0, 0, 0) \wedge stable(Status_p)$
      $\wedge\, UpdateTag(p, op, add, ft, sd, allocate)$
      $\wedge\, AddEv(p, op, add, u = CacheOut)\}$
    else $BusyRes(p)\}$
    $\}$

$MissRes(p, op, add) \mathrel{\widehat{=}}$
$\{Allocate2Loc(p)$
  $\wedge$if $(Status_p)$
  then $\{Hit := false \wedge stable(Status_p)$
    $\wedge$ if $(MemRes)$
      then $\{Cache(p, 0, write, Mem[add], ft, sd, 0, 1, 0, 0)$
        $\wedge\, UpdateTag(p, op, add, ft, sd, allocate)$
        $\wedge\, AddEv(p, op, add, u = Mem[add])\}$
      else $\{Memory(write, add, Bus)$
        $\wedge\, Cache(p, 0, write, Bus, ft, sd, 0, 1, 0, 0)$
        $\wedge\, UpdateTag(p, op, add, ft, sd, allocate)$
        $\wedge\, AddEv(p, op, add, u = Bus)\}\}$

  else $\{ BusyRes(p) \wedge Hit := true \}$
$\}$

$CommitRes(p, op) \mathrel{\widehat{=}}$
$\{Status_p := true \wedge MakeProFree(p)$
$\wedge\, UpdateTag(p, op, \perp, \perp, \perp, \perp)$
$\wedge\, AddEv(p, op, \perp, \oplus)\}$

$AbortRes(p, op) \mathrel{\widehat{=}}$
$\{Status_p := true \wedge MakeProFree(p)$
$\wedge\, UpdateTag(p, op, \perp, \perp, \perp, \perp)$
$\wedge\, AddEv(p, op, \perp, \otimes)\}$

$MakeProFree(p) \mathrel{\widehat{=}} \{Active_p := false \wedge FlushEvList(p)\}$

FIGURE 5.8: Second core part of tm_{imp} executable specification

Consider the FIFO queue system shown in Fig. 5.9. It stores its elements in memory, which, for simplicity, we will assume is a fixed queue with two additional memory elements for the indices ($head = mem[0]$, $tail = mem[1]$).

```
proc Initialize() ≡
  {head = 0;                          location of head in memory
  mem[head] = 3;                      first location of queue
  tail = 1;                           location of tail in memory
  mem[tail] = 3;                      first location of queue
  mem[2] = 0; }                       start shared counter

proc Producer() ≡
  {Phead = read(mem[head]);
  Ptail = read(mem[tail]);
  if (Ptail − Phead = Qsize)
    then Abort()
    else
        {Pshared = read(mem[Ptail − 1]);
        write(mem[Ptail], Pshared + 1);
        write(mem[tail], Ptail + 1);
        Commit(); }
        }

proc Consumer() ≡
 {Chead = read(mem[head]);
  Ctail = read(mem[tail]);
    if (Chead = Ctail)
      then Abort();
    else
     {Cshared = read(mem[Chead]);
      write(mem[head], Chead + 1);
          Commit(); }
       }
```

FIGURE 5.9: Concurrent queue algorithm

The left process, produces a shared counter by reading the number at the end of the queue, incrementing it, then extending the queue and putting the incremented one at the end of the queue. The right process, consumes the shared counter, for simplicity, by just reading the first element in the queue and decreases the head of the queue.

The first index points to the $head$ of the queue and the second 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 (see Fig. 5.10).

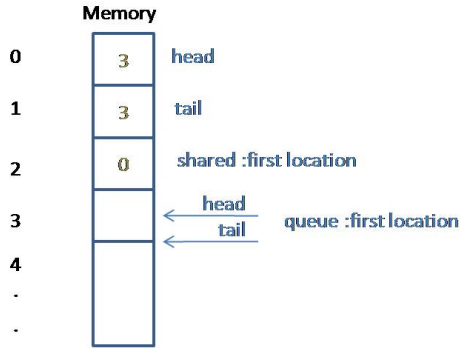

FIGURE 5.10: Queue example: memory initialize

If the producer process, after reading $head$ and $tail$, finds that the queue is full, then it fails. Otherwise, it will read and increment the shared counter (initially at $mem[2]$) at the point of memory entry $(tail - 1)$ and stores the incremented shared counter at the memory entry $tail$, and then increments $tail$. If the consumer process, after reading $head$ and $tail$, finds that the queue is empty, by checking the equality of $head$ and $tail$, then it fails. Otherwise, it will read the shared counter at the memory entry $head$, and then increment $head$ (see Fig. 5.11).

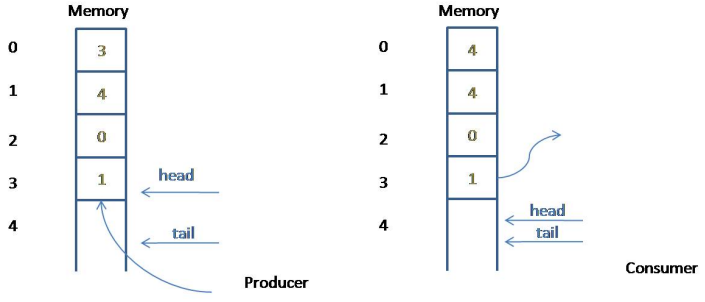

FIGURE 5.11: Queue example: produce and consume the first shared counter

5.3.3 Queue with TM Execution and Animation

To execute the concurrent queue algorithm using the tm_{imp} executable specification and for seeking simplicity, we just used two concurrent processes to represent the producer and consumer. Two additional functions are used:

- *QueueFullCheck()* for the producer process that checks if the queue is full.
- *QueueEmptyCheck()* for the consumer process that checks if the queue is empty.

Moreover, seven state variables and two static variables (memory address) $head$ and $tail$ are used. The state variables are *Phead, Chead* with the initial value *Mem[head]*, *Ptail, Ctail* with initial value *Mem[tail]* and *Pshared, Cshared* equal to $\perp$. Here is the complete executable form of a concurrent queue algorithm with TM: $tm_{imp}q \mathrel{\widehat{=}}$ PRODUCER$_{spec}$ $\wedge$ CONSUMER$_{spec}$

PRODUCER$_{spec}$

$$
\begin{aligned}
\mathrel{\widehat{=}}\ & Active_p = false \wedge Status_p = true \wedge Hit_p = true \\
& \wedge \Big(\big((\text{ repeat } TranInvResOp(p, lt, head, \perp) \text{ until } Hit_p \text{ }); \\
& \qquad (\text{ repeat } TranInvResOp(p, ltx, tail, \perp) \text{ until } Hit_p \text{ }); \\
& \qquad QueueFullCheck() \big)^*; \\
& \quad (\text{ repeat } TranInvResOp(p, ltx, Ptail - 1, \perp) \text{ until } Hit_p \text{ }); \\
& \quad (\text{ repeat } TranInvResOp(p, st, Ptail, Pshared + 1) \text{ until } Hit_p \text{ }); \\
& \quad (\text{ repeat } TranInvResOp(p, st, tail, Ptail + 1) \text{ until } Hit_p \text{ }) \\
& \quad ;\ TranInvResEnd(p, commit) \Big)^*
\end{aligned}
$$

CONSUMER$_{spec}$

$$
\begin{aligned}
\widehat{=} \quad & Active_c = false \wedge Status_c = true \wedge Hit_c = true \\
& \wedge \Big(\big(\text{(repeat } TranInvResOp(c, ltx, head, \perp) \text{ until } Hit_c \text{)}; \\
& \qquad \text{(repeat } TranInvResOp(c, lt, tail, \perp) \text{ until } Hit_c \text{)}; \\
& \qquad QueueEmptyCheck() \big)^*; \\
& \quad \text{(repeat } TranInvResOp(c, ltx, Ctail - 1, \perp) \text{ until } Hit_c \text{)}; \\
& \quad \text{(repeat } TranInvResOp(c, st, tail, Chead + 1) \text{ until } Hit_c \text{)} \\
& \quad ; \ TranInvResEnd(c, commit) \Big)^*
\end{aligned}
$$

As shown in Figs. 5.12 and 5.13, the user interface for the graphical output is divided into four parts:

1. The timer axis which represents the state number.
2. Two processes where P_0 represents the producer specification $PRODUCER_{spec}$ and P_1 represents the consumer specification $CONSUMER_{spec}$.
3. Two caches, where each process owns its local memory and contains 7 locations.
4. An area between the process number and its cache for showing the transaction number and its sequence of operations and responses.

We can notice clearly the memory synchronization in the output concurrent execution between the $CONSUMER_{spec}$ and $PRODUCER_{spec}$ processes. The $PRODUCER_{spec}$ process starts by producing a shared counter and updating the value of $tail$, while the $CONSUMER_{spec}$ process

aborts its self because of the refusal from the $PRODUCER_{spec}$ to revoke $head$. As soon as the ownership of $head$ and $tail$ of the queue is revoked by the producer, the consumer owns these variables, updates the $head$ and consumes the shared counter.

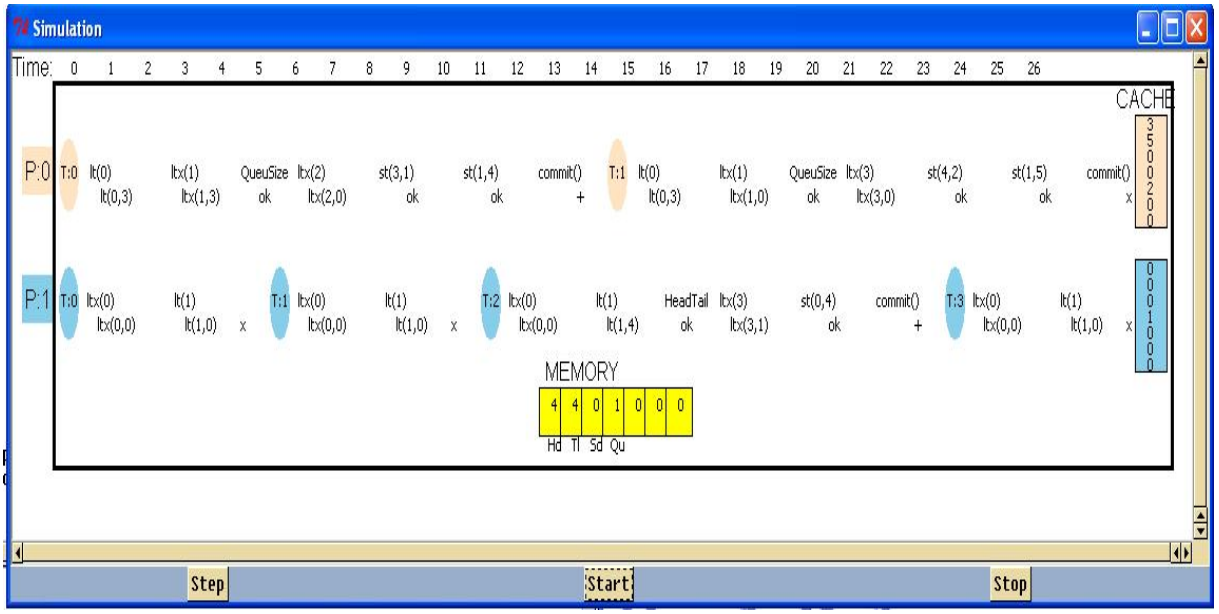

FIGURE 5.12: Queue with TM system $tm_{imp}q$ output before modification

This model represents the lazy resolution for conflict. It is cleared in the beginning that P_1 continues to execute after it has a conflict and returns random data (0), because the owning of location 0 by P_0. It is re-invoked because the queue is empty. The same thing for P_0, it continues to execute and aborts its self at the commit time.

The graphical animation has facilities to execute this example step by step. We can therefore check and validate that the TM safety conditions such as read local and global consistency are satisfied. However, the doomed consistency safety condition may be violated. The reason is that the tm_{imp} detects a conflict early and then it responds with random values, for simplicity we represent the random value by 0, and continues to execute after it has been aborted (resolve the conflict at commit time) which is called orphan transaction, see Fig. 5.12.

In actual fact, this situation may violate the doomed consistency condition and cause illegal action. However, the original proposed TM system by Herlihy and Moss [6], cover this case by using explicit validate instruction which is considered as overhead and depends on the software developer. Although this case can appear in our proposed tm_{spec}, the $ValidRead$ function protect the aborted transaction to fail in such erroneous case.

To cover the previous case, we suggest an implicit validate trigger for each read. This trigger can stimulate the abort function as soon as it detects that $Status$ flag is converted to false. Although, this solution will transfer the conflict resolution mode from lazy to eager, the satisfaction of the TM safety conditions will be granted and the side effect will appear in the performance, see Fig. 5.13.

In the specification of tm_{imp} in Subsection 5.2.2, the modification will be in the formula *BusyRes()* which is responsible for letting the aborted transaction continue to execute. The formula *BusyRes()* is used after checking that $Status$ is false in each transactional operation invocation state. Also, it keeps $Status$ stable, drops all x_{abort} entries, sets all x_{commit} entries to $normal$ and returns arbitrary data. The stability of $Status$, which is false in this case, and $Active$ lets the aborted transaction continue to execute in the next state.

The predicate *Abort(p)* is similar to *BusyRes(p)*, but *Abort(p)* sets $Status_p$ to true and resets $Active_p$. So, if we exchange each *BusyRes(p)* with *Abort(p)*, the aborted transaction will not continue to execute (see Fig. 5.13). To distinguish the tm_{imp} before and after the modification, we will call the modified model tm'_{imp}. Table 5.3 shows tm'_{imp}.

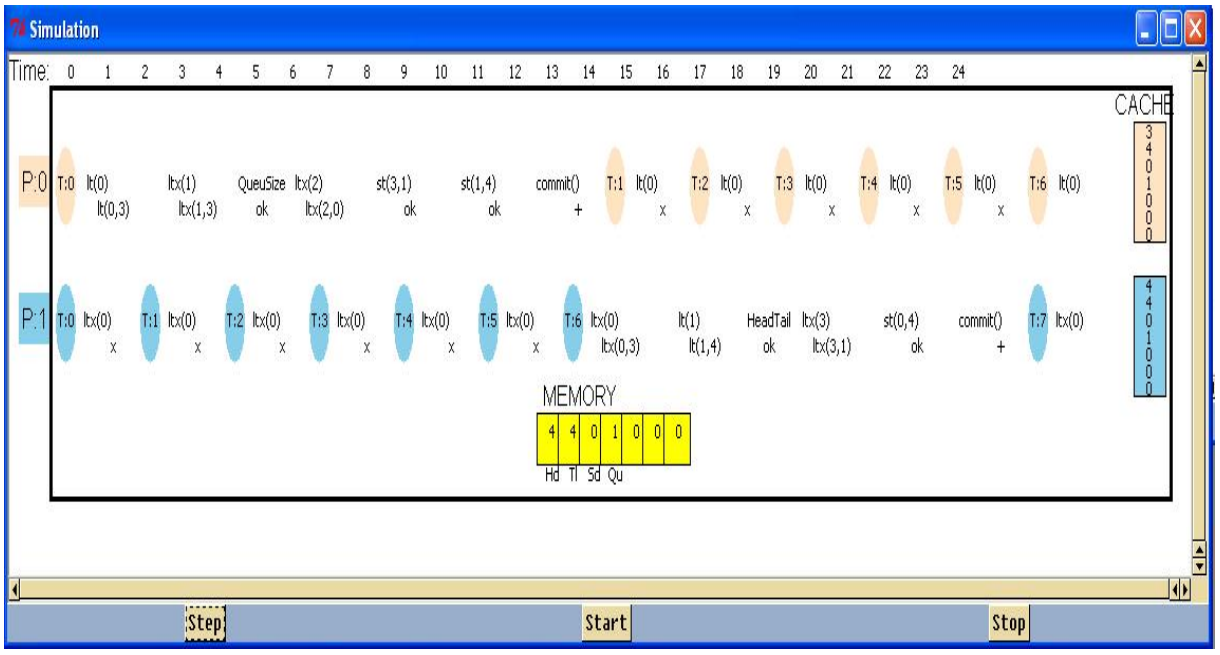

FIGURE 5.13: Queue with TM system $tm'_{imp}q$ output after modification

This model after modifying the lazy to eager. It is noticed in the beginning that P_1 re-invoked many times because the conflict with P_0. The same thing for P_0 later.

Moreover, although, a case appears in favour of running two concurrent transactions which abort each other. We add a priority condition for the producer to solve this issue. However, this solution makes another problem which is the starvation property. Where, in some cases the consumer continues aborting its self while the producer has not reached the maximum size of the queue. In the next chapter we will use a priority queue to solve this problem.

TABLE 5.3: The response actions of the modified specification tm'_{imp}

Response States				
Case*		Preconditions	Actions	Event list
$MissRes(p, ltx, x)$	k_0	$Status_p \wedge \neg ReqReady$	$\bigcirc$ Hit=$false \wedge$ $ReqData(p, op, x)$	$ltx(x)$
	k_1	$Status_p \wedge ReqReady \wedge MemRes$	$\bigcirc$ Hit=$false \wedge$ $Allocate2Loc(p) \wedge$ $Cache[ft]$=$Mem_{imp}[x] \wedge$ $Cache[sd]$=$Mem_{imp}[x] \wedge$ $UpdateTag(p, x, ft, sd)$	$ltx(x, u)$ u= Mem_{imp} [x]
	k_2	$Status_p \wedge ReqReady \wedge \neg MemRes$	$\bigcirc$ Hit=$false \wedge$ $Allocate2Loc(p) \wedge$ $Mem_{imp}[x]$=$Bus \wedge$ $Cache[ft]$=$Bus \wedge$ $Cache[sd]$=$Bus \wedge$ $UpdateTag(p, x, ft, sd)$	$ltx(x, u)$ u=Bus
	k_3	$\neg Status_p$	$\bigcirc$ Hit=$true$ $\wedge$ **AbortRes(p)**	$\otimes$
$HitRes(p, ltx, x, \perp)$	k_4	$Status_p \wedge$ $Tag_p[i][c] = shared \wedge$ $\neg ReqReady$	$\bigcirc$ $Hit = false \wedge$ $ReqInvOther(p, op, add)$	$ltx(x)$
	k_5	$\neg(Tag_p[i][c] = shared)$ $\vee (Tag_p[i][c] = shared$ $\wedge ReqReady$ $\wedge Status_p)$	$\bigcirc$ $Hit = true \wedge$ $u = Cache[ft] \wedge$ $UpdateTag(p, x, ft, sd)$	$ltx(x, u)$
	k_6	$Tag_p[i][c] = shared \wedge$ $ReqReady \wedge$ $\neg Status_p$	$\bigcirc$ $Hit = true \wedge$ **AbortRes(p)**	$\otimes$
$MissRes(p, lt, x)$	k_7	The same as one of k_0-k_3	The same as one of k_0- k_3	$lt(x)$
$HitRes(p, lt, x, \perp)$	k_8	no further condition	The same actions as k_5	$lt(x, u)$
$HitRes(p, st, x, u)$	k_9	for simplicity we assume that each st has ltx before, so there is no miss	Cache[ft]=u$\wedge$ $UpdateTag(p, x, ft, sd)$	ok

* For better readability the conditions for each state's actions is divided into case and preconditions.

* This table is variant of Table 5.2. Differences are shown in **bold**.

5.4 Verification Using Refinement Mapping

Now we have three TM specifications: the correctness verified tm_{spec}, the one proposed by Herlihy and Moss [6] which we called tm_{imp} and the modified version tm'_{imp}. The third one tm'_{imp} differs from tm_{imp} only in a state that may violate the doomed consistency safety condition. We prove in this section the correctness of tm'_{imp} and mention the state that invalidates the correctness of tm_{imp}. The abstract mapping method of Abadi and Lamport [63] is used to prove that tm'_{imp} satisfies specification of tm_{spec}. They use a refinement mapping technique that maps states of two systems and that satisfies certain properties.

Theorem 5.1. $tm'_{imp} \sqsubseteq_F tm_{spec}$

Before starting the proof, we will remind the readers about the following briefly description of the refinement mapping rules which are discussed in Subsection 3.4.3: Let C and A be two systems that denote respectively $concrete$ and $abstract$ systems. According to the mapping technique, a specification of system C implements a specification of system A denoted $C \sqsubseteq_F A$, iff there exists a mapping function F between C and A states, such that the following all hold:

- R1: The external visible state components are preserved by F.
- R2: Every initial C state has an $F-$mapped initial A state.
- R3: Every C transition can be emulated by an A transition.
- R4: F maps behaviours allowed by C into behaviours that satisfy A's supplementary property.

Proof: [Theorem 5.1]

In this section we sketch a proof, using the previous refinement mapping rules, showing that $tm'_{imp} \sqsubseteq_F tm_{spec}$. In fact, tm'_{imp} contains more state components and uses different data representations and components than tm_{spec}. For example, only tm'_{imp} has a cache. We construct a function F to facilitate applying the mapping rule R_1 of the externally observed components of tm'_{imp} $\{Mem_{imp}, Active, Status, E_{imp}\}$ and tm_{spec} $\{Mem, P, T, E\}$: where F maps states of tm'_{imp} to states of tm_{spec} such that for any state s of tm'_{imp} the following hold:

- Since there is a difference in the memory and data consistency behaviour in the two specifications, we can't directly map the value of location x in Mem to the same location in Mem_{imp}. The permanent value of any Mem_{imp}'s location doesn't move from cache to memory until it has been requested by another cache or its cache location has been overwritten. This definition facilitates mapping each value of the Mem to Mem_{imp}. (note: $0 \leq p < |Processes|$, $0 \leq i < |Cache|$, a=address, c=coherency and t=transactional):

  $\forall$ memory address $x \in < |Locations|$,

$$F(s).Mem[x] = \begin{cases} Cache_p[i] & if \quad (Tag_p[i][a] = x \\ & \quad \wedge Tag_p[i][c] = modified \\ & \quad \wedge Tag_p[i][t] = normal) \\ Mem_{imp}[x] & otherwise \end{cases} \tag{5.5}$$

- tm'_{imp} uses a flag which is called $Active$ for each process to indicate whether that process is in progress $true$ or not $false$. This task is accomplished in tm_{spec} by using a state variable called P for each process with state values $free$ and $busy$. The mapping between the two is: $F(s).P_p = Active_p$

$$P_p = \begin{cases} false & if \quad P_p = free \\ true & otherwise \end{cases} \tag{5.6}$$

- In addition, tm'_{imp} uses another flag which is called $Status$ for each transaction to indicate whether that transaction is active $true$ or aborted $false$. For the same task, tm_{spec} uses a state variable called T for each transaction with states values {*idle, active, doomed, finished* }. The mapping between the two is as follows: $F(s).T_p = Status_p$

$$T_p = \begin{cases} false & if \quad T_p = doomed \\ true & otherwise \end{cases} \tag{5.7}$$

- In case of a list of events or output of each operation E, we consider the following mapping for tm_{spec} and tm'_{imp} : W=st, $tryCom = commit$ and $tryAbort = abort$. In addition, the ltx and lt read operations in tm'_{imp} are equivalent to R in tm_{spec}. However, there is one difference appearing in the ltx case which is stronger in the conflict detection. This difference is solved by using an *eagerly-strong* conflict detection type ε_{e^s}, which is discussed in Subsection 4.3.2, with R instruction in this case to emulate ltx. After applying this definition, we can use $F(s).E_{imp}p = E_p$.

$$R \equiv \begin{cases} ltx & if \quad \varepsilon = \varepsilon_{e^s} \\ lt & otherwise \end{cases} \tag{5.8}$$

R_2 stipulates equality of the initial state between tm'_{imp} and tm_{spec}. We notice the applicable of this rule from their specifications and the following Table 5.4:

TABLE 5.4: The initial values of tm'_{imp} and tm_{spec}

	State variables	Initial value
tm_{spec}	Mem	$\perp$
	P_p	$free$ according to Definition 5.6 = $false$
	T_p	$idle$ according to Definition 5.6 = $true$
	E_p	no_{ev}
tm'_{imp}	Mem_{imp}	$\perp$
	$Active_p$	$false$
	$Status_p$	$true$
	$E_{imp}p$	no_{ev}

To validate R_3, we will consider each of the tm'_{imp} transition states that are described in Table 5.2. (where $\widehat{k}_i \equiv \bigvee_{j=0}^{1} m_j \wedge k_i$):

- $\widehat{k}_0$: This transition is the first one which can possibly respond to a missed read-exclusive operation ltx. The preconditions are that the requested data that has address x does not exist in the local cache, the transaction status is still active and no data-request has been issued. The action issues a data request for address x from either other caches or from the main memory. *ReqData(p)* detects the tag status of the x in the other caches and resets the status of the local transaction flag $Status_p$ if it finds the x in another cache q with transactional status equal x_{abort} which means that x is being used (for writing or

reading) by another concurrent transaction. Otherwise, the ***ReqData(p)*** keeps the status of the local transaction flag $Status_p$ stable. Actually, the task of ***ReqData(p)*** is similar to the ***ConflictDetRes(p,ε)*** task of the invocation states in the tm_{spec} with conflict detection type ε equals ε_{e^s} ***(strong-eager-conflict)*** and resolution type (ε_r) equals (ε_{ro}) ***eagerly-own-arbitration***. In addition, the preconditions of the invocation states in tm_{spec} can be emulated by the preconditions of $\widehat{k}_0$. So, we can match $(k_0 \wedge m_0)$ to s_0 and $(k_0 \wedge m_1)$ to s_1.

- $\widehat{k}_1$: This transition receives the requested data for address x from the main memory, places in local cache, updates its tag to maintain coherency and adds $ltx(x, u)$ to the output list (where $u = Mem_{imp}[x]$). The preconditions are no local data (no write before by the same transaction) and that the local transaction flag $Status$ is true. This can be emulated by the tm_{spec} transition $\widehat{s}_2$, whose output list is equal to $R(x, u = Mem[x])$ and its preconditions are that there is no local write and $\neg(T_p = doomed)$.

- $\widehat{k}_2$: The actions and preconditions of $\widehat{k}_2$ are similar to $\widehat{k}_1$ the only difference is that the requested data for address x comes from some other cache and not from the main memory. According to Definition 5.5, this transition can be emulated by the tm_{spec}'s transition $\widehat{s}_2$.

- $\widehat{k}_3$: This aborts the transaction that has a local transaction flag $Status$ equal to *false* and outputs $\otimes$. The flag is reset by $\widehat{k}_0$ after detecting a conflict with another concurrent transaction. This transition can be matched with the tm_{spec} transition $\widehat{s}_1$ that aborts the doomed transaction and outputs $\otimes$. In tm_{spec}, one of its invocation states can change the

status of its active transaction to doomed after detecting a conflict with another concurrent transaction.

In this case, the tm_{imp} cannot be mapped to the tm_{spec}, since there is no state in tm_{spec} that allows the aborted transaction to continue to execute without checking the violation of doomed consistency. If we assume that the following situation happens in the tm_{imp}: the local transaction flag $Status_p$ equal to false, there is a doomed inconsistency state and there is no $validate$ instruction following this state. Then, the tm_{imp} will issue the $BusyRes(p)$ that allows to continue executing the aborted transaction and the divided by zero or illegal action may happen.

- $\widehat{k}_4$: The transition issues a request to own x exclusively by invalidating other copies of x using ***ReqInvOther()***. ***ReqInvOther()*** detects the tag status of the x in other caches and resets the status of the local transaction flag $Status_p$ if it finds the x in another cache q with transactional status equal to x_{abort}. This task is similar to ***ConflictDetRes(p)*** in tm_{spec} as well as ***ReqData()*** in $\widehat{k}_0$ without retrieving data. So, it can be handled like $\widehat{k}_0$.

- $\widehat{k}_5$: This reads the value u of x from the local cache and outputs $ltx(x, u)$. In simple terms, its preconditions are that x's coherency status is equal to *exclusive* or *modified* which means that x is preparing for write or it is already updated. Transition $\widehat{s}_0$ in the tm_{spec} reads the value u of x from the previous write instruction by the same transaction and outputs $R(x, u)$. So, we can say that $\widehat{s}_0$ allows $\widehat{k}_5$.

- $\widehat{k}_6$: This aborts the transaction that has a conflict with other concurrent transactions. The conflict is detected by $\widehat{k}_4$. Similar as $\widehat{k}_3$, this transition can be emulated by $\widehat{s}_1$. This case cannot be mapped to the tm_{imp} for the same reasons mentioned in $\widehat{k}_3$.

- $\widehat{k}_7$ and $\widehat{k}_8$: These transitions deal with the lt operation. According to Definition 5.8 these transitions can be handled like $\widehat{k}_0 - \widehat{k}_3$ and $\widehat{k}_5$, respectively.

- $\widehat{k}_9$: This deals with st operation and outputs ok. In this transition there is an assumption that there is a successful ltx before each st which means that there is no conflict with each st. Transition $\widehat{s}_3$ in the tm_{spec} deals with the $write$ operation and outputs ok when its transaction isn't doomed. So, the two transitions can be matched.

- m_2: This aborts any invocation that does not match the preconditions of $(m_0 \wedge m_1)$. It matches s_3.

- m_3: This aborts a transaction if the value of its $Status$ flag is false or the it issues an abort operation . It emulates $\widehat{s}_5$ if $Status$ equals false and $\widehat{s}_7$ there is an abort request.

- m_4: This commits a transaction with regarding its $Status$ flag. It matches $\widehat{s}_4$.

- m_5: it returns the status of $Status$ without action.

To verify rule R_4, we can obtain the validity of this rule from the complete specification of tm'_{imp}. $TraInvResEnd()$ follows each sequence of transactional operations invocation to commit or abort any active transaction. This behaviour can be mapped to $TranInvEnd()$ followed by $CommitTran()$ or $AbortTran()$ in tm_{spec} to commit or abort any *active* or *doomed* transaction.

5.5 Summary

The main feature of the proposed formal TM model is the ability to use it as a basis for proving the correctness of a variety of TM systems. In this chapter a case study, a specification for the original well-known HTM system proposed by Herlihy and Moss is given using ITL. To validate the correctness of this specification, we show an executable version and use it to execute one of the most highly studied concurrent data structures, the lock-free FIFO queue, using AnaTempura. However, we found a violation for the doomed consistency safety condition in this specification, and discussed for a workaround which aborts each doomed transaction. The integration of this workaround with the specification of the selected TM system is correctly proved using a refinement mapping technique that maps all its possible states with the states of the provable abstract TM model.

Chapter 6

Specification of Chip Dual Processor

6.1 Introduction

The main reason for proposing TM techniques is to solve memory synchronization problems in a shared memory environment such as a CMP. Therefore the integration of TM techniques with such environments is essential in TM design and evaluation [1].

Moreover and as mentioned previously, we would like to transform a high-level TM system specification in ITL such as tm'_{imp} in Chapter 5 to lower-level or to a real hardware design expressed in a Hardware Description Language (HDL). However, the HDL for a TM system as a single unit, independent of a shared memory environment, is not considered sufficiently worthwhile to be regarded as a final hardware product. So, we extend our formal framework to support a specification of a shared memory system.

As shown in the framework's Fig. 6.1, this chapter focuses on the steps inside the circle. We describe a fully executable specification of a Chip-Dual-Processor (CDP) using AnaTempura and integrate within the executable specification of the verified transactional memory model tm'_{imp}, which is described in the previous chapter as a case study. We then transform the dual processors with the TM system from a high level description into a hardware description language, using proposed refinement and restriction rules.

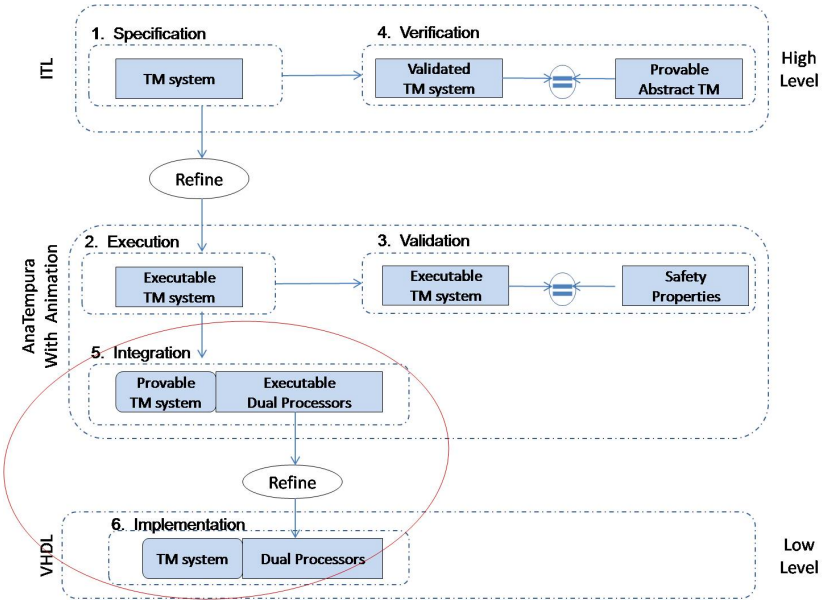

FIGURE 6.1: Proposed framework.

We start the specification of CDP with a simple single-processor design and then incrementally add features and components until we reach our final model. A structure overview of the CDP is presented and system components are listed first before we get deeper into the design details. Formulae and schematics will be used as necessary to describe the functionality of major system modules in the course of our discussions.

6.2 CDP Architecture Overview

The proposed CDP architecture model consists of 2 processors connected by a snoopy bus. Each processor has a full associative cache (transactional memory), in addition to a regular direct mapped data cache as shown in Fig. 6.2. This model also supports normal load (lw) and store (sw) instructions for non-transactional operations, plus transactional memory instructions such as load transactional for exclusive (ltx), load transactional (lt) and store transactional (st) [1].

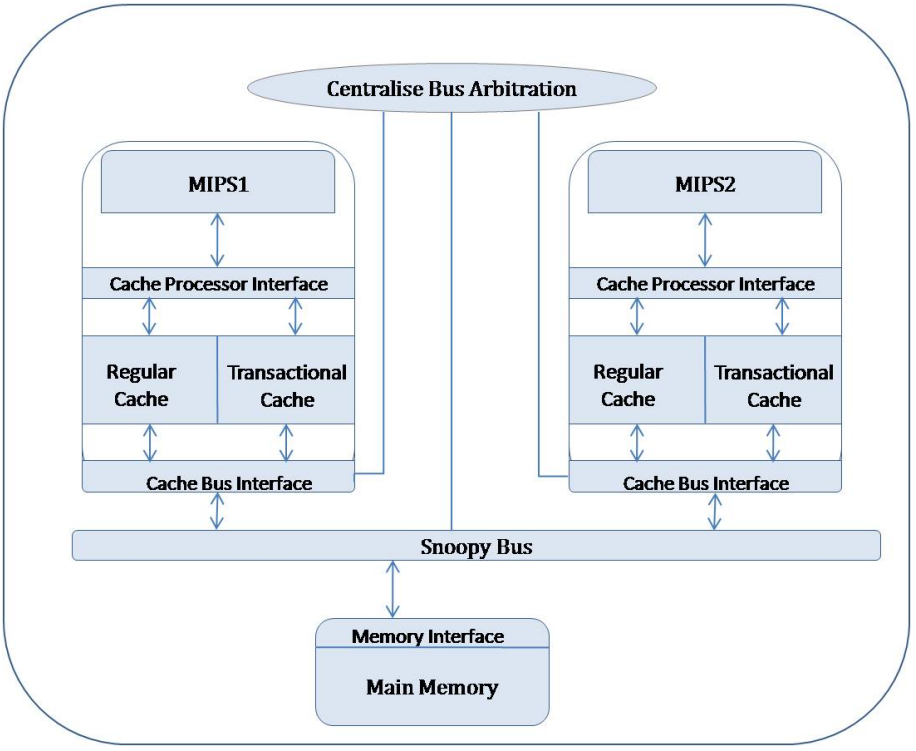

FIGURE 6.2: Chip dual processor.

Each processor contacts the data cache through the processor-side of the cache interface, and each processor contacts the other processor and the memory through bus-side of the cache

interface and snoopy bus. The arbitration of the snoopy bus and memory driver on the chip are controlled by the *Centralised Bus Arbitration Mechanisms (CBAM)*, which will be explained in the snoopy bus section. The specification of the CDP is a large ITL formula. The general structure of the formula is as follows:

$$
\begin{aligned}
&\textit{CDP()} \ \widehat{=} \\
&\quad \exists\, MemData, DataBus, AddBus, CmdBus, Queue, Grant, \\
&\quad CacheData, CacheTM, TagPro, TagBus, Pc, InsMem, RegFile, \\
&\quad Active, Status :\{ \\
&\quad init() \wedge \\
&\quad repeat \Big(\\
&\qquad skip \wedge \\
&\qquad \big(\big(\textit{Processor}_1(Pc_1) \wedge (Pc_1 \leq \textit{prog}_{size1}) \big) \vee \\
&\qquad\quad \big(\textit{StablePro}_1() \wedge (Pc_1 > \textit{prog}_{size1}) \big) \big) \wedge \\
&\qquad \big(\big(\textit{Processor}_2(Pc_2) \wedge (Pc_2 \leq \textit{prog}_{size2}) \big) \vee \\
&\qquad\quad \big(\textit{StablePro}_2() \wedge (Pc_2 > \textit{prog}_{size2}) \big) \big) \wedge \\
&\qquad \textit{CacheBusInterface}_1() \wedge \\
&\qquad \textit{CacheBusInterface}_2() \wedge \\
&\qquad \textit{CentraliseBusArbitration}() \wedge \\
&\qquad \textit{MemoryInterface}() \\
&\qquad \Big) \text{ until } \big((Pc_1 > \textit{prog}_{size1}) \wedge (Pc_2 > \textit{prog}_{size2}) \big) \\
&\quad \}
\end{aligned}
$$

The main state variables of CDP are the memory data block *MemData*, snoopy bus components

DataBus, AddBus, CmdBus, Queue,Grant and processor and data cache components *Cache-Data, TagPro, TagBus, Pc, InsMem, RegFile,* $Active$, $Status$. Each component is defined as a list of one or more dimensions, so we can specify the size of the component and the number of the processor that it belongs to. The individual tasks for each of them will be shown later in this chapter.

The *init()* formula initialises the values of the variables. The repeat statement repeats executing, in parallel, the formulae (between () of repeat) until the program counter *Pc* for both of the two processors equal to their size of a program $prog_{size}$. If the *Pc* of just one processor *Processor()* reaches the end of the program, the other one can continue to execute until it reaches the end of its program as well.

The *CentraliseBusArbitration()* formula coordinates the usage of the bus by the processors and manages the connection between the *CacheBusInterface()* of the two processors with each together and with the *MemoryInterface()* of the memory block. We separate the *CacheBusInterface()* part of *Processor()* formula to keep its data cache block active when its processor reaches to the end of the program and processor's components are stable. The $skip$ formula describes that we use an interval of two states, namely the stable states of the CDP before and after each clock cycle. The overall structure of each processor's main formulae and the control flow and representation of the data are similar to the model in [58]. More details for these components and formulae are given in the following sections.

6.3 Microprocessor

In our model, the processor's architecture is based on the Million-Instructions-Per-Second (MIPS) architecture which is a simpler 32-bit version of the real MIPS R2000 Microprocessor [69, 73]. This has a single-cycle data path design, but on cache misses, the running program counter freezes until the missing data is retrieved from the main memory. The reason for not having a pipeline is obviously to simplify the internal processor design. Such a design does not affect the behaviour of the cache coherency and memory synchronisation, as memory access patterns and times are highly similar to those of pipelined designs.

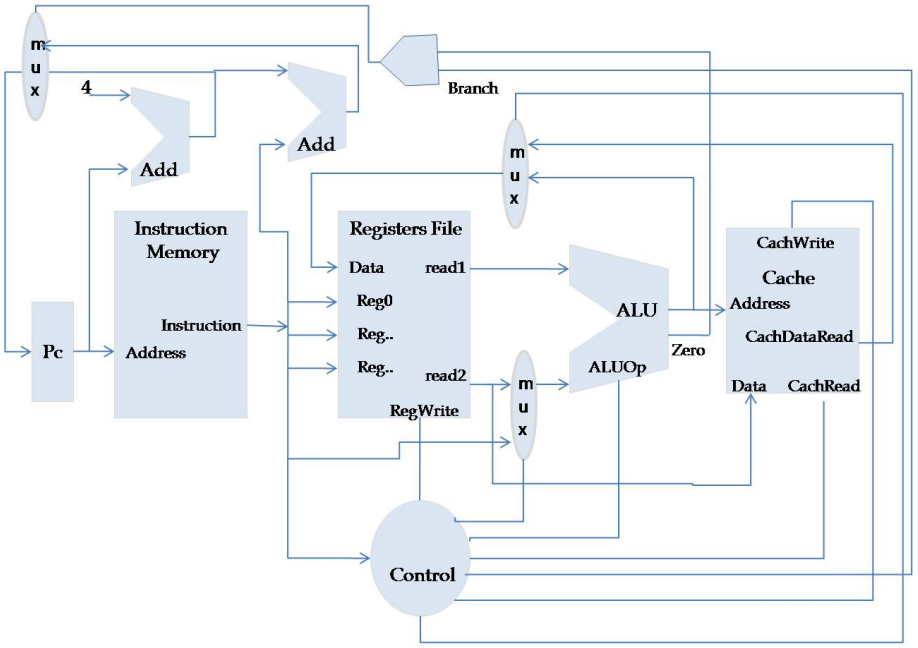

FIGURE 6.3: Abstract view of MIPS processor architecture.

As we show in Fig. 6.3, the data path of this processor consists of five major functional entities which are: instruction fetch, instruction decode, execute unit, control unit and cache-processor

interface for regular data-cache and transactional cache unit. To execute an instruction on MIPS, we must start by fetching it from the instruction memory using the Program Counter *Pc*. Then, according to an MIPS instruction classification in Fig. 6.5, it is decoded by specifying the register's number in the register file $RegFile$ and then fetches register operands. Once the operands have been fetched, three actions can be done according to the instruction class. Firstly, if the instruction is load or store then the operands will be used to calculate a memory address. Secondly, if the instruction belongs to the arithmetic-logic class then the operands will be used to compute an arithmetic result. Finally, they can be used to compare each other for a branch instruction. The execution unit output is written back into the register if the instruction is arithmetic-logical, used as an address if the instruction is (load or store) or used to determine the next address of a branch operation. The general specification of the five entities and the main state variables is represented in the formula $Processor()$ as follows:

$$
\begin{aligned}
&\textit{Processor()} \ \widehat{=} \\
&\qquad \exists\, ReadData1, ReadData2, RegWr, RegDst, MemtoReg, \\
&\qquad ALUSrc, ALUOp, Zero, ALU\_result, CachRead, \\
&\qquad CachWrite, CachDataRead, Inst :\{ \\
&\qquad InstFetch() \wedge \\
&\qquad InstDecode() \wedge \\
&\qquad ControlUnit() \wedge \\
&\qquad ExecuteUnit() \wedge \\
&\qquad CacheProIntererface() \\
&\qquad \}
\end{aligned}
$$

The state variables of the formula *Processor()* represent the major connections between the five entities or formulae. The description of the first four formulae is shown below with some parts of their executable specification and the formula *CacheProIntererface()* will be described in the cache interface section (see Section 6.4).

Instruction Fetch

The formula $InstFetch()$ uses the instruction memory *InsMem*, the program counter *Pc*, and subformula *adder* to specify the next value of *Pc*. The instruction memory is used to store the predetermined instructions that are to be executed by the processor. The high-level representation of instructions should be converted into their MIPS binary equivalents and stored in the instruction memory sequentially starting at the address zero.

The subformula *adder* controls the flow of the program execution in the next state. It works in much the same way as purpose the *Hit* flag of the specification tm_{imp} in the previous chapter. It freezes the *Pc* when the requested address is missed in the local cache and a request for a bus is issued, so in the next state the same instruction will be invoked again. The *adder* sets the *Pc* to the branch address *BranchAdd* that is resident in the branch instruction if there is a branch instruction and the compare result is zero. Otherwise, it increments the *Pc* by four because each instruction has a four byte long.

$$\begin{aligned} \mathit{adder} \;\widehat{=}\quad & (\mathit{Opcode} = \mathit{branch} \wedge \mathit{Zero} = \mathit{false} \wedge \bigcirc \mathit{Pc} = \mathit{BranchAdd}) \\ & \vee (\neg(\mathit{Opcode} = \mathit{branch}) \wedge \mathit{CachBusReq} \wedge \bigcirc \mathit{Pc} = \mathit{Pc}) \\ & \vee (\neg(\mathit{Opcode} = \mathit{branch}) \wedge \neg \mathit{CachBusReq} \wedge \bigcirc \mathit{Pc} = \mathit{Pc} + 4) \end{aligned}$$

Here, the variable *Opcode* is the last 6 bits of each instruction to specify the instruction's type. For the transactional memory instructions, we use a branch instruction after each commit operation. In addition, we reserve register number zero for the instruction commit. If the transaction commit succeeds, register number zero is set else it is reset. The branch instruction compares one to the value of register number zero. If the result of the comparison is zero, the next instruction will be invoked, otherwise the same transaction will be invoked again.

Control Unit

The formula $\mathit{ControlUnit}()$ in Fig. 6.4 examines the instruction's *Opcode* bits and generates eight control signals used by other stages of the processor. The executable specification of the control unit is shown in Fig. 6.4. The binary flag *R_format* represents the class of arithmetic and logic instructions such as *add, sub, or* and *and*. Moreover, Fig. 6.4 shows the integration of the load and store transactional operation *LT, LTX, ST* with non-transactional instructions in this entity. This specification will be used again later in this chapter.

$$
\begin{aligned}
&\textit{ControlUnit()}\\
&\quad \widehat{=} \quad \exists \;\; R\_format, LT, LTX, ST, LW, SW, Bne :\{\\
&\qquad \textit{if Opcode=”000000” then R\_format=t else (R\_format=f)} \wedge\\
&\qquad \textit{if Opcode=”100011” then LW=t else (LW=f)} \;\wedge\\
&\qquad \textit{if Opcode=”101011” then SW=t else (SW=f)} \;\wedge\\
&\qquad \textit{if Opcode=”000001” then LT=t else (LT=f)} \;\wedge\\
&\qquad \textit{if Opcode=”000010” then LTX=t else (LTX=f)} \;\wedge\\
&\qquad \textit{if Opcode=”000011” then ST=t else (ST=f)} \;\wedge\\
&\qquad \textit{if Opcode=”000101” then Bne=t else (Bne=f)} \;\wedge\\
&\qquad \textit{RegDst=R\_format} \;\wedge\\
&\qquad \textit{ALUSrc=(LT} \vee \textit{LTX} \vee \textit{ST} \vee \textit{LW} \vee \textit{SW)} \;\wedge\\
&\qquad \textit{MemtoReg=(LT} \vee \textit{LTX} \vee \textit{LW)} \;\wedge\\
&\qquad \textit{RegWr=(LT} \vee \textit{LTX} \vee \textit{R\_format} \vee \textit{LW)} \;\wedge\\
&\qquad \textit{CachRead= LW} \;\wedge\\
&\qquad \textit{CachWrite= SW} \wedge\\
&\qquad \textit{ALUOp[opLen-1]= R\_format}\wedge\\
&\qquad \textit{ALUOp[opLen-0]= Bne}\\
&\quad \}
\end{aligned}
$$

FIGURE 6.4: Specification of Control Unit.
(t and f are abbreviations of true and false)

Instruction Decode

The formula *InstDecode()* decodes the fetched instruction from the instruction memory. We will illustrate the MIPS instruction formats to help understanding the decode stage. An MIPS instruction is a list of 32 boolean positions and is classified into three types. As shown in Fig. 6.5, the MIPS instruction classes are:

Bit Positions	31:26	25:21	20:16	15:11	10:6	5:0
R_type	Opcode	rs	rt	rd	shamt	func
Load-Store	Opcode	rs	rt	address		
Branch	Opcode	rs	rt	address		

FIGURE 6.5: MIPS 32-bit instruction formats.

- R_type: This represents the arithmetic and logic instruction classes such as *add, sub, or* and *and*. It uses two registers which are specified by the rs and rt fields at positions 25:21 and 20:16. In addition, it uses a destination register to write the result at position 15:11 (rd). Position 5:0 specifies which instructions are in an R_type.

- Load-Store: This load instruction uses a destination register to write data from the main memory at position 20:16 (rt) while the store instruction uses a register which is read at position 20:16 (rt). Both of them use a base address register at position 25:21 (rs) and a 16bit offset at position 15:00.

- Branch: This compares two values then jumps to the offset. It uses two registers, which are specified by the rs and rt fields, at positions 25:21 and 20:16. Also, it uses a 16bit offset at position 15:00.

As shown in Fig. 6.6, The formula *InstDecode()* uses a register file *RegFile* with length 32 and width 32-bit *(MIPS contains thirty two 32-bit registers)* to temporarily store the data that comes from Arithmetic Logic Unit (ALU) or cache as follows: Firstly, it decodes the fetched instruction which is called *Inst*, which is a list of 32 boolean positions, by specifying the two registers to be read using *ReadAddrReg1* and *ReadAddrReg2* at positions 25:21 and 20:16 of *Inst* for R_type and store instructions, and the two destination registers by similar use of *WriteAddrReg1* and *WriteAddrReg2*. For a load it is in *Inst*'s boolean positions 20:16, while for an R_type instruction it is in *Inst*'s boolean positions 15:11. To specify a sublist in Tempura, we only need to specify the number of the first and the last location of the sublist. We use a static variable

length, which equals 32, and subtract it from the bit position number, to match the hardware specification.

InstDecode() $\widehat{=}$

$\exists\ WrData, AddrWr, ReadAddrReg1, ReadAddrReg2,$
$WriteAddrReg1, WriteAddrReg2, InstImmedValue :\{$
ReadAddrReg1 =Inst[(length-1)-25 to length-21] $\wedge$
ReadAddrReg2 = Inst[(length-1)-20 to length-16] $\wedge$
WriteAddrReg1 = Inst[(length-1)-15 to length-11] $\wedge$
WriteAddrReg2 = Inst[(length-1)-20 to length-16] $\wedge$
InstImmedValue= Inst[(length-1)-15 to length-0] $\wedge$
ReadData1=RegFile[Conv_Integer(ReadAddrReg1)] $\wedge$
ReadData2=RegFile[Conv_Integer(ReadAddrReg2)] $\wedge$
if MemtoReg then (WrData= CachDataRead)
else (WrData= ALU_result) $\wedge$
if RegDst then (AddrWr=WriteAddrReg1)
else (AddrWr=WriteAddrReg2) $\wedge$
if Reset then forall i<rows:
$\bigcirc$ *RegFile[i]=Conv_Std_Logic_Vector(i)*
else if RegWr then ($\bigcirc$ *RegFile[Conv_Integer(AddrWr)]= WrData)*
else StableOther(-1)
}

FIGURE 6.6: Specification of Instruction Decode Unit.
(t and f are abbreviations of true and false)

Secondly, the formula *InstDecode()* fetches two operands from *RegFile* and puts them in the output state variables *ReadData1* and *ReadData2* by using *Conv_Integer()* subformula that accepts list of booleans and returns integer number. Finally, the formula *InstDecode()* uses two subformulae to write back into *RegFile*. The first one checks *RegDes* to select which field of the instruction is used to indicate the register number to be written. The second checks *MemtoReg* to select which data (R-type from execution unit *ALU_result*, Load from cache *CachDataRead*) will be written in the register file. The three state variables *RegDes, MemtoReg and RegWr* are

set by *ControlUnit()*. The function *Conv_Std_Logic_Vector(i)* converts the integer i to a list of boolean(bits), whereas the function *Conv_Integer(AddrWr)* converts the list of boolean *AddWr* to an integer. These functions help to access lists using an index.

Execute Unit

The formula *ExecuteUnit()* describes the data *ALU* that handles all arithmetic and logical operations and also contains a branch address $adder$ used for *Pc*'s relative branch instruction. It has a subformula that selects the data (a register file or a sign-extended unit) for the *ALU* input, see Appendix B.

6.4 Cache Structure and Specification

As shown in Fig. 6.7, the cache diagram consists of cache data block, dual tags, comparators, a bus-side controller, a processor-side controller and a bus interface. In addition to the transactional cache, a direct mapped data cache type is used as a regular cache. The direct mapped cache is the simplest form of cache and the easiest to check for a hit. Since there is only one possible place that any memory location can be cached, there is nothing to search. It works for non-transactional load and store instructions. Both caches are accessed directly by the processor. The data block is placed in one of the two caches, not both. The write policy used in this design is write-back, which copies a block back to memory in two cases: first when it is replaced and second when it is invalidated by another processor, as will be explained in the next

section. Using a write-back policy to reduce bus traffic and thereby allowing more processors in a single bus.

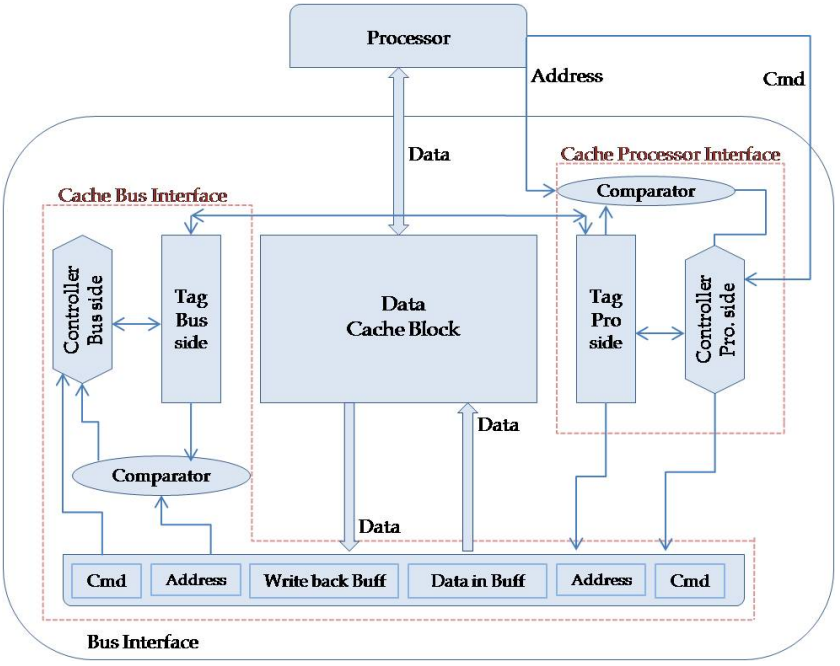

FIGURE 6.7: Cache interfaces structure.

The purpose of using dual tags with two controllers is to allow the two controllers to access simultaneously the array of the tags. This will increase the processor performance and effective bus bandwidth; the processor will only be locked out from accessing the cache if the bus side controller performs a tag check while being hit. Doing so, the processor needs to update both copies in the two tags. Each tag consists of two parts, the address and the coherency status. The tag coherency status part is a sublist of 2 booleans wide to represent one of four *MESI* states *(modify, exclusive, shared and invalid)*. The bus interface works as a layer between the processor and snoopy bus. It receives two sets of inputs: the first one is from the processor when it issues memory requests. The second input is from the bus; in every bus transaction

the bus interface captures the address and command from the bus. Then, it sends the address and the command to the bus side controller in order to use it according to the cache coherence protocol.

Cache Specification

The specification of the cache structure uses three main state variables *(CacheData, TagPro, TagBus)* and two main interface formulae which are *CacheProInterface()* to connect the cache with the processor and *CacheBusInterface()* to connect the cache with the snoopy bus. Each interface monitors external events from its side. In either case, when an operation occurs, the interface uses the controller to access the cache tag, and then gets the result from the comparator.

The *CacheProInterface()* deals with the processor's loads and stores transactional and non-transactional instructions. It receives the command and the address from the processor and triggers one of the two cache's processor side controllers according to the type of commend as follows:

$$\textit{CacheProInterface()} \;\widehat{=}\; (ProCont_{reg}() \wedge (sw \vee lw)) \vee (ProCont_{tm}() \wedge (st \vee lt \vee ltx)))$$

The processor-side controller of regular cache $ProCont_{reg}()$ deals with non-transactional load *lw* and store *sw* instructions. It performs two checks. First, it compares a portion of the address with the tag processor-side *TagPro*, and uses the remaining address as an index for the tag. Second, it checks the tag coherency status *(modify tt, exclusive ft, shared tf, and invalid ff)* for

the same index. Then the appropriate operations are performed on the data and the tag status according to the command.

For example, when a store *sw* instruction hits the data, the results of checking the tag status will be one of three cases. The first is *modify*, where the controller asserts a *write* variable to update the data cache block. The second is *exclusive*, where it will convert the tag's block status of both *TagPro* and *TagBus* to *modify* and then update the data block. The third is *shared*, where it will assert the bus request flag *CacheBusReq* and send the address and commend to the bus interface. In the next state, the controller checks the *ReqReady* flag, that is set by the bus interface when it gets the snoopy bus, if it is true then the controller converts the tag's block status of both *TagPro, TagBus* from *shared* to *modify* and then updates the data block.

The processor-side controller for the transactional cache $ProCont_{tm}()$ deals with the following transactional instructions: load *lt, ltx*, store *st*, commit a transaction *commit*, abort a transaction *abort* and validate the status of a transaction *validate*. It handles the hit and miss data cases as *FindData()* formula in tm_{imp}. The difference here is at cases of requesting miss data and requesting to invalidate other copies. In these cases, the $ProCont_{tm}()$ sets the *CacheBusReq* and sends the address and the command to the bus interface.

The *CacheBusInterface()* deals with commands and addresses that come from the snoopy bus using the bus-side controller of both transactional formula $BusCont_{tm}$ and regular cache formula $BusCont_{reg}$. In addition, it handles request commands by a cache to use the snoopy bus

via the *BusInterface()* subformula.

$$\textit{CacheBusInterface()} \;\widehat{=}\; BusCont_{reg}() \wedge BusCont_{tm}() \wedge BusInterface() \tag{6.1}$$

On every bus state, the bus-side controller $BusCont_{reg}()$ receives the command and the address from bus lines and makes two comparisons, with tag address and tag coherency statuses of the tag bus-side *TagBus*. If the check fails (and the coherency status is invalid), no action needs to be taken. If the check hits, the bus-side performs a sequence of operations according to *MESI* protocol.

As shown in Fig. 6.8, the bus-side controller for transactional cache $BusCont_{tm}()$ has two different responses to deal with bus commands *CmdBus*, if the following are satisfied: the data on the bus is not issued by the same cache (*Grant* flag is false), it finds a tag's block address part equal to *AddBus*, coherency status of the same block is not *invalid* and its transactional status is not *empty*. The two possible responses are: firstly, setting the busy out state variable *BusyOut*, if the tag's block transactional status is not *normal*, which means that this cache block is used by a transaction. Secondly, issuing a request to write back the requested data by setting *BusReqWB* flag and transferring the data block from the cache to write back buffer *BuffWB*, if the tag's coherency status is *modified* and transactional status is *normal*. Also, it changes its tag's coherency status to *invalid*, if the *CmdBus* is read for exclusive $readFx_{tm}$ or invalid other copy $invCopy_{tm}$, otherwise it changes the tag's coherency status to *shared*.

The *BusInterface()* deals with a request by the processor-side or bus-side controller to acquire

$$
\begin{array}{l}
BusCont_{tm}(CmdBus, AddBus, p, i) \mathrel{\widehat{=}} \\
\{\textsf{if } (\neg Grant_p \wedge i < cache_{length}) \\
\quad \textsf{then if } (TagBus_p[i][a] = AddBus \\
\qquad \wedge\neg(TagBus_p[i][c] = invalid \vee TagBus_p[i][t] = empty)) \\
\qquad \textsf{then if } TagBus_p[i][t] = normal \\
\qquad\quad \textsf{then } \{\bigcirc HitBus = t \wedge \bigcirc BusyOut = f \wedge \\
\qquad\qquad \textsf{if } CmdBus = read_{tm} \\
\qquad\qquad\quad \textsf{then } UpdateTag(AddBus, p, shared) \\
\qquad\qquad\quad \textsf{else } UpdateTag(AddBus, p, invalid) \\
\qquad\qquad \wedge \\
\qquad\qquad\ \textsf{if } TagBus_p[i][t] = modify \\
\qquad\qquad\quad \textsf{then } \{\bigcirc BusReqWB_p = t \wedge \\
\qquad\qquad\qquad\qquad \bigcirc BuffWB = Cache(read, i, p)\} \\
\qquad\qquad\quad\ \textsf{else } \bigcirc BusReqWB_p = f \\
\qquad\qquad \} \\
\qquad\quad \textsf{else } \{ \\
\qquad\qquad \bigcirc HitBus = f \wedge \bigcirc BusReqWB_p = f \wedge \\
\qquad\qquad\ \textsf{if } (CmdBus = read_{tm} \wedge TagBus_p[i][t] = shared) \\
\qquad\qquad\quad \textsf{then } \bigcirc BusyOut = f \\
\qquad\qquad\quad\ \textsf{else } \bigcirc BusyOut = t \\
\qquad\qquad \} \\
\quad \textsf{else } BusCont_{tm}(CmdBus, AddBus, p, i + 1) \\
\}
\end{array}
$$

FIGURE 6.8: Specification of bus-side controller of the transactional cache.

the bus line . There are three cases for demanding to acquire the bus: Firstly, missing a data in the local cache or requesting to exclusively own a data, in this case the *CacheBusReq* flag is set by the processor-side controller. Secondly, replacing the data cache block, the *ProReqWB* flag is set. Finally, responding to the bus command to get a data cache block, the *BusReqWB* flag is set by the bus-side controller.

As shown in Fig. 6.9 the *BusInterface()* performs a sequence of steps for the first case as follows: 1) waits for bus grant, 2) puts address *AddCache* and command *CmdCache* on bus, 3) waits for

$$
\begin{array}{l}
BusInterface(p) \mathrel{\widehat{=}} \\
\{\textsf{if } (CachBusReq_p \wedge Grant_p) \\
\quad \textsf{then if } (\neg BusDataReady) \\
\qquad \textsf{then } \{\bigcirc AddBus = AddCache \wedge \\
\qquad\quad \bigcirc CmdBus = CmdCache \wedge \\
\qquad\quad \bigcirc ReqReady = f\} \\
\qquad \textsf{else } \{\bigcirc ReqReady = t \wedge \\
\qquad\quad \textsf{if } (CmdCache = readFx_{tm} \vee \\
\qquad\qquad CmdCache = read_{tm} \vee \\
\qquad\qquad CmdCache = invCopy_{tm}) \\
\qquad\qquad \textsf{then if } (\neg BusyIn) \\
\qquad\qquad\quad \textsf{then } \{\bigcirc BuffInData = BusData \wedge \bigcirc Status_p = t\} \\
\qquad\qquad\quad\quad \textsf{else } \bigcirc Status_p = f \\
\qquad\qquad \textsf{else } \bigcirc BuffInData = BusData\} \\
\quad \wedge \\
\quad \textsf{if } (ProReqWB_p \wedge Grant_p) \\
\quad \textsf{then } \{\bigcirc AddBus = AddCache \wedge \bigcirc CmdBus = write \wedge \bigcirc DataBus = BuffWB\} \\
\quad \wedge \\
\quad \textsf{if } (BusReqWB_p \wedge SendBlk) \\
\qquad \textsf{then } \bigcirc DataBus = BuffWB \\
\}
\end{array}
$$

FIGURE 6.9: Specification of the bus interface.

acknowledgment *BusDataReady*, and 4) transfers data from *BusData* to *BuffInData*. In case of transactional operations and after it receiving acknowledgment *BusDataReady*, it checks the *BusyIn* (indicates conflict detection with other concurrent transaction) and sets the *BusyIn* flag if *BusyIn* is true, otherwise it is reset.

In the second case, the *BusInterface()* 1) waits for the bus grant, 2) puts the *write* command, address and data on the bus. In the final case, It waits for acknowledgement to send the data cache block *SendBlk* and then transfers data from *BuffWB* to *BusData*.

6.5 Snoopy Bus Structure and Specification

We use an asynchronous snoopy bus system that uses a handshaking protocol for coordinating usage rather than a clock (synchronous). The advantages of choosing asynchrony are the ability to accommodate a wide number of processors and devices of differing speeds, and the cache-to-cache handshake is simple; the disadvantage is that the design requires extra hardware and signals [69].

The snoopy bus structure consists of three components: A data bus with 32-bit width, address/-command lines with four bus cycles for non-transactional operations ($read$: for shared cache line, $readFx$: read for exclusive, $invCopy$: invalid copies in other caches, and $write$). We also add four cycles bus for transactional operations: ($read_{tm}$, $readFx_{tm}$, $invCopy_{tm}$ and $BusyIn$: for refusing a transactional request when responding by busy signals $BusyOut$), and the centralised bus arbitration, which contains a hardware queue and a group of signals. This group of signals works according to the following sequence: when the processors need to get the bus, they assert their bus request signals. The arbitration checks all bus signal requests every cycle and puts the asserted one in the queue. When the bus is empty, the arbitration removes the first requested processor in the queue and responds by asserting its grant signal. Upon receiving the grant signal, the selected processor places one of the commands mentioned above on the bus command and the address on the bus address line [4].

Snoopy Bus Specification

As mentioned previously in the specification of CDP, the main formula for coordinating usage of the bus by the two processors is $CentraliseBusArbitration()$. Its specification uses four main state variables *(DataBus, AddBus, CmdBus, Queue, Grant)* and two main subformulae which are *CheckReqBus()* to check the bus availability and the request of acquiring the bus from the processors. and *CheckWbRes()* to check and mange the response for a cache request by another cache or the memory. Here is the formula of $CentraliseBusArbitration()$.

$$CentraliseBusArbitration() \mathrel{\widehat{=}} CheckReqBus() \wedge CheckWbRes() \tag{6.2}$$

In every bus cycle, each cache checks the address bus *AddBus* against its tags using the the formula *CacheBusInterface()*, and the bus arbitration *CheckWbRes()* detects the result of the snoop from all caches by checking their *BusReqWB* and *BusyOut* flags. As shown in Fig. 6.10, one function of the snoop result is to inform the main memory or a cache that is holding a modified copy of the block to respond to the request. The design guarantees that the snoop results are available after three clock cycles from the issue of the address on the bus. In the first cycle, the *CacheBusInterface()* of each cache checks the address against the tags. If the check hits, it requests to write back the data by setting the *BusReqWB* flag. In the second cycle, the *CheckWbRes()* delays for one cycle because the bus-side controller may not be able to access its tag when the processor-side controller updates the same tag. In the third cycle, the *CheckWbRes()* firstly checks the command bus *CmdBus* and the *BusyOut* flag of each cache. If *CmdBus* equals to one of the transactional commands and one of the caches responds with busy,

$$
\begin{array}{l}
CheckWbRes(p) \mathrel{\widehat{=}} \\
\quad \{\textsf{if } (p = numPro) \\
\qquad \textsf{then } (\bigcirc\, Delay = 0 \wedge \bigcirc\, BusDataReady = f) \\
\qquad\ \textsf{else if } (Grant_p \wedge CachBusReq_p) \\
\qquad\qquad \textsf{then if } (Delay = 3) \\
\qquad\qquad\qquad \textsf{then if } (CmdBus = readFx_{tm} \vee CmdBus = read_{tm} \vee \\
\qquad\qquad\qquad\qquad CmdBus = invCopy_{tm}) \wedge (BusyOut[0] \vee BusyOut[1]) \\
\qquad\qquad\qquad\qquad \textsf{then } (\bigcirc\, BusyIn = t \wedge \bigcirc\, BusDataReady = t) \\
\qquad\qquad\qquad\qquad\ \textsf{else } (\bigcirc\bigcirc\, BusDataReady = t \wedge \\
\qquad\qquad\qquad\qquad\qquad \textsf{if } (BusReqWB[0] \vee BusReqWB[1]) \\
\qquad\qquad\qquad\qquad\qquad\ \textsf{then } \bigcirc\, SendBlk = t \\
\qquad\qquad\qquad\qquad\qquad\ \textsf{else } \bigcirc\, MemSendBlk = t) \\
\qquad\qquad\qquad\ \textsf{else } \bigcirc\, Delay = Delay + 1 \\
\qquad\qquad\ \textsf{else } CheckWbRes(p + 1) \\
\quad \}
\end{array}
$$

FIGURE 6.10: Part of snoopy bus specification.

the *CheckWbRes()* sets the *BusyIn* flag to inform the request that a conflict has been detected. If *CmdBus* isn't one of the transactional commands or there is no busy response from the caches, the *CheckWbRes()* checks the *BusReqWB* of each cache. If there is a *BusReqWB* which equals true, the *CheckWbRes()* sets the *SendBlk* flag to inform the cache that is holding a modified copy of the block to transfer the data. Otherwise, the *CheckWbRes()* sets the *MemSendBlk* to order the main memory to transfer the data. The possibility of the processor changing the tag state during the second cycle doesn't exist, because the bus side controller has priority and it changes the state of both tags directly after it hits.

Bus time cycle

As shown in Figs. 6.9 and 6.10, the description of the time bus cycle for the $readFx_{tm}$, as a sample, transition is as follows:

- T0: $Processor_1()$ requests the bus by setting the *CacheBusReq* flag and waiting for the bus grant.
- T1: The bus arbitration $CentraliseBusArbitration()$ sets the grant line *Grant[1] = true* when the bus is empty; if not, it places the processor number in the queue list *Queue* .
- T2: The cache bus-interface *BusInterface()* of the $Processor_1()$ puts the command and the address on the bus.
- T3: The cache bus-side controller which holds a modified copy sets the write-back flag *BusReqWB* .
- T4: The bus arbitration delays for one cycle.
- T5: The bus arbitration checks the write-back flags and sets the *SendBlk*.
- T6: The cache bus-interface, which holds the modified copy, places the data block on the bus. Also, the bus arbitration sets the *BusDataReady*.
- T7: The cache bus-interface of the $Processor_1()$ receives the data from *DataBus*.

6.6 Discussion

Deadlock

In the SMP the deadlock occurs when each of two cache controllers has an outstanding transaction that the other needs to respond to, and both are refusing to handle requests. The proposed specification avoids this situation by dividing the bus-interface into two parts which work simultaneously. The first part works with processor-side controller which attempting to issue its request. The second part works with bus-side controller which services incoming transaction which may cause it to flush blocks onto the bus.

For example, suppose that a bus read instruction for a block B appears on the bus while a processor P1 has a $readFx$ request outstanding to another block A and is waiting for the bus. If P1 has a modified copy of B, its controller supplies the data and changes the state from modified to shared while it is waiting to acquire the bus.

Livelock

The traditional livelock problem in an invalidation-based cache-coherent memory system is caused by all processors attempting to write to the same memory location. It is possible that the block is brought into the cache in a modified state, but before the processor is able to complete its write the block is invalidated by a bus $readFx_{tm}$ request from another processor. The processor misses again and this cycle can repeat indefinitely.

The proposed specification guarantees that this kind of livelock cannot happen because the bus arbitration does not grant the other requests before getting acknowledgment that the first request has received the data block. Moreover, the other request needs five cycles (as explained in the previous section) before it invalidates the data which has been received by the first request.

Starvation

With multiple processors competing for a bus, it is possible that some processors requests may be repeatedly granted by the bus while others processors are ignored and therefore become starved. The proposed solution for this problem is to use a priority queue in the bus arbitration.

6.7 Refinement and Validation

Modern hardware design is largely based on using HDLs and once we have the specification of our model in form of a HDL, hardware synthesis can be performed automatically using several commercially available synthesisers.

The transformation process between a Tempura specification and a HDL specification should be based on sound techniques such as a refinement calculus. The refinement relation $\sqsubseteq$ is defined on a system as follows: A system X is refined by the system Y, denoted $X \sqsubseteq Y$, if and only if the formula $Y \supset X$ is valid. In actual fact, the denotational, ITL-based semantics of the HDL should be given to formally verify that the HDL specification refines its ITL/TEMPURA

behaviour specification. However, this is out of this thesis's scope and requires much effort and time. We therefor instead propose refinement laws based on some restriction rules to partially accomplish the transformation. To validate this transformation, we execute a shared counter example on both specifications and match the results.

There are two major hardware descriptive languages currently on the market: VHDL, which is an acronym of VHSIC (Very High Speed Integrated Circuit) Hardware Description Language, and Verilog. We have chosen VHDL is selected to be used as the hardware description language in this work for the following reasons:

1. There is a similarity in its behavioural description with AnaTempura.
2. It has the capable of handling large designs.
3. Most designs of Field Programmable Gate Array (FPGA) are in VHDL. This facility allows automated synthesis, via several commercially available synthesisers, of a VHDL description on a chip and testing it against the circuit.
4. We have previous experience in writing of VHDL code.

Moreover, the VHDL has many features: Firstly, designs may be decomposed into sub-designs, and interconnected between those sub-designs. Secondly, behavioural specification can use either a familiar programming language or an actual hardware structure to describe an element's operation. Thirdly, timing and clocking can be modelled. VHDL allows the use of explicit time delays. In particular, it is possible to say that a statement is executed after a certain time delay [74, 75].

6.7.1 VHDL Structure and Modeling

VHDL Code Structure

As shown in Fig. 6.11, VHDL code is composed of at least three fundamental sections [75]:

- Library declaration: This contains a list of all libraries to be used in the design, such as ieee.
- Entity: This is the VHDL representation of such a block and can be considered to be at the top of the design hierarchy.
- Architecture: An implementation of the entity containing VHDL code which describes the circuit behaviour.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY example IS
  PORT(      X : IN     STD_LOGIC_VECTOR( 5 DOWNTO 0 );
             Y : IN     STD_LOGIC;
             Z : OUT    STD_LOGIC
              );
END ENTITY example;

ARCHITECTURE example OF example IS
            SIGNAL  R  : STD_LOGIC;
BEGIN
                  R  <=  '1'  WHEN  X = "000000"  ELSE '0';
                  Z  <=  R and Y;

 END ARCHITECTURE example;
```

FIGURE 6.11: Example of VHDL program.

VHDL Modeling Styles

An architecture block can be written in one of the VHDL modeling styles as follows:

- Dataflow: represents the concurrent execution style. It describes the circuit in terms of the flow of data and operations through the circuit. Its style architecture includes: operators logical, relational and mathematical. In addition to the concurrent assignments statements.
- Behavioural: represents the sequential execution style. However, it contains concurrent statements with section of sequential statements that describe the output of the circuit.
- Structural: represents the interconnection of components. It describes the circuit in term of components. The main topics associated with this style are: components declaration and port mapping, in addition to signals for interconnection.

6.7.2 Restrictions and Refinement Rules

Restrictions

In order to make the transformation from Tempura to VHDL more straightforward and according to the mutual properties for both Tempura and VHDL, we list the following rules concerning restrictions:

1. The architecture section and its dataflow design style only concerns executing the statements concurrently an arbitrary ordered (involved the process statements).

2. The delay mechanisms such as delta and inertial delays are not considered here, so we only handle the terminated computation.

3. The synthesieble types only are considered (no scalar type) which are either the in and out signals in the entity declaration, or the signals in the architecture declaration.

4. The binary numbers in VHDL such as '1' and '0' represents the Boolean type in Tempura (true and false).

5. The terminated computation of the VHDL state that happened between two clock events is equivalent to a one Tempura state.

6. Signal assignment statements with more than one waveform are not considered here. We will consider the output at the end time of the clock.

7. The signals that regarded as out signals in VHDL are not used as an input or checked in the corresponding Tempura.

Refinement Rules

The following rules enable the transformation of Tempura constructs into VHDL constructs with regards to the previous restrictions. Where f is a formula and A and B are state variables (Tempura) and signals (VHDL).

TABLE 6.1: The proposed refinement rules for Tempura/VHDL transformation

VHDL		Tempura
{1} $A <= f_0;\ B <= f_1$	$\sqsubseteq$	$A = f_0 \wedge B = f_1$
{2} $A <='\ 1';\ B <='\ 0'$	$\sqsubseteq$	$A = true \wedge B = false$
{3} f_1 WHEN f_0 ELSE f_2;	$\sqsubseteq$	if f_0 then f_1 else f_2
{4} PROCESS BEGIN WAIT UNTIL clock'EVENT AND clock = '1'; IF f_0THEN f_1 ELSE f_2 END IF; END PROCESS	$\sqsubseteq$	if f_0 then $(skip;\ f_1)$ else f_2
{5} label:FOR i IN 0 TO j GENERATE f_0 END GENERATE	$\sqsubseteq$	$(\bigwedge_{i=0}^{j} f_0)$
{6} PROCESS BEGIN WAIT UNTIL clock'EVENT AND clock = '1'; FOR i IN 0 TO j LOOP f_0 END LOOP; END PROCESS	$\sqsubseteq$	(forall $i < j$:$\{skip;\ f_0\}$)
{7} $L\ ((n-1) - x$ downto $n - y)$	$\sqsubseteq$	Represent sublist L, length n $L[x$ to $y]$
{9} CONV_STD_LOGIC_VECTOR(i,l)	$\sqsubseteq$	Conv_Std_Logic_Vector(i,l)
{10} CONV_INTEGER(i)	$\sqsubseteq$	Conv_Integer(i)
{11} $X\&Y$	$\sqsubseteq$	Concatenate(X,Y)

6.7.3 Transformation and Validation

To simplify the transformation process and its correctness validation of the proposed CDP specification, we start the transformation with the main formulae that represent the major functional entities of the processor data path, such as the control and execution units, and then gradually transfer other formulae such as the centralised bus arbitration until we reach our final model. After each transformation, we validate its correctness by executing a shared counter example on both specifications and comparing their results.

Mentor-Graphics 6.3 is used as an editing and simulation tool for VHDL code since it is easy to learn and use and it supports more than one synthesis tool such as Precision and LeonardoSpectrum.

In this section, transformation of two formulae of the CDP specification is shown, as an example, which are the control and instruction decode units. The other VHDL codes of the CDP are shown in Appendix C.

Control Unit

The VHDL equivalent of the control unit specification in Tempura that is presented in this chapter (see Section 6.3), is shown in Fig. 6.12.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;

ENTITY control IS
  PORT(
      Opcode      :IN     STD_LOGIC_VECTOR( 5 DOWNTO 0 );
      RegDst      :OUT    STD_LOGIC;
      ALUSrc      :OUT    STD_LOGIC;
      MemtoReg    :OUT    STD_LOGIC;
      RegWrite    :OUT    STD_LOGIC;
      CacheRead   :OUT    STD_LOGIC;
      cacheWrite  :OUT    STD_LOGIC;
      ALUop       :OUT    STD_LOGIC_VECTOR( 1 DOWNTO 0 ));
END control;

ARCHITECTURE control OF control IS
SIGNAL  R_format, Lt,Ltx,St,Lw,Sw ,Bne     : STD_LOGIC;
BEGIN

      R_format    <= '1' WHEN  Opcode = "000000" ELSE '0';
      Lw          <= '1' WHEN  Opcode = "100011"  ELSE '0';
      Sw          <= '1' WHEN  Opcode = "101011"  ELSE '0';
      Lt          <= '1' WHEN  Opcode = "000001"  ELSE '0';
      Ltx         <= '1' WHEN  Opcode = "000010"  ELSE '0';
      St          <= '1' WHEN  Opcode = "000011"  ELSE '0';
      Bne         <= '1' WHEN  Opcode = "000101"  ELSE '0';
      RegDst      <= R_format;
      ALUSrc      <= Lt OR Ltx OR St OR Lw OR Sw ;
      MemtoReg    <= Lt OR Ltx OR Lw;
      RegWrite    <= Lt OR Ltx OR R_format OR Lw ;
      CacheRead   <= Lw ;
       CacheWrite <= Sw ;
       ALUOp(1)   <= R_format;
       ALUOp(0)   <=  Bne;

 END control;
```

FIGURE 6.12: The VHDL equivalent of the control unit specification.

Instruction Decode

The VHDL equivalent of the instruction decode entity that is presented in this chapter (see Section 6.3), is shown in Fig. 6.13.

```
BEGIN

      ReadAddrReg1      <= Inst( 25 DOWNTO 21 );
       ReadAddrReg2     <= Inst( 20 DOWNTO 16 );
       WriteAddrReg1    <= Inst( 15 DOWNTO 11 );
       WriteAddrReg2    <= Inst( 20 DOWNTO 16 );
       InstImmedValue   <= Inst( 15 DOWNTO 0 ) ;

       ReadData1 <= RegFi(CONV_INTEGER( ReadAddrReg1  ) );
       ReadData2 <= RegFi(CONV_INTEGER( ReadAddrReg2  ) );

       WrData <= ALU_result          WHEN  MemtoReg = '0'   ELSE MemDataread;

       AddrWr <= WriteAddrReg1       WHEN RegDst = '1'      ELSE WriteAddrReg2;

       PROCESS
       BEGIN

        WAIT UNTIL clock'EVENT AND clock = '1';
          IF reset = '1' THEN
            FOR i IN 0 TO 31 LOOP
            RegFi(i) <= CONV_STD_LOGIC_VECTOR( i, 32 );
             END LOOP;

          ELSIF RegWrite = '1'  THEN
              RegFi( CONV_INTEGER( AddrWr)) <=WrData;

          END IF;

    END PROCESS;


END ARCHITECTURE InstDecode;
```

FIGURE 6.13: Part of the VHDL equivalent of the instruction decode specification.

Shared Counter Example

To validate the correctness of the Tempura and VHDL specification of the CDP, we execute a simple shared counter example on both specifications and compare their results.

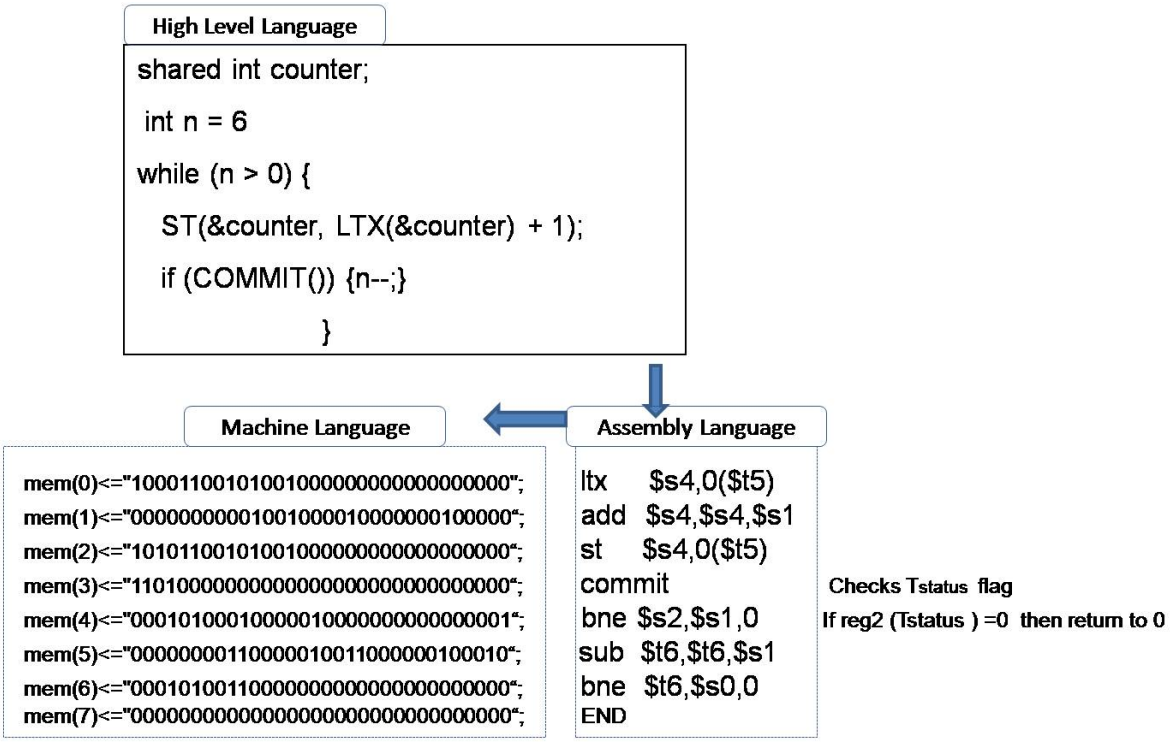

FIGURE 6.14: Example of the shared counter.

As shown in Fig. 6.14 , a simple shared counter example which increments the counter (initialised to 0) 6 times is transferred from high-level to assembly and then to machine language in order to be executed on both specifications. It reads the shared counter from its address in the main memory $(\&counter)$ into a local register (register number 4) using ltx and then it increments the counter by 1. The $commit$ instruction then attempts to make the temporary update of

the shared counter permanent by checking the $Status$ flag . To simplify the control flow of the program after executing a $commit$ instruction, we let $commit$ to set or reset register number 2 in the register file of each processor according to the status of the transaction flag $Status$. The branch statement bne, that follows the $commit$ instruction, can check register number 2 and return back to execute the transaction from the initial statement if it is reset, transaction aborted, otherwise continue the program.

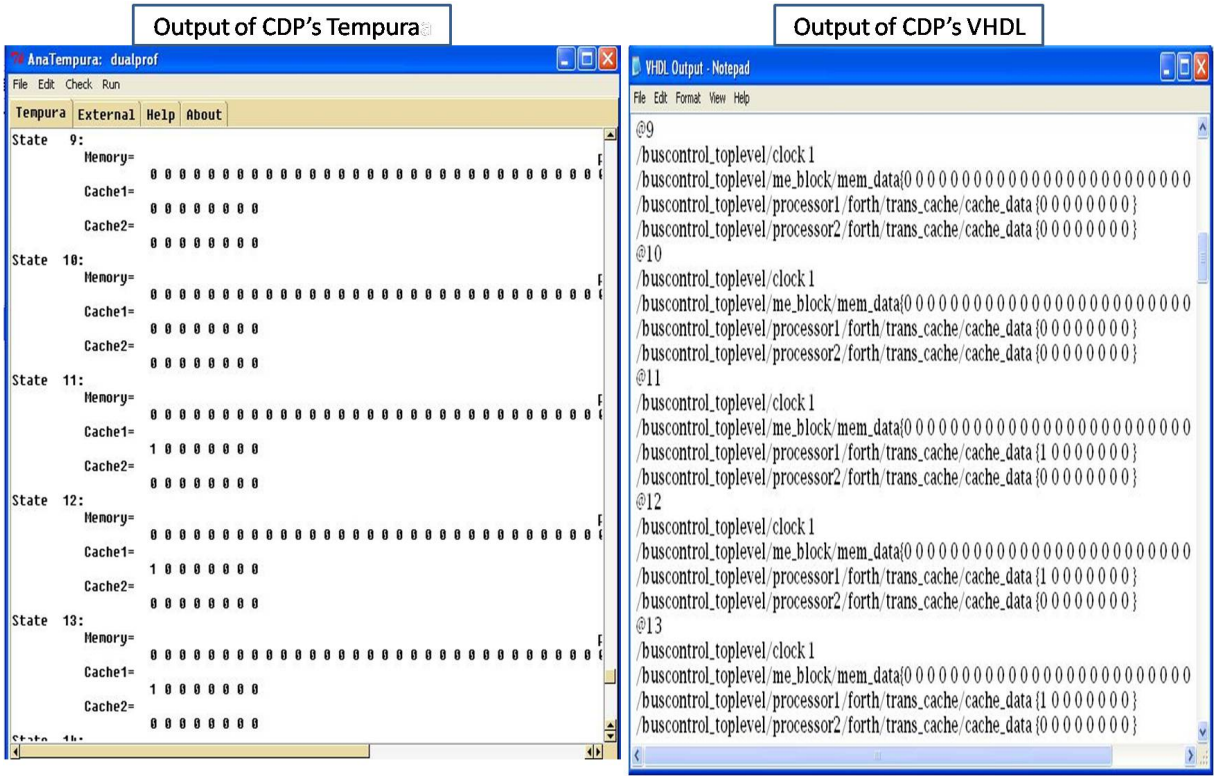

FIGURE 6.15: Output of the shared counter execution (part 1).

Figs. 6.15 to 6.17 show the AnaTempura and Mentor-Graphics output results of executing the shared counter example on the executable and VHDL specification of CDP. Fig. 6.15 shows the

cache value of the shared counter in the first processor after changing from zero to one at state 11 (clock 11) , where each processor needs 8 states to read a value from the main memory into its cache and two states to update it, see Section 6.5.

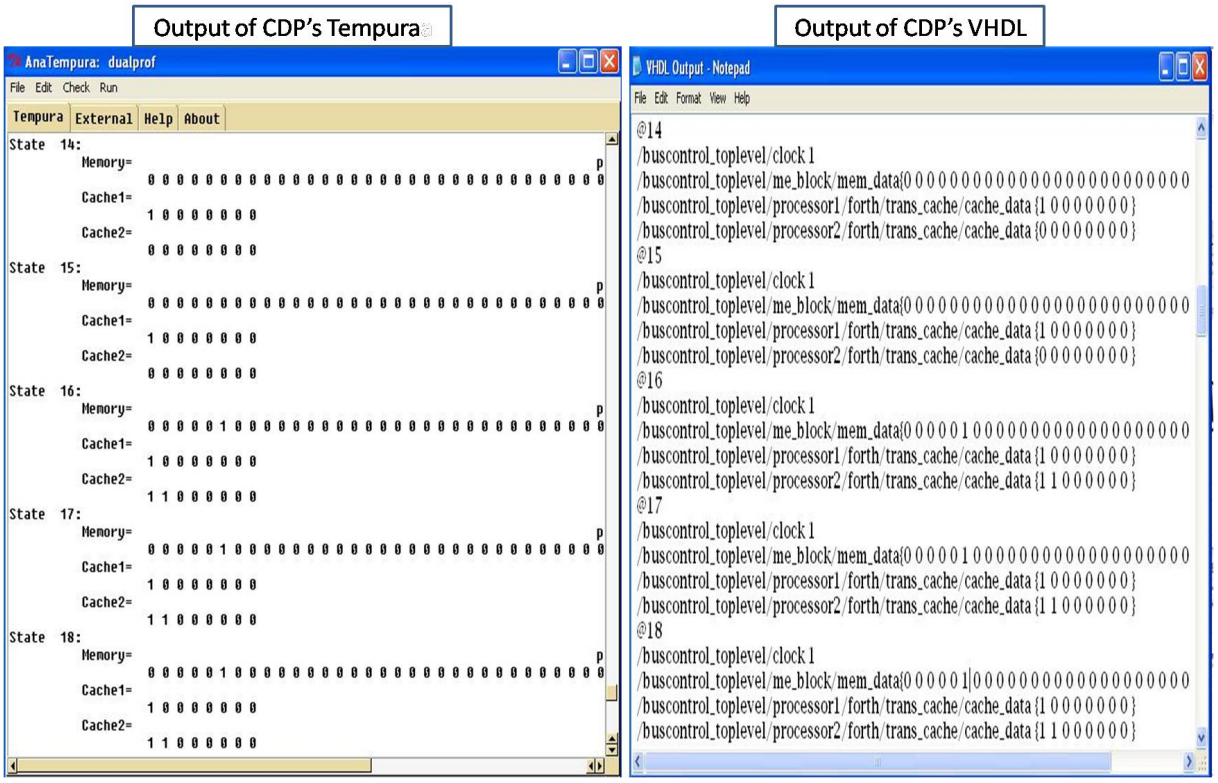

FIGURE 6.16: Output of the shared counter execution (part 2).

As shown in the Fig. 6.16, the value of the shared counter in the main memory (location 5) will not be updated until the second processor issues a request for the counter and the first processor sends the new value of the counter at state 16 (clock 16), see Section 6.4.

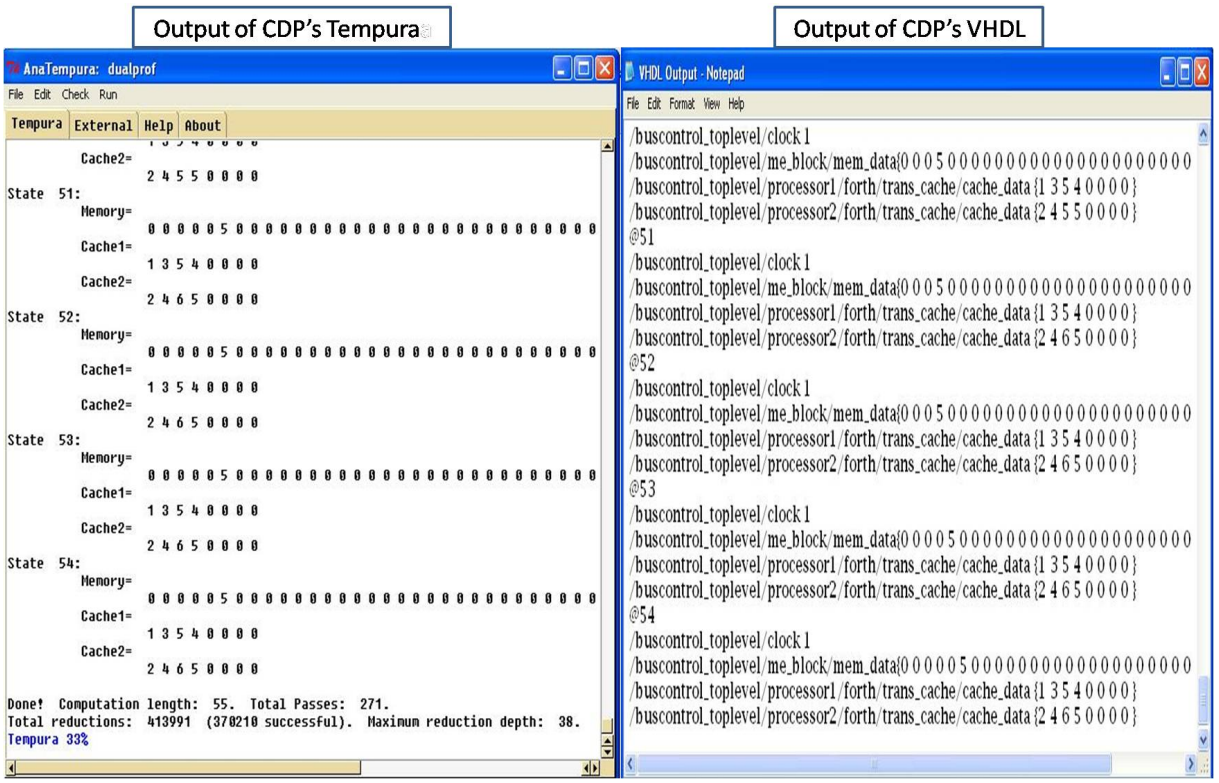

FIGURE 6.17: Output of the shared counter execution (part 3).

Fig. 6.17 shows the final value of the shared counter in cache2 after updating its value from 5 to 6 at state 51. The rest of the states to check the final value of the increment times of the shared counter and end the while loop.

6.8 Summary

In this chapter, an extension to the proposed formal TM framework is presented which is a specification of a shared memory system with dual processors CDP is presented using AnaTempura. The major benefit of this extension is the ability to evaluate the provable TM system in a real shared memory environment. In addition, the integration process of the provable TM system with a shared memory system is more complex at the low-level description than the high-level one. Moreover, the formal specification of a shared memory system that is integrated with a TM technique can eventually be verified correct.

We faced some difficulties in the specification of the CDP using AnaTempura such as its limitations (no memory model, no data structures, it cannot represent all ITL operators and system can run only one time). Even so, it has some advantages such as being closer to a normal programming language and having a simulation tool which can represent results graphically.

Also in this chapter, a refinement of rules based on some restrictions for transferring the executable specification of CDP with TM to the hardware description language VHDL and its validation are presented.

Chapter 7

Conclusion and Future Work

7.1 Summary of Thesis

In this thesis, a general and flexible formal TM framework that allows specifying, validating, verifying and implementing a TM system is developed. In addition, modelling and verification of the standard safety properties in the TM community are provided. The main feature of this framework is that it can specify, validate and analyse a TM system's behaviour within a single logical formalism, namely ITL and its executable subset, AnaTempura.

The construction of the framework's main part which is a provably correct abstract TM model involves four stages: Firstly, a computation model for an abstract TM is specified. Secondly, various TM correctness conditions such as the read-consistency and strict serialisability are specified. Moreover, a new conflict detection policy such as strong-eager-conflict and a new

arbitration function such as the eager-own-arbitration are modelled. Thirdly, a set of sound refinement rules are used to transform the abstract TM specification into an executable model. This model is validated and animations through testing using AnaTempura are given. Fourthly, a simplification of the mathematical verification method is proposed and used to prove the correctness of the proposed abstract TM.

As a case study, the well-known, original Hardware TM (HTM) system of Herlihy and Moss [6] was selected. We provide its correctness in three steps: Firstly, a concrete specification, close to reality, is given. Secondly, a validation to get the right specification was presented, but a violation for the doomed consistency in this model was captured and a modified model proposed. Thirdly, the correctness of the modified HTM system is proven by using a refinement mapping technique which maps its transition behaviour states with the provably abstract TM model states.

A unique characteristic of the proposed framework is that it can integrate the provable HTM system within a real shared memory environment and transform them both to the low-level hardware description language VHDL. The shared memory system CDP is built by specifying dual single cycle MIPS processor, a direct-mapped data cache equipped with a MESI cache coherency protocol with each processor and snoopy bus protocol. The similarity in the VHDL behaviour description with AnaTempura allows for a straightforward construction of the CDP with TM from their specifications.

7.2 Contributions

This thesis develops a unified formal framework for specifying, validating, verifying and implementing a TM system using a single well-defined formalism. This framework involves:

- A general computational model for an abstract TM.
- A formal description of the standard TM safety conditions such as policies for doomed consistency, strict serialisability, and conflict detection and resolution.
- An executable version for the abstract TM model.
- A verification technique for correctness of a TM model based on a mathematical proof.
- A high-level specification of a selected TM system from the literature to serve as a case study, and its executable version.
- A correctness verification for the TM case study by using a refinement mapping technique that maps its transition behaviour states to the provably correct abstract TM model states.
- A formal executable specification for a chip-dual single-cycle MIPS processor with an MESI cache coherence protocol, as a shared memory environment, and integration of the TM case study which we have verified.
- A transformation of the dual processors with the TM system from a high level description into a hardware description language, using proposed refinement and restriction rules.

7.3 Success Criteria Revisited

A set of criteria are presented in Chapter 1 to judge the success of the proposed research. This section revisits these measures of success.

- *The formal specification of the TM safety properties.*

  This important criterion to verify and validate the correctness of the proposed approach. The most common correctness requirement in TM community which is strict serializability with respect to doomed consistency is formalised. In addition, many other safety conditions that can help to verify the correctness such as global and local consistency are formalised (see Chapter 4).

- *The simplification of the TM formal verification.*

  The research investigation shows correctness verification for a TM system by constructing a provably correct abstract TM model and mapping its states to the transition behaviour states of the target TM system using refinement mapping rules. The correctness verification of the abstract TM is simplified by viewing the TM 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 (see Chapters 4 and 5).

- *The capability of the validation process in the proposed approach using ITL framework.*

The work-bench of ITL, including its executable subset, Tempura, and its simulation and the animation tool, AnaTempura, help to define, execute and simulate properties of interest efficiently and correctly. In additional to the benefits of the ITL operators in capturing the concurrent behaviours of transactions, the proposed approach uses this work-bench in the construction of abstract TM, case study and CDP with TM to get the right specification and validate its correctness. For example, the specification of the case study is validated by running a concurrent data structure example in Chapter 5. A violation of TM safety property is captured and a modification for the case study is proposed.

- *The realisation capability of the proposed approach. For example, the possibility to build a TM system from high-level specification to low-level hardware.*

  Quite a lot attention is paid to the practical part of the approach during the development. The infrastructure of the shared memory environment is built to make our approach worthwhile. The main components of our approach that involve the provably correct abstract TM model and the formal specification of the TM safety properties in additional to the high-level specification and low-level hardware description of the CDP are efficient enough for real practice in designing , verifying and implementing many proposed TM systems. The validation of the high-level specification of the case study and its transformation to low-level implementation by running a real concurrent shared counter example show that our approach is a practical one (see Chapter 6). One of the main advantages of the infrastructure of CDP and TM is to help the researchers to develop various models of interactions between transactions and non-transactional code.

7.4 Limitations

The proposed research described in this thesis has the following limitations:

- The generality of the proposed abstract TM model gives us the capability to use it as a standard and match it to different TM systems. However, this generality is not quite enough and cannot be matched to all existing proposed systems. There are still other TM aspects should be imported to the provable abstract TM such as nested transactions, mechanisms of updating the memory.

- Applying a TM system on our framework in order to check its correctness and then transform it to hardware level needs much effort in the specification stage. Its specification should be close to reality and more than just an abstract concept. However, the cost of building a specification close to implementation is where the complexity lies. In actual fact, our proposed framework requires much effort in order to understand a TM system before applying it.

- The verification and the refinement into Tempura and VHDL stages in the proposed framework are still manual, which is more complex as a result.

7.5 Future Work

ITL and TM are both challenge. The thesis provides a foundation for future research in this promising area. Many aspects and properties of TM rely on the history of concurrent transactions to be correctly investigated such as the relationship between transactional and non-transactional access. In [55] the time reversal technique was proposed for compositional verification. This technique can be used also in the compositional verification of TM properties.

Moreover, here are some tasks for future investigation:

- Verify the correctness of different HTM systems. Moreover, investigate and prove the correctness of Software TM (STM) and Hybrid TM (HyTM) systems.
- Provide mechanical verification using special-purpose theorem provers such as PVS or KIV.
- Expand the generality of the provable abstract TM model to involve the other important aspects of the TM design such as nested transactions, mechanisms of updating the memory and more conflict management policies.
- Soundly prove the refinement rules of the transformation from AnaTempura to VHDL.

Bibliography

[1] K. Olukotun, L. Hammond, and J. Laudon. *Chip Multiprocessor Architecture: Techniques to Improve Throughput and Latency*. Morgan and Claypool, 1st edition, 2007.

[2] N. Njoroge, J. Casper, S. Wee, Y. Teslyar, D. Ge, C. Kozyrakis, and K. Olukotun. ATLAS: A Chip-Multiprocessor with Transactional Memory Support. In *Proceedings of the Conference on Design Automation and Test in Europe*, pages 3–8. EDA Consortium, April 2007.

[3] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The Case for a Single Chip Multiprocessor. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 2–11. ACM, October 1996.

[4] D. Culler, J. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1st edition, 1998.

[5] J. Larus and C. Kozyrakis. Transactional Memory. *Communications of the ACM*, 51(7):80–88, December 2008.

[6] M. Herlihy and J. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. *SIGARCH Computer Architecture News*, 21(2):289–300, May 1993.

[7] J. Larus and R. Rajwar. *Transactional Memory*. Morgan and Claypool, 2nd edition, 2010.

[8] J. Larus and C. Kozyrakis. Is TM the Answer for Improving Parallel Programming? *Communication of the ACM*, 51(7):80–88, July 2008.

[9] M. Tremblay. Transactional Memory for a Modern Microprocessor. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, PODC '07, pages 1–1. ACM, 2007.

[10] L. Hammond, V. Wong, M. Chen, B. Carlstrom, J. Davis, B. Hertzberg, M. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional Memory Coherence and Consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, page 102. IEEE Computer Society, Jun 2004.

[11] A. El-Kustaban, A. El-Mahdy, and O. Ismail. A CMP with Transactional Memory: Design and Implementation Using FPGA Technology. In *Proceeding of the International MultiConference of Engineers and Computer Scientist*, IMECS '07, pages 1680–1685. Newswood Limited, 2007.

[12] V. Marathe, W. Scherer III, and M. Scott. Adaptive Software Transactional Memory. In *Proceedings of the 19th International Conference on Distributed Computing (DISC 2005)*. LNCS, Springer, September 2005.

[13] A. Shriraman, M. Spear, H. Hossain, V. Marathe, S. Dwarkadas, and M. Scott. An Integrated Hardware-Software Approach To Flexible Transactional Memory. In *Proceedings of the 34rd Annual International Symposium on Computer Architecture*, ISCA '07, pages 104–115. ACM, 2007.

[14] T. Harris, A. Cristal, O. Unsal, E. Ayguade, F. Gagliardi, B. Smith, and M. Valero. Transactional Memory: An Overview. *Micro, IEEE*, 27(3):8 –29, May-June 2007.

[15] B. Moszkowski. Some Very Compositional Temporal Properties. In E.-R. Olderog, editor, *Programming Concepts, Methods and Calculi*, volume A-56 of *IFIP Transactions*, pages 307–326. IFIP, North Holland, 1994.

[16] B. Moszkowski. Compositional Reasoning about Projected and Infinite Time. In *Proceedings of the First IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 1995)*, pages 238–245. IEEE Computer Society Press, 1995.

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

[18] A. Cau, B. Moszkowski, and H. Zedan. Interval Temporal Logic, 2012. [Webpages] http://www.tech.dmu.ac.uk/~STRL/ITL/index.html.

[19] H. El-Rewini and M. Abd-El-Barr. *Advanced Computer Architecture and Parallel Processing*. John Wiley and Sons, 2005.

[20] M. J. Flynn. Very high-speed computing systems. *IEEE*, 54(12):1901–1909, December 1966.

[21] R. Duncan. A Survey of Parallel Computer Architecture. *Computer*, 23(2):5–16, February 1990.

[22] E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569–, 1965.

[23] E. W. Dijkstra. Cooperating Sequential Processes. In F. Genuys, editor, *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968. Originally appeared as EWD123 in 1965.

[24] L. Lamport. A New Solution of Dijkstra's Concurrent Programming Problem. *Communication of the ACM*, 17(18):453–455, 1974.

[25] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[26] K. Fraser. *Practical Lock-Freedom*. PhD thesis, University of Cambridge, 2003.

[27] E. Jensen, G. Hagensen, and J. Broughton. A New Approach to Exclusive Data Access in Shared Memory Multiprocessor. Technical Report UCRL-97663, Lawrence Liverome National Laboratory, November 1987.

[28] M. Herlihy. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, 1991.

[29] M. Herlihy, V. Luchango, and M. Moir. Obstruction-free Synchronization: Double-ended queues as an example . In *Proceedings of the 23rd IEEE International Conference on Distributed Computing System*, ICDCS '03, pages 522–529. IEEE Computer Society, 2003.

[30] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw- Hill, New York, 2000.

[31] G. Taubenfeld. *Synchronization Algorithms and Concurrent Programming*. Pearson Education. Pearson/Prentice Hall, 2006.

[32] G. Taubenfeld. Concurrent Programming, Mutual Exclusion. In *Encyclopedia of Algorithms*. 2008.

[33] D. Porter and E. Witchel. Understanding Transactional Memory Performance. In *Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Software Systems*, pages 97–108. IEEE Computer Society, March 2010.

[34] K. Moore, J. Bobba, M. Moravan, M. Hill, and D. Wood. LogTM: Log-based Transactional Memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, pages 254–265. IEEE Computer Society, February 2006.

[35] N. Shavit and D. Touitou. Software Transactional Memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213. ACM, 1995.

[36] S. Lie. Hardware Support for Unbounded Transactional Memory. Master's thesis, Massachusetts Institute of Technology, May 2004.

[37] L. Hammond, B. Hubbert, M. Siu, M. Prabhu, M. Chen, and K. Olukotun. The Stanford Hydra CMP. *IEEE Micro.*, 20(2), 2000.

[38] J. Wawrzynek and D. Patterson et al. RAMP: Research Accelerator for Multiple Processors. *IEEE Micro*, 27(2):46–57, 2007.

[39] M. 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. [Online]. Available: http://www.cs.rochester.edu/u/scott/papers/2006_TRANSACT_formal_STM.pdf.

[40] R. Guerraoui, T. 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. [Online]. Available: http://www.cs.utah.edu/ec2/kapalka.pdf.

[41] 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)*, pages 175–184. ACM, 2008.

[42] A. Cohen, J. O'Leary, A. Pnueli, M. Tuttle, and L. Zuck. Verifying Correctness of Transactional Memories. In *Proceedings of the 7th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 37–44. IEEE Computer Society, November 2007.

[43] R. Guerraoui, T. Henzinger, and V. Singh. Completeness and Nondeterminism in Model Checking Transactional Memories. In *Proceedings of the 19th Conference on Concurrency Theory (CONCUR 2008)*, pages 21–35. Springer, August 2008.

[44] A. Sinha and S. Malik. Runtime Checking of Serializability in Software Transactional Memory. In *Proceeding of the IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–12. IEEE, April 2010.

[45] S. Tasiran. A Compositional Method for Verifying Software Transactional Memory Implementations. Technical Report MSR-TR-2008-56, Microsoft Research, Redmond, USA, April 2008. [Online]. Available: http://research.microsoft.com/pubs/70570/tr-2008-56.pdf.

[46] W. Weihl. Local Atomicity Properties: Modular Concurrency Control for Abstract Data Types. *ACM Transactions on Programming Languages and Systems*, 11(2):249–282, April 1989.

[47] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[48] A. Pnueli and T. Arons. TLPVS: A PVS-based LTL Verification System. In *Proceedings of an International Symposium In Verification-Theory and Practice*, pages 84–98. Springer-Verlag, 2003.

[49] A. Cohen, A. Pnueli, and L. Zuck. Verification of Transactional Memories that Support Non-Transactional Memory Accesses. In *TRANSACT '08: 3rd Workshop on Transactional Computing*, February 2008. [Online]. Available: http://www.unine.ch/transact08/papers/Cohen-Verification.pdf.

[50] K. Moore and D. Grossman. High-level small-step operational semantics for transactions. In *Proceeding of the 35th ACM Symposium on Principles of Programming Languages*, pages 51–62. ACM, 2008.

[51] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing Memory Transactions. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 14–25. ACM, 2006.

[52] J. Dimitrov. *Developing Semantics of Verilog HDL in Formal Compositional Design of Mixed Hardware / Software Systems*. PhD thesis, Software Technology Research Laboratory, De Montfort University, 2002.

[53] A. Cau, R. Hale, J. Dimitrov, H. Zedan, B. Moszkowski, M. Manjunathaiah, and M. Spivey. A Compositional Framework for Hardware/Software Co-Design. *Design Automation for Embedded Systems*, 6(4):367–399, 2002.

[54] A. Cau and H. Zedan. Refining Interval Temporal Logic Specifications. In *Proceedings of the 4th International AMAST Workshop on Real-Time Systems and Concurrent and Distributed Software: Transformation-Based Reactive Systems Development*, ARTS '97, pages 79–94. Springer-Verlag, 1997.

[55] B. Moszkowski. Compositional Reasoning using Intervals and Time Reversal. In *Proceedings of the 18th International Symposium on Temporal Representation and Reasoning (TIME 2011)*, pages 107–114. IEEE Computer Society, 2011.

[56] Zhenhua Duan, Xiaoxiao Yang, and Maciej Koutny. Framed Temporal Logic Programming. *Science Computer Program.*, 70(1):31–61, 2008.

[57] A. Cau, C. Czarnecki, and H. Zedan. Designing a Provably Correct Robot Control System using a 'Lean' Formal Method. In Anders P. Ravn and Hans Rischel, editors, *Proceedings of the 5th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT'98)*, pages 123–132. Springer Verlag, 1998.

[58] A. Cau, N. Coleman H. Zedan, and B. Moszkowski. Using ITL and TEMPURA for Large Scale Specification and Simulation . In *The 4th Euro micro Workshop on Parallel and Distributed Processing*, pages 493–500. IEEE Computer Society Press, 1996.

[59] H. Janicke, A. Cau, F. Siewe, and H. Zedan. Deriving Enforcement Mechanisms from Policies. In *Proceedings of the 8th IEEE International Workshop on Policies for Distributed Systems and Networks*, POLICY '07, pages 161–172. IEEE Computer Society, 2007.

[60] H. Zedan, A. Cau, and B. Moszkowski. Compositional modelling: The formal perspective. In David Bustard, editor, *Proceedings of Workshop on Systems Modelling for Business Process Improvement*, pages 333–354. Artech House, 2000.

[61] H. Zedan and A. Cau. Voice Over IP: Correct Hardware/Software Co-design. In *8th IEEE Workshop on Future Trends of Distributed Computer Systems (FTDCS 2001), 31 October, 2 November 2001, Bologna, Italy, Proceedings*, pages 194–200. IEEE Computer Society, 2001.

[62] F. Siewe. *A Compositional Framework for the Development of Secure Access Control Systems*. PhD thesis, Software Technology Research Laboratory, De Montfort University, Leicester, 2005.

[63] M. Abadi and L. Lamport. The Existence of Refinement Mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[64] M. Emmi, R. Majumdar, and R. Manevich. Parameterized Verification of Transactional Memories. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation (PLDI 2010)*. ACM, 2010.

[65] H. Papadimitriou. The Serializability of Concurrent Database Updates. *J. ACM*, 26(4):631–653, 1979.

[66] W.P. de Roever, F. Boer, U. Hannemann, J. Hooman, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge University Press, 2001.

[67] G. Schellhorn and S. Baumler. Formal Verification of Lock-Free Algorithms. In *Proceedings of the 9th International Conference on Application of Concurrency to System Design (ACSD 2009)*, pages 13–18. IEEE Computer Society, 2009.

[68] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th annual international symposium on Computer Architecture*, ISCA '90, pages 148–159. ACM, 1990.

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

[70] B.Welch. *Practical Programming in Tcl and Tk*. Prentice Hall, New Jersey, 2nd edition, 1997.

[71] D. Libes. *Exploring Expect*. O'Reilly and Associates, 1995.

[72] H. Gao, Y. Fu, and W. Hesselink. Verification of a Lock-Free Implementation of Multiword LL/SC Object. In *Proceedings of the 8th IEEE International Conference on Dependable, Autonomic and Secure Computing*, DASC '09, pages 31–36. IEEE Computer Society, 2009.

[73] D. Patterson and J. Hennessy. *Computer Organization and Design - the Hardware /Software Interface*. Morgan Kaufmann, 3rd edition, 2007.

[74] P. Ashenden. *The VHDL Cookbook*. Department of Computer Science University of Adelaide, South Australia, 1990.

[75] D. L. Perry. *VHDL: Programming by Example*. McGraw-Hill, 2002.

Appendix A. Executable Specification of Abstract TM

In order to validate the correctness of the proposed abstract TM tm_{spec} and make such examinations for TM safety properties, we build an executable specification for tm_{spec}.

The main components of the tm_{spec} are represented in Tempura as follows: Processes' status P is represented as an array with state values $\{free, busy\}$. Also, transactions' status T is represented as an array with states values $\{idle, active, doomed, finished\}$. In addition, events E is represented as an array of lists recording each event with possible values $\{no_{ev}, r, w, ok, tryCom, tryAbort, \oplus, \otimes\}$.

As shown in Fig. 1, the transaction operations described in Tables 4.2 (page 51) and 4.3 (page 54) are represented as four main functions: Firstly, the $TranInvOp(p, t, op, \varepsilon, \varepsilon_r)$ to deal with the invocation operations (op) *read* and *write* . The ε and ε_r in any functions to specify the type of conflict detection and resolution respectively. Secondly, the $TranResOp(p, t)$ to response the last operation by transaction t and process p. Thirdly, $TranInvEnd(p, t, op, \varepsilon, \varepsilon_r)$ to deal

with the invocation operations *tryCom* and *tryAbort*. Finally, the $TranResEnd(p, t)$ to commit or abort transaction t and release process p.

Some auxiliary functions are used in the executable model of the tm_{spec} such as the $AddEv()$ that is used 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 lists 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.

As shown in Fig. 1, there are others functions to represent the main formulae of tm_{spec} such as the conflict detection and resolution formula $ConflictDetRes()$, that uses one of conflict detection types which are explained in detail in the previous section, and the response actions of read operation $ValidRead()$.

State variables:
P_p : Process status $\in \{free, busy\}$; where $(0 \leq p < |Processes|)$; initially $free$
T_p^t: Transaction status $\in \{idle, active, doomed, finished\}$; where $(0 \leq t < |Tr|)$; initially $idle$
E_p^t : An array of lists recording each event $\in \{no_{ev}, r, w, ok, tryCom, tryAbort, \oplus, \otimes\}$; initially no_{ev}
$Mem[obj]$: Persistent memory $(0 \leq obj < |Locations|)$; 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 $\{AddEv(p, t, op)$
     $\wedge\ ConflictDetRes(p, t, \varepsilon, \varepsilon_r)\}$
    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 $T_p^t := doomed\}$

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

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

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

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

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

FIGURE 1: Core part of TM executable specification

Testing with Animation

We use the executable specification tm_{spec}, which is refined into AnaTempura, to execute some examples. In addition, some animation for our model is provided to make it more understandable and enable the reader to gain better insight into the TM system.

As shown in Figs. 2- 7, the user interface for the graphical output is divided into five parts: Firstly, the timer grade which represents the number of state. Secondly, three processes where each contains the number of the process and is covered by a unique colour. Thirdly, the global memory block, that is represented for the permanent write. Finally, a transaction number and its sequence of operations and responses are shown in the space between the process number and the memory. Although there are differences between database and memory transactions such as the computation time in memory which is negligible relative to access time in the database, for the sake of simplicity we use the bank account and airline reservation examples that are described in Chapter 2 to illustrate the validation of our model.

Example 1: Bank Account

Bank accounts are accessed simultaneously by more than one operation. The conflicts between two operations should be detected and resolved:

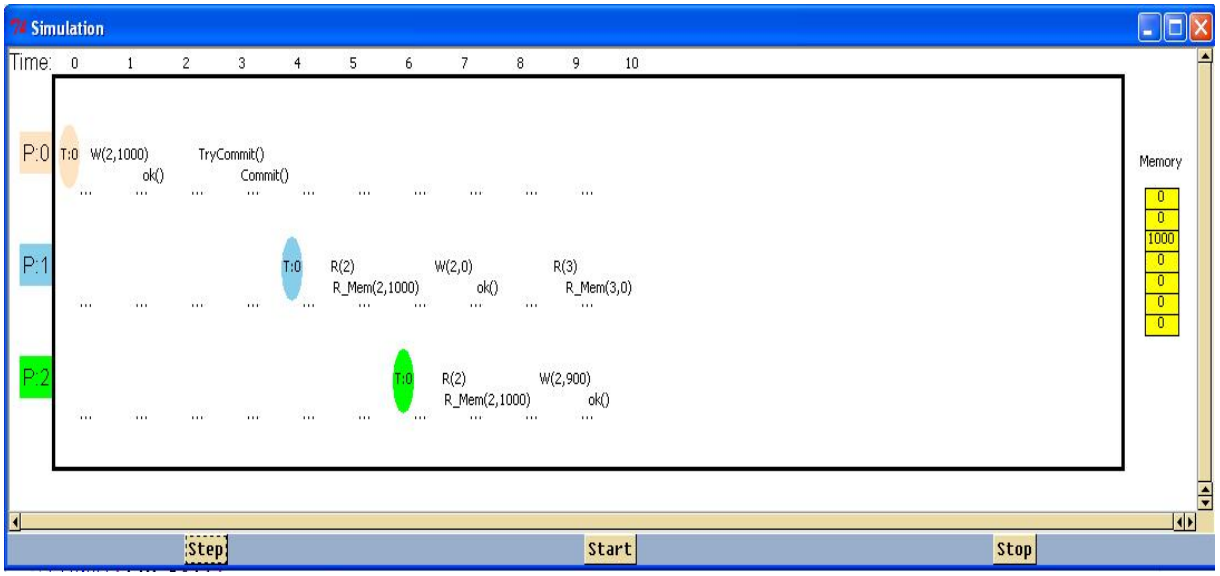

FIGURE 2: Part 1 of example 1

P_0 invokes T_0 to deposit 1000. After P_0 commits T_0^0, the 1000 is resident in the memory location [2]. Then P_1 invokes T_1^0 to transfer the 1000 from location [2] to location [3]. In the same period, P_2 invokes a transaction to withdraw 100 from location [2].

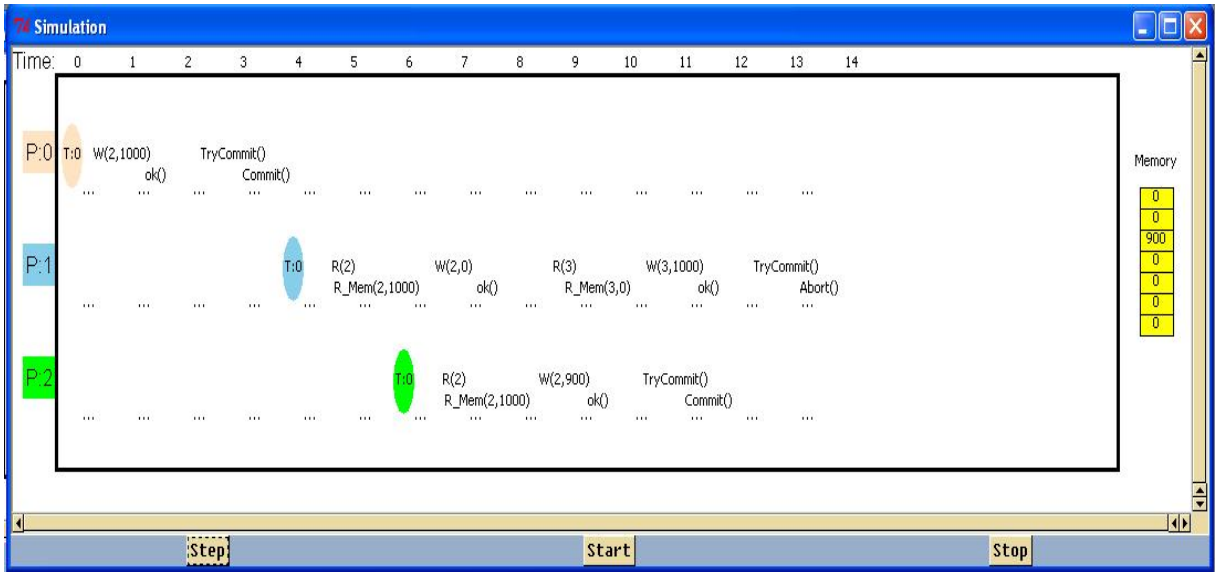

FIGURE 3: Part 2 of example 1

When T_2^0 invokes *TryCommit*, the tm_{spec} responds with *commit* and changes the value of location [2] in the memory to 900 . Also, the tm_{spec} detects a conflict between *write* operation in T_2^0 and *read* in T_1^0. Since the resolution policy of this model is *Lazily*, which says whoever tries to commit first wins, it responds to T_2^0 with *commit* and to T_1^0 with *abort*. The effect of T_1^0's transferring steps is deleted, as if it never happened at all, and location [2] still equals 900. The global read consistency is satisfied here, since the successful T_2^0 returns the most recent $W_0^0(2, 1000)$ in a committed transaction.

Example 2: Airline Reservation

Multiple transactions can read a specific location at the same time but only one can modify it.

Here, more than one conflict should be detected and resolved.

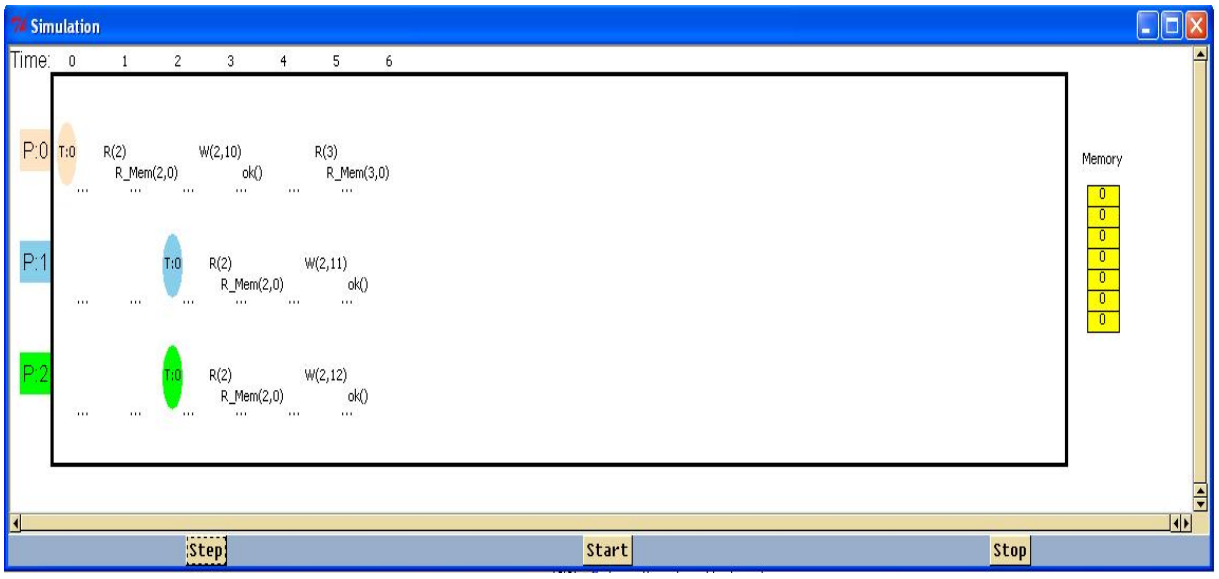

FIGURE 4: Part 1 of example 2

P_0 invokes T_0 to make a reservation for two seats (locations [1] and [2]) by writing 10 to these locations. Then T_1^0 and T_2^0 are invoked at the same time from different terminals to reserve seat [2] by writing 11 and 12, respectively.

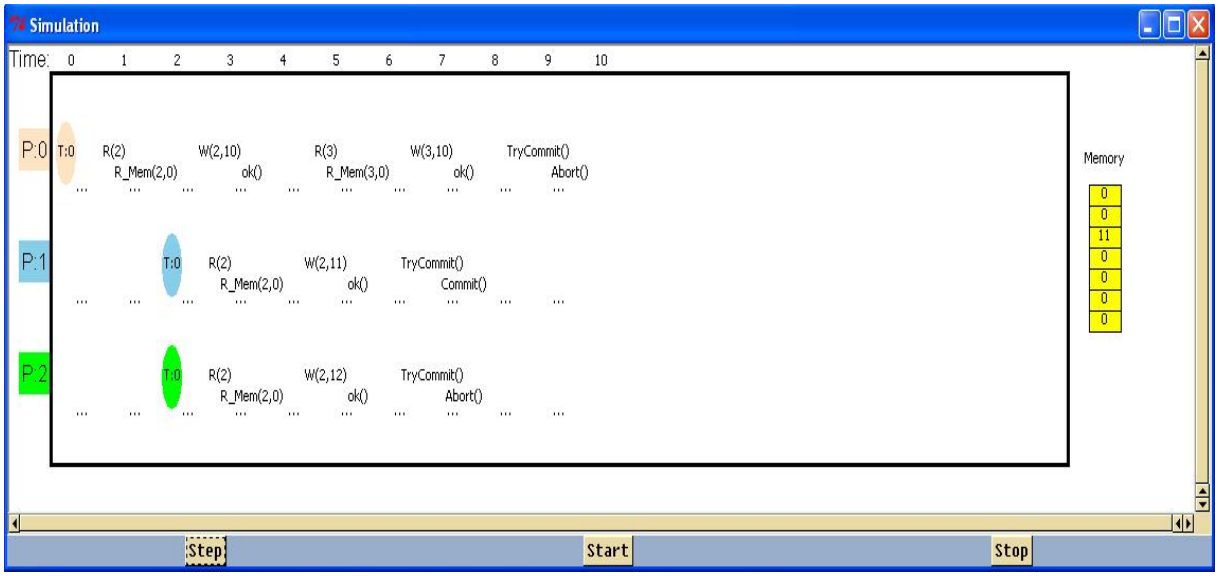

FIGURE 5: Part 2 of example 2

Only T_1^0 is committed and the others are aborted. The explanation is as follows: When T_1^0 and T_2^0 issue a *tryCommit* operation at the same state, the tm_{spec} detects a conflict in write operation between them. Here, a special case appears which is detected as a conflict at the same time between T_1^0 and T_2^0 which are invoked at the same state as well. We cover this case by adding a priority policy P0, P1, P2 to the resolution function. So, this policy commits T_1^0 and aborts T_2^0, while lazily aggressive policy aborts T_0^0 because of the conflict with the committed transaction T_1^0. So, we can observe that seat [2] equals 11 which means that it is reserved by T_1^0

Example 3: Doomed Consistency Validation

To illustrate doomed consistency and how the tm_{spec} reacts when the doomed transaction tries to access an inconsistent value, we show the execution of three concurrent transactions. Transactions T_0^0 and T_2^0 try to read the value of object B that is modified by the third transaction T_1^0. However, T_0^0 has early read for object A which is modified later by T_1^0.

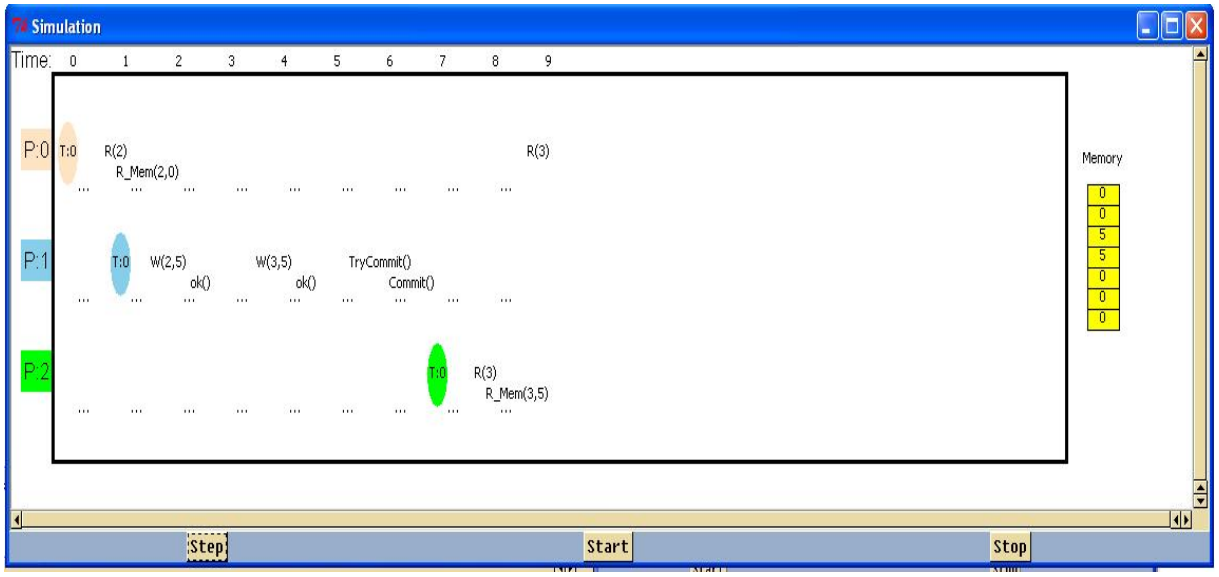

FIGURE 6: Part 1 of example 3

T_0^0 starts by reading the value of object A (location [2]). Then, it is suspended for a period until T_1^0 changes the values of object A (location [2]) and object B (location [3]) to 5. When T_1^0 commit, T_0^0 resumes its operations by reading object B. Also, T_2^0 is invoked to read object B which is responded to with 5 (global consistency).

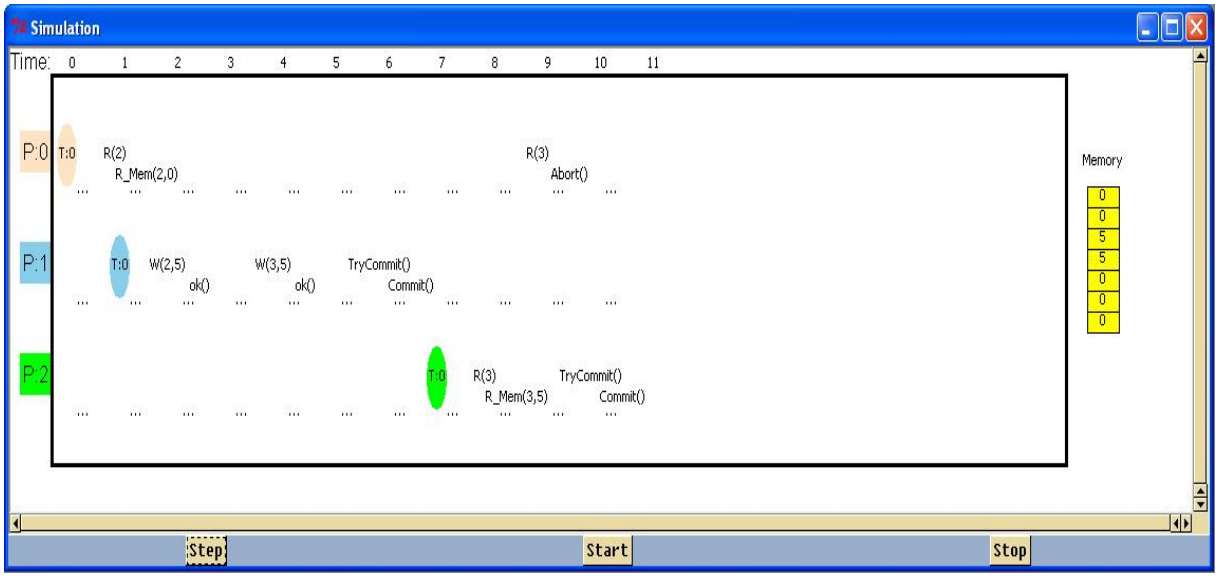

FIGURE 7: Part 2 of example 3

When T_0^0 attempts to read the value of object B (location [3]), it gets abort. However, when T_2^0 attempts to read B, it is responded with 5 and then commit. The reason for aborting T_0^0 is the inconsistent state that is captured by *InconsRead()* function, which validates the doomed consistency of every reading response. The inconsistent read case appears when T_0^0 attempts to get a response for reading object B, *InconsRead()* finds that T_0^0 issues a reading operation before this for object A and its value is changed later by T_1^0.

Appendix B. Executable Specification of CDP

Figs. 8 and 9 show the executable specification of the microprocessor's execution unit .

```
define ExecuteUnit(Func, Signed_extend)={

        exists ALU_ctl,Ainput,Binput,Binput0,Bnegate,ALU_mux:{

                ALU_ctl[ctLen-0]= (( Func[fuLen-0] or Func[fuLen-3] ) and ALUOp[opLen-1])      and
                ALU_ctl[ctLen-1]= (( ~ Func[fuLen-2]) or (~ ALUOp[opLen-1] ) )                 and
                ALU_ctl[ctLen-2]= (( Func[fuLen-1] and ALUOp[opLen-1] ) or ALUOp[opLen-0])     and
                 Ainput=ReadData1                                                              and
                    /* funcfield sub=010 add=000 and=100 or=101*/
                          if (ALU_ctl=[t,t,f] or ALU_ctl=[t,t,t])
                            then
                                 Binput0= Negate(ReadData2)
                                 else {Binput0=ReadData2}                                  and
                          if ALUSrc=f
                            then
                                  Binput= Binput0
                                  else {Binput= Signed_extend}                              and
                           if (ALU_ctl=[t,t,f] or ALU_ctl=[t,t,t])
                              then
                                     Bnegate=t
                                else   {Bnegate=f}                                        and
                                       /* zero set */
                              if ALU_mux =X(f,32)   /* function returns a Boolean list of 32 equals false*/
                                then
                                     Zero = t
                                else {Zero=f}                                             and
```

FIGURE 8: Specification of Execution Unit (part1).

```
                    if (ALU_ctl=[t,t,t])
                       then
                            ALU_result=Concatenate(X(f,31) ,ALU_mux[0])
                        else {ALU_result=ALU_mux}                              and

                      if (ALU_ctl=[t,t,f] or ALU_ctl=[f,t,f] )
                         then
                              {ALU_mux=FullAdd(Ainput,Binput,Bnegate)        and
                               Overflow=OverflowFun(Ainput,Binput,Bnegate)}
                          else if (ALU_ctl=[t,t,t])
                                  then
                                        {ALU_mux=FullAdd(Ainput,Binput,Bnegate)     and
                                             Overflow=f}
                                 else if (ALU_ctl=[f,f,f])
                                         then
                                              ALU_mux=AND(Ainput,Binput)
                                         else  if (ALU_ctl=[f,f,t])
                                                  then
                                                       ALU_mux=OR(Ainput,Binput)
                                                  else {ALU_mux= X(f,32)
                     }
}
```

FIGURE 9: Specification of Execution Unit (part2).

Fig. 10 shows the executable specification of the transactional full associative cache .

```
Define Cache(p, Read,   Write, CacheData_in , Address_xab, Address_xcom, allocate,
                    MemDataValid, WBft, WBsd) = {

if Reset = t
  then  forall i<cacheLen: {others(CacheData[p][i],data_width,"X")}
  else  if (Read=1)
          then{ if (allocate=1)
                   then{  CacheData[p][Conv_Integer(Address_xcom)]:=CacheData[p][Conv_Integer(Address_xab)]} and

                if (WBft=0 and WBsd=0)
                     then   Cache2CPU[p]:=CacheData[p][Conv_Integer(Address_xab)]
                     else    BuffWBft[p]:=CacheData[p][Conv_Integer(Address_xab)]
                     }

          else if (Write=1)
                  then{
                        CacheData[p][Conv_Integer(Address_xab)]:=CacheData_in and
                        if(MemDataValid=1)
                          then  /*it means that data comes from bus so must be written into two Loc*/
                               CacheData[p][Conv_Integer(Address_xcom)]:=CacheData_in}
                          }

}
```

FIGURE 10: Transactional Cache.

Figs. 11 and 12 show the executable specification of the replacement technique *Allocate2Loc()* and *AllocateXcommit()*. The function *Allocate2Loc()* searches for two locations in $Cache_p$ when this cache needs space for a new entry.

```
define  Allocate2Loc(p)={

                ft=Allocate1Loc(p,empty1,0,0) and
                sd=Allocate1Loc(p,empty1,-1,0) and
                    CheckWriteBack(p,ft,sd)
                                                }
```

```
define  Allocate1Loc(p,state1,fs,index)={

            if (fs~=-1 and TagPro[p][index][5 to 7]=state1)
               then
                    {index}
                else if (fs=-1 and TagPro[p][index][5 to 7]=state1)
                        then
                              Allocate1Loc(p,state1,0,index+1)
                         else  if (state1=empty1 and index=cacheLen)
                                  then
                                       Allocate1Loc(p,normal,fs,0)
                                   else if (state1=normal and index=cacheLen)
                                           then
                                                Allocate1Loc(p,xcommit,fs,0)
                                            else
                                                Allocate1Loc(p,state1,fs,index+1)
}
```

FIGURE 11: Allocate two locations in the transactional cache.

```
define  AllocateXcommit(p,state1,fs,index)={

          if (index~=fs and TagPro[p][index][5 to 7]=state1)
            then
                 {index}
            else
                 if (state1=empty1 and index=cacheLen)
                   then
                      AllocateXcommit(p,normal,fs,0)
                  else
                       if (state1=normal and index=cacheLen)
                         then
                              AllocateXcommit(p,xcommit,fs,0)
                       else
                              AllocateXcommit(p,state1,fs,index+1)
}
```

```
define  FindSDcopy(p,stateSD,CPU_add,index)={

          if (CPU_add[0 to 3]  = TagPro[p][index][0 to 3]  and
             TagPro[p][index][3 to 5]~= invalid              and
             TagPro[p][index][5 to 7]= stateSD )
           then      { index }
           else
                FindSDcopy(p,stateSD,CPU_add,index+1)
}
```

FIGURE 12: Allocate the second location for x_{commit} entry in the transactional cache.

Fig. 13 , shows the executable specification of the queue operations part in the centerlised bus arbitration.

```
define  RemoveQueue()={
                        forall i<Tail: Queue[i]:=Queue[i+1] }and
define  InQueue(i,index)={

                  if index=numPro
                     then 0
                     else if Queue[index]=i
                             then 1
                             else  InQueue(i,index+1)

 }and


define  InsertOthersQu(i,tail,remove)={
        if i=numPro
          then StableQueue(tail,remove)
          else { Grant[i]:=0 and
                 if (CachReqBus[i]=1 and (InQueue(i,0)=0))
                   then
                      { Queue[tail]:=i  and
                        InsertOthersQu(i+1,tail+1,remove)
                      }
                   else InsertOthersQu(i+1,tail,remove)
                 }
}and

define StableQueue(tail,remove)={
       if Tail=tail
         then{(forall i<numPro: Queue[i]:=Queue[i]) and Tail:=Tail}
          else  { Tail:=tail and
                  if remove=1 then {
                                    (forall  i<numPro: if (i>tail)
                                                          then Queue[i]:=Queue[i])
                                                          else
                                                              (forall  i<numPro: if ((i<Tail) or (i>tail-1)) then Queue[i]:=Queue[i])
                                    }

}and
```

FIGURE 13: Queue operations.

Appendix C. VHDL Code of CDP

Figs. 14 and 15 show the equivalent VHDL code of the executable specification of the micro-processor's execution unit .

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE IEEE.STD_LOGIC_SIGNED.ALL;
---------------------------------------------
ENTITY ExecuteUnit IS
     port (
          ReadData1,ReadData2:     in   std_logic_vector(31 downto 0);
          ALUSrc:                  in   std_logic;
          ALUOp:                   in   std_logic_vector(1 downto 0);
          Func:                    in   std_logic_vector(5 downto 0);
          Sign_extend:             in   std_logic_vector(31 downto 0);
          Overflow,Zero:           out  std_logic;
          Alu_result:              out  std_logic_vector(31 downto 0));
   END MIPSexecute ;

 ARCHITECTURE ExecuteUnit OF ExecuteUnit IS
signal Binput,Ainput,Binput0:        std_logic_vector(31 downto 0);
signal Alu_mux:                       std_logic_vector(31 downto 0);
signal Bnegate:                       std_logic;
signal ALU_ctl:                       std_logic_vector(2 downto 0);

BEGIN
 ALU_ctl( 0 ) <= ( Func( 0 ) OR Func( 3 ) ) AND ALUOp(1 );
 ALU_ctl( 1 ) <= ( NOT Func( 2 ) ) OR (NOT ALUOp( 1 ) );
 ALU_ctl( 2 ) <= (Func( 1 ) AND ALUOp( 1 )) OR ALUOp( 0 );
```

FIGURE 14: The VHDL equivalent of the Execution Unit formula (part 1).

```
 Ainput<=ReadData1;
 Binput0<= not(ReadData2)     when (ALU_ctl="110" or ALU_ctl="111")
             else   ReadData2;
 Binput<= Binput0                when ALUSrc<='0'
            else   Sign_extend  ;

 Bnegate<='1'                 when (ALU_ctl="110" or ALU_ctl="111" )
              else '0';

 Zero <= '1'               when Alu_mux =x"00000000"
              else '0';

               -- slt set
 ALU_result <= "00000000000000000000000000000000" & Alu_mux(31)       when  ALU_ctl = "111"
                    else   Alu_mux;


Alu_mux<= FullAdd(Ainput,Binput,Bnegate)                when  (ALU_ctl ="110"  or ALU_ctl ="010"  )
         else  Alu_mux<= FullAdd(Ainput,Binput,Bnegate)    when  ALU_ctl <="111"
         else  Alu_mux<= Ainput and Binput                  when  ALU_ctl <="000"
         else  Alu_mux<= Ainput or Binput                   when  ALU_ctl <="001"
         else  Alu_mux<= x"00000000" ;
Overflow<=OverflowFun(Ainput,Binput,Bnegate) when  (ALU_ctl ="110"  or ALU_ctl ="010"  )
          else  '0' ;

END ARCHITECTURE ExecuteUnit ;
```

FIGURE 15: The VHDL equivalent of the Execution Unit formula (part 2).

Figs. 16 and 17, show the equivalent VHDL code of the executable specification of the transactional full associative cache .

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.std_logic_unsigned.all;
ENTITY Cache IS

 port (
        clk,Reset          : IN     std_logic;
        Read               : IN     std_logic;
       Write               : IN     std_logic;
       allocate            : IN     std_logic;
       Cache2CPU           : OUT    std_logic_vector (31 DOWNTO 0);
       BuffWBft            : OUT    std_logic_vector (31 DOWNTO 0);
       CacheData_in        : IN     std_logic_vector (31 DOWNTO 0);
       Address_xabo        : IN     std_logic_vector (address-1 DOWNTO 0);
       Address_xcom        : IN     std_logic_vector (address-1 DOWNTO 0);
      MemDataValid,WBsd,WBft   : IN std_logic

             );
END ENTITY Cache;
```

FIGURE 16: Transactional Cache (part 1).

```
ARCHITECTURE Cache OF Cache IS
type cache_type is array( 0 to 2**index_width-1)of std_logic_vector(data_width-1 downto 0 );

  SIGNAL CacheData    : cache_type;

BEGIN
    ReadWrite:process(clk,reset)
       begin
        if (clk'event and clk='1' )
           then if reset = '1'
                   then
                         FOR i IN 0 TO 2**index_width-1) LOOP
                         CacheData(i)<= (others=>'X');
                         END LOOP;
                  else
                     if (Read='1')
                        then  if allocate='1'
                                 then  CacheData(CONV_INTEGER(Address_xcom))<=CacheData(CONV_INTEGER(Address_xab));
                                end if;

                              if(WBft='0' and WBsd='0')
                                 then   Cache2CPU<= CacheData(CONV_INTEGER(Address_xab));
                                 else   BuffWBft <= CacheData(CONV_INTEGER(Address_xab));
                              end if;
                         else  if (Write='1')
                                  then (CONV_INTEGER(Address_xab))<=CacheData_in;
                                        if(MemDataValid='1')
                                          then   CacheData(CONV_INTEGER(Address_xcom))<=CacheData_in;
                                        end if;
                                  end if;
                           end if;
                     end if;
              end if;
        end process;

    END ARCHITECTURE Cache;
```

FIGURE 17: Transactional Cache (part2).