A Programming Language Approach to Transactional Memory Consistency

Sandeep Hans
A Programming Language Approach to Transactional Memory Consistency

Research Thesis

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Sandeep Hans

Submitted to the Senate of the Technion — Israel Institute of Technology
Elul 5775 Haifa September 2015
This research was carried out under the supervision of Prof. Hagit Attiya, in the Faculty of Computer Science, with Dr. Noam Rinetzky, from Tel-Aviv University, as co-advisor.

Some results in this thesis have been published as articles by the author and research collaborators in conferences and journals during the course of the author’s doctoral research period, the most up-to-date versions of which being:


Acknowledgements

Firstly, I would like to express my sincere gratitude to my advisor Prof. Hagit Attiya for the continuous support of my Ph.D study and related research, for her patience, motivation, and immense knowledge. Her guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my Ph.D study. I hope that one day I would become as good an advisor to my students as Hagit has been to me.

I would also like to thank my co-advisor, Dr. Noam Rinetzky, who has been always there to listen and give advice. I am deeply grateful to him for the long discussions that helped me sort out the technical details of my work.

I would also like to thank my co-authors Alexey Gotsman, Petr Kuznetsov and Srivatsan Ravi, who allowed me to include the research done with them in this thesis. Special thanks to Alexey for encouraging the use of correct definitions and consistent notations in my writings, and for carefully reading and commenting on countless revisions of our joint work.

I would like to thank Prof. Panagiota Fatourou and Eleni Kanellou for hosting me in Crete, Greece. The research discussions with them and other group members was very helpful in improving my knowledge in the area.

I would also like to thank all my friends in Technion who made my stay in Technion memorable and enjoyable. Their support and care helped me overcome setbacks and stay focused on my graduate study.

Most importantly, I would like to thank my family, especially my parents, for supporting me throughout my Ph.D and my life in general. This thesis is dedicated to them.

Finally, I appreciate the financial support from the European Commission for Marie Curie fellowship that funded parts of the research discussed in this dissertation.
6 Other Consistency Conditions

6.1 Opacity ................................................................. 65
6.2 Deferred-update (DU) opacity ...................................... 68
6.3 Virtual World Consistency (VWC) .............................. 69
  6.3.1 Sufficiency of VWC ............................................. 72
6.4 Transactional Memory Specification 2 (TMS2) ............... 78

7 STMS is a safety property ............................................. 79

8 Related Work

  8.1 Transactional Memory Basics ................................. 83
  8.2 Database Consistency Conditions ........................... 84
  8.3 Programming Language Semantics for TM .................. 84
  8.4 Observational Refinement .................................... 85

9 Conclusions ............................................................. 87

Hebrew Abstract .......................................................... i
List of Figures

1.1 Example of transactional memory usage; global is initially 0. 8
1.2 Example of transactional memory usage; X is initially 0, Y is initially 1, global is initially 0. 9
1.3 Relations between TM consistency conditions. 10

2.1 The definition of $\text{Tr}(P)$. A trace $\tau \in \text{interleave}(\tau_1, \ldots, \tau_m)$ if and only if every action in $\tau$ is performed by some thread $t \in \{1, \ldots, m\}$, and $\tau|_t = \tau_t$ for every $t \in \{1, \ldots, m\}$. The function $\text{prefix}(\cdot)$ prefix-closes a given set of traces. 21

3.1 Transactions $T_1$, $T_3$, $T_4$ and $T_5$ form one possible past of the history $H$ of the trace shown. Allowed status of transactions in $H$ is denoted as follows: committed – C, aborted – A, commit-pending – CP, live – L. The transaction $T_5$ executes only primitive actions after $\psi$ in the trace. $\chi_1$ and $\chi_2$ are non-transactional actions by threads $t_1$ and $t_2$, respectively, and $g$ and $g'$ are global variables. 30

4.1 An illustration of the transformations performed in the proof of Lemma 4.1 36
4.2 Cases in the proof of Lemma 4.4. * all actions by $t$ are transactional; ** all actions by $t$ come from a single transaction, started before or by $\psi^b$. 42

5.1 Constants derived from the history $H$ that are used to construct $P_H$ (see Figure 5.3). 49
5.2 The variables used in $P_H$ and their their intended meaning (see Figure 5.3). 50
5.3 The construction of $P_H$ for Lemma 5.1, Lemma 5.2 (the case of $t \neq t_0$), Lemma 5.3 and Lemma 5.4 (the case of $t \neq t_0$). For conciseness, we use an extension of the programming language with conditionals without an “else” clause. The loop command can be implemented by $\text{while}(\text{true}) \text{ do skip}$. 52
5.4 The construction of $\tau^t_i$ for Lemma 5.1, Lemma 5.2 (the case of $t \neq t_0$), Lemma 5.3 and Lemma 5.4 (the case of $t \neq t_0$). 53
5.5 Summary of operations used to generate $H^c$ from $H = \text{history}(\tau)$ and $S$ from $S_1 = \text{history}(\tau')$ in Lemma 5.1. $T$ is a maximal transaction in $H$ and $T'$ is the corresponding transaction in $S$. Case cp/cp - requires the check inside. Check constant is not uninitialized - not aborted! 

6.1 Examples of STMS, VWC, Opaque, DU-opaque and TMS2 histories. $w(x,v)$ denotes a write object operation that writes the value $v$ to $x$, and $r(x,v)$ denotes a read object operation that reads the value of $x$, and gets $v$ as response. $x$ and $y$ are initially 0.

8.1 Relations between Database consistency conditions.
Abstract

Transactional memory (TM) has been hailed as a paradigm for simplifying concurrent programming. It eases the task of writing concurrent applications by letting the programmer designate certain code blocks as atomic.

One of the main challenges in stating the correctness of TM systems is the need to provide guarantees on the system state observed by live transactions, i.e., those that have not yet committed or aborted. While several consistency conditions have been suggested for TM, they fall short of formalizing the intuitive semantics of atomic blocks, the interface through which a TM is used in a programming language. A TM consistency condition should be weak enough to allow flexibility in implementation, yet strong enough to disallow undesirable TM behavior, which can lead to run-time errors in live transactions. The latter feature is formalized by observational refinement between TM implementations, stating that properties of a program using a concrete TM implementation can be established by analyzing its behavior with an abstract TM, serving as a specification of the concrete one.

We show that Strong Transactional Memory Specification (STMS), a variant of Transactional Memory Specification 1 (TMS1), is necessary and sufficient for observational refinement for a programming language in which local variables are not rolled back upon an abort. We also show that TMS1 is necessary and sufficient for observational refinement for the common programming model in which local variables are rolled back upon a transaction abort. This makes STMS and TMS1 the weakest acceptable conditions for these programming models.

We show that STMS is prefix-closed, that is, every prefix of a STMS history is also STMS. We then show that under certain restrictions, STMS is limit-closed, i.e., the limit of any sequence of ever extending STMS histories is also STMS. Therefore, STMS is a safety property, and to prove that a TM implementation is STMS, it suffices to prove that all its finite histories are STMS. We also show that STMS is weaker than Opacity, a well-known consistency condition, and that under certain restrictions, STMS is equivalent to Opacity.

Our results suggest a new approach to evaluate and compare TM consistency conditions. They can also reduce the effort of proving that a TM implements its programming language interface correctly, by only requiring its developer to show that it satisfies the corresponding consistency condition.
# Abbreviations and Notations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aborted(τ)</td>
<td>The set of all aborted transactions in τ.</td>
</tr>
<tr>
<td>addab(H)</td>
<td>The set of all immediate abort extensions of H.</td>
</tr>
<tr>
<td>committed(τ)</td>
<td>The set of all committed transactions in τ.</td>
</tr>
<tr>
<td>comp(H)</td>
<td>The set of suffix-completions of H.</td>
</tr>
<tr>
<td>completed(τ)</td>
<td>The set of all completed transactions in τ.</td>
</tr>
<tr>
<td>copending(τ)</td>
<td>The set of all commit-pending transactions in τ.</td>
</tr>
<tr>
<td>cpcomplete(H)</td>
<td>The set of cp-completions of H.</td>
</tr>
<tr>
<td>cSTMSpast(H)</td>
<td>The set of completed possible pasts of H.</td>
</tr>
<tr>
<td>cTMSpast(H)</td>
<td>The set of completed visible possible pasts of H.</td>
</tr>
<tr>
<td>cVWCpast(H)</td>
<td>The set of completed vwc pasts of H.</td>
</tr>
<tr>
<td>eval&lt;sub&gt;noRB&lt;/sub&gt;</td>
<td>Evaluation function under the semantics without rollback.</td>
</tr>
<tr>
<td>eval&lt;sub&gt;RB&lt;/sub&gt;</td>
<td>Evaluation function under the semantics with rollback.</td>
</tr>
<tr>
<td>fault</td>
<td>A special command which stops the execution of the program in an error state.</td>
</tr>
<tr>
<td>forceaborted(τ)</td>
<td>The set of all force-aborted transactions in τ.</td>
</tr>
<tr>
<td>GVar</td>
<td>The set of global variables.</td>
</tr>
<tr>
<td>History</td>
<td>The set of all histories.</td>
</tr>
<tr>
<td>history(τ)</td>
<td>The projection of τ to TM interface actions.</td>
</tr>
<tr>
<td>justify(H)</td>
<td>The set of commit justifications of H.</td>
</tr>
<tr>
<td>live(τ)</td>
<td>The set of all live transactions in τ.</td>
</tr>
<tr>
<td>loop</td>
<td>A shorthand for a non-terminating loop.</td>
</tr>
<tr>
<td>LVar&lt;sub&gt;t&lt;/sub&gt;</td>
<td>The set of local variables of thread t.</td>
</tr>
<tr>
<td>Term</td>
<td>Definition</td>
</tr>
<tr>
<td>------</td>
<td>------------</td>
</tr>
<tr>
<td>\text{maxtx}(H)</td>
<td>The set of all maximal transactions in $H$.</td>
</tr>
<tr>
<td>\text{nicompleteall}(H)</td>
<td>The set of non-interleaved completions of $H$.</td>
</tr>
<tr>
<td>\text{nocomplete}(H)</td>
<td>The set of non-interleaved cp-completions of $H$.</td>
</tr>
<tr>
<td>Obj</td>
<td>A fixed collection of transactional objects.</td>
</tr>
<tr>
<td>\text{OVWCpast}(H)</td>
<td>The set of ovwc possible pasts of $H$.</td>
</tr>
<tr>
<td>PComm</td>
<td>Primitive commands.</td>
</tr>
<tr>
<td>\text{skip}</td>
<td>A command that does not change any variable.</td>
</tr>
<tr>
<td>\text{STMSpast}(H)</td>
<td>The set of possible pasts of $H$.</td>
</tr>
<tr>
<td>tmaborted(\tau)</td>
<td>The set of all tm-aborted transactions in $\tau$.</td>
</tr>
<tr>
<td>\text{tprefix}(H)</td>
<td>The set of all thread-prefixes of $H$.</td>
</tr>
<tr>
<td>Traces</td>
<td>The set of all traces.</td>
</tr>
<tr>
<td>\text{tx}(\tau)</td>
<td>The set of all transactions in $\tau$.</td>
</tr>
<tr>
<td>Var</td>
<td>The set of all program variables.</td>
</tr>
<tr>
<td>\text{visible}(\tau)</td>
<td>The set of all visible transactions in $\tau$.</td>
</tr>
<tr>
<td>\text{VWCpast}(H)</td>
<td>The set of vwc possible pasts of $H$.</td>
</tr>
<tr>
<td>WfHistory</td>
<td>The set of all well-formed histories.</td>
</tr>
<tr>
<td>WfTraces</td>
<td>The set of all well-formed traces.</td>
</tr>
<tr>
<td>$H \Downarrow \psi$</td>
<td>The prefix of $H$ up to the action $\psi$.</td>
</tr>
<tr>
<td>$</td>
<td>\tau</td>
</tr>
<tr>
<td>$\tau \Downarrow_i$</td>
<td>The prefix of $\tau$ containing $i$ actions.</td>
</tr>
<tr>
<td>$\tau \Downarrow_{\text{com}}$</td>
<td>The projection of $\tau$ to actions by committed transactions.</td>
</tr>
<tr>
<td>$\tau \Downarrow_{t}$</td>
<td>The projection of trace $\tau$ onto actions of thread $t$.</td>
</tr>
<tr>
<td>$\tau \Downarrow_{-t}$</td>
<td>The projection of trace $\tau$ onto actions of threads other than $t$.</td>
</tr>
<tr>
<td>$\tau \Downarrow_{\text{trans}}$</td>
<td>The projection of $\tau$ to transactional actions.</td>
</tr>
<tr>
<td>$\tau \Downarrow_{-\text{trans}}$</td>
<td>The projection of $\tau$ to non-transactional actions.</td>
</tr>
<tr>
<td>$\tau \Downarrow_{\text{abortact}}$</td>
<td>The trace obtained by removing all actions inside aborted transactions.</td>
</tr>
<tr>
<td>Symbol</td>
<td>Definition</td>
</tr>
<tr>
<td>--------</td>
<td>------------</td>
</tr>
<tr>
<td>$\tau</td>
<td>_{\neg \text{aborted tx}}$</td>
</tr>
<tr>
<td>$\tau</td>
<td>_{\neg \text{live}}$</td>
</tr>
<tr>
<td>$\tau_1</td>
<td>_{\neg \tau_2}$</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>The empty trace.</td>
</tr>
<tr>
<td>$[c]$</td>
<td>A function that defines how the program state is transformed by executing $c$.</td>
</tr>
<tr>
<td>$[o]$</td>
<td>A sequential specification of an object $o$.</td>
</tr>
<tr>
<td>$\sim$</td>
<td>Observational equivalence.</td>
</tr>
<tr>
<td>$\approx$</td>
<td>Strong trace equivalence.</td>
</tr>
<tr>
<td>$\equiv$</td>
<td>Trace equivalence up to action identifiers.</td>
</tr>
<tr>
<td>$\parallel$</td>
<td>Parallel composition.</td>
</tr>
<tr>
<td>$&lt;_\tau$</td>
<td>The order of actions in $\tau$.</td>
</tr>
<tr>
<td>$&lt;_H$</td>
<td>The real-time order of actions in history $H$.</td>
</tr>
<tr>
<td>$\preceq_{\text{noRB}}$</td>
<td>Observational refinement under the semantics without rollback.</td>
</tr>
<tr>
<td>$\preceq_{\text{RB}}$</td>
<td>Observational refinement under the semantics with rollback.</td>
</tr>
<tr>
<td>$\preceq_{\text{wRB}}$</td>
<td>Weak observational refinement under the semantics with rollback.</td>
</tr>
<tr>
<td>$\sqsubseteq_{\text{ovwc}}$</td>
<td>Original VWC relation.</td>
</tr>
<tr>
<td>$\sqsubseteq_{\text{RT}}$</td>
<td>Real-time relation.</td>
</tr>
<tr>
<td>$\sqsubseteq_{\text{seq}}$</td>
<td>Sequential consistency relation.</td>
</tr>
<tr>
<td>$\sqsubseteq_{\text{stms}}$</td>
<td>STMS relation.</td>
</tr>
<tr>
<td>$\sqsubseteq_{\text{tms1}}$</td>
<td>TMS1 relation.</td>
</tr>
<tr>
<td>$\sqsubseteq_{\text{tms2}}$</td>
<td>TMS2 relation.</td>
</tr>
<tr>
<td>$\sqsubseteq_{\text{vwc}}$</td>
<td>VWC relation.</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Transaction memory (TM) eases the task of writing concurrent applications by letting the programmer designate certain code blocks as atomic. TM allows developing a program and reasoning about its correctness as if each atomic block executes as a transaction—atomically and without interleaving with other blocks—even though in reality the blocks can be executed concurrently. Figure 1.1 shows a program that consists of two transactions by two different threads. The transaction by the first thread adds a new node to a stack. The transaction by the second thread reads the value of the top node of the stack. This example shows the relative ease in the use of transactional memory by enclosing entire methods into a single atomic block. Both transactions can be executed concurrently, however, TM makes sure that these are executed as if they are atomic. The global variable written by the first thread makes sure that the transaction by the second thread is executed only after the transaction by the first thread is executed successfully.

Many TM implementations have been proposed, e.g., [8, 10, 18, 23, 30, 34]. They use a myriad of design approaches that, for efficiency, may execute transactions concurrently, yet aim to provide the programmer with an illusion that they are executed atomically. This illusion is not always perfect—for example, as evident from Figure 1.1, transactions may abort, typically due to conflicts with concurrently running transactions, and need to be restarted.

How can we be sure that a TM indeed implements atomic blocks correctly? The common approach to stating TM correctness is through a consistency condition that restricts the possible TM executions. This is similar to database consistency conditions, such as serializability [33], which ensure that execution of concurrent transactions is equivalent to their execution atomically in some order. However, serializability provides no guarantees for live transactions, i.e., those that have not yet committed or aborted. Because live transactions can always be aborted, one might think it is not necessary to provide any guarantee for them. However, in the setting of TM, guarantees must be provided on the state of transactional objects observed by live transactions.

For example, Figure 1.2 shows a program that consists of two transactions by two
different threads. The transaction by the first thread reads the values of shared variables \( X \) and \( Y \). The transaction by the second thread writes to \( X \) and \( Y \). Note that even if the transaction by first thread is aborted, the value of the local variable \( z \) inside the transaction should be defined (which might not hold if \( X \) and \( Y \) have the same value). If we allow the transaction to read values of \( X \) and \( Y \) violating the invariant (counting on it to abort later, due to inconsistency), this will lead to the program faulting due to a division by zero. For example, if the transaction by the first thread reads the value 1 of \( X \) written by the second thread and the initial value 1 of \( Y \), this will lead the program to fault when computing the value of \( z \), due to a division by zero.

Several TM consistency conditions have been proposed, e.g., [4, 11, 16, 26]. \textit{Opacity} [16] is the earliest of them; roughly speaking, it requires that for any sequence of interactions between the program and the TM, dubbed a \textit{history}, there exists another history where:

(i) the interactions of every separate thread are the same as in the original history;

(ii) the order of non-overlapping transactions in the original history is preserved; and

(iii) each transaction executes atomically.

\textit{Transactional Memory Specification 1 (TMS1)} [11] and \textit{Virtual World Consistency (VWC)} [26] were proposed as relaxations of opacity and allow every response in a live or aborted transaction to be justified by a separate history. TMS1 allows the justifying histories to include transactions that aborted in the original history, while excluding some committed transactions. VWC places different constraints on the choice of histories, which do not take into account the order of non-overlapping transactions in the original history. \textit{DU-opacity} [4] and \textit{Transactional Memory Specification 2 (TMS2)} [11] are conditions stronger than opacity which put additional restrictions on the justifying history. DU-opacity restricts the set of transactions a transaction is allowed to read from, while TMS2 requires the justifying history to include transactions in their completion order in the original history.

Figure 1.1: Example of transactional memory usage; \texttt{global} is initially 0.
result := atomic {
    x := X.read();
    y := Y.read();
    z := 42 / (x - y);
}
global := z;

result := atomic {
    X.write(1);
    Y.write(0);
}

Figure 1.2: Example of transactional memory usage; X is initially 0, Y is initially 1, 
global is initially 0.

Unfortunately, all these consistency conditions are given from the TM’s point of view, as a restriction on the set of histories a TM can produce. They do not take into account the semantics of the programming language. In fact, it is not clear which consistency condition to use: On one hand, the consistency condition should be strong enough to provide the programmer with behaviors that correspond to the intuitive notion of atomic blocks and, in particular, to disallow undesirable behaviors such as division by zero. On the other hand, the consistency condition should be weak enough to put the minimal restrictions on TM implementations needed to achieve this, and allow as many optimizations as possible.

The first contribution of this thesis is a framework for systematic evaluation of TM consistency conditions by formalizing the intuitive expectations of a programmer as observational refinement [21, 22] between TM implementations. Consider two TM implementations—a concrete one, such as an efficient TM, and an abstract one, such as a TM executing each atomic block atomically. Informally, the concrete TM observationally refines the abstract one for a given programming language if every behavior of a program P in this language that a user can observe, using the concrete TM, can also be observed when P uses the abstract TM instead. This allows the programmer to reason about the behavior of P (e.g., the preservation of the invariant X ≠ Y in Figure 1.2) using the expected intuitive semantics formalized by the abstract TM. Observational refinement implies that the conclusions (e.g., the safety of the division) will carry over to the case when P uses the concrete TM.

We introduce Strong Transactional Memory Specification (STMS), a new consistency condition that is weaker than opacity. We prove that STMS is necessary and sufficient for observational refinement for a programming language in which local variables modified by a transaction are not rolled back upon an abort. This assumption on the programming language holds in some situations (e.g., Scala STM [35]).

The key feature of opacity is that the behavior of all transactions of the concrete TM, including aborted and live ones, has to be justified by a single history of the abstract TM. STMS relaxes this requirement by requiring only completed transactions in the concrete history to be justified by a single abstract history obeying (i)–(iii)
above. Each response obtained from the concrete TM in an aborted or live transaction may be justified by a different abstract TM history. The constraints on the choice of the abstract history are subtle. Somewhat counter-intuitively, STMS allows it to include transactions that aborted in the concrete history, with their status changed to committed, and exclude some that committed. However, this is subject to certain carefully chosen constraints. The flexibility in the choice of the abstract history is meant to allow the concrete TM implementation to perform as many optimizations as possible. However, it is not straightforward to establish that this flexibility does not invalidate observational refinement (and hence, the informal guarantees that programmers expect from a TM).

We prove that STMS is sufficient for observational refinement by showing that a live transaction cannot notice the changes in the committed/aborted status of transactions concurrent with it that are allowed by STMS. Proving that STMS is necessary for observational refinement is challenging as well, as this requires us to devise multiple programs that can observe whether the subtle constraints governing the change of transaction status in STMS are fulfilled by the TM.

STMS is derived from TMS1, which also allows each response in a live or aborted transaction to be justified by a separate history. However, TMS1 does not allow the justifying history to contain previous aborted transactions, while STMS requires the justifying history to contain all the previous committed and aborted transactions. Figure 1.3 summarizes the containment relations between different consistency conditions.

Another key result of the thesis is proving that when local variables are rolled back on transaction abort, TMS1 is necessary and sufficient for observational refinement.

Our results ensure that the STMS and TMS1 definitions cannot be weakened further. Informally, if local variables are not rolled back when transactions abort, threads can communicate to each other the observations they make inside aborted transactions about the state of transactional objects. This requires the TM to provide a consistent
view of this state across all future transactions, as formalized by the inclusion of all previous aborted transactions in the abstract history in STMS. However, if local variables are rolled back upon an abort, no information can leak out of an uncommitted transaction, possibly apart from the fact that the code in the transaction has faulted, stopping the computation. To ensure observational refinement for STMS and TMS1, we only need to make sure that a fault in the transaction occurring with the concrete TM could be reproduced with the abstract one. For this, it is sufficient to require that the state of transactional objects seen by every live transaction can be justified by some abstract history; different transactions can be justified by different histories.

The results for STMS and TMS1 depend on several closure properties on the set of histories produced by the abstract TM. Although intuitive, these properties are not necessarily provided by an arbitrary TM, and our results demonstrate their importance.

We also show that STMS is limit-closed, under certain restrictions. Specifically, assuming that every transaction completes in an infinite history, the limit of any sequence of ever extending STMS histories is also STMS. Therefore, under this assumption, STMS is a safety property, and to prove that a TM implementation that complies with the assumption is STMS, it suffices to prove that all its finite histories are STMS.

Our results depend on the particular choices of programming language and the notion of observations, with the following features:

- Variables are statically separated into transactional and non-transactional.
- Threads can access shared global variables (such as `global` in Figure 1.1 and Figure 1.2) outside transactions, but not inside them. Thread-local variables (such as `z` in Figure 1.2) can be accessed both inside and outside transactions.
- Nesting of atomic blocks is not allowed.

The thesis is organized as follows. In Chapter 2, we introduce our model definitions, define the semantics of our programming language, and define observational refinement in our setting. In Chapter 3, we define STMS, formalize TMS1, and state the key results. In Chapter 4 and Chapter 5, we prove the sufficiency and necessity results for STMS and TMS1. In Chapter 6, we relate STMS to other TM consistency conditions, such as Opacity, VWC, DU-opacity and TMS2, and prove that VWC is sufficient for a weaker notion of observational refinement. In Chapter 7, we discuss the safety of STMS. In Chapter 8, we discuss previous work on database consistency conditions and the semantics of different programming languages with atomic blocks. Chapter 9 concludes the thesis.
Chapter 2

Model

In this chapter, we will introduce our model definitions, and define the semantics of our programming language.

2.1 Programming Language Syntax

We consider a language with programs consisting of a fixed, but arbitrary, number \( m \) of threads, identified by \( \text{ThreadID} = \{1,\ldots,m\} \). Every thread \( t \in \text{ThreadID} \) has a private set of local variables \( \text{LVar}_t = \{x,y,\ldots\} \) and threads share a set of global variables \( \text{GVar} = \{g,\ldots\} \), all of type integer. We let \( \text{Var} = \text{GVar} \cup \bigcup_{t=1}^{m} \text{LVar}_t \) be the set of all program variables. Threads can also access a transactional memory, which manages a fixed collection of transactional objects \( \text{Obj} = \{o,\ldots\} \), each with a set of methods that threads can call. For simplicity, we assume that each method takes one integer parameter and returns an integer value, and that all objects have the same set of methods \( \text{Method} = \{f,\ldots\} \).

The syntax of the language is as follows:

\[
C ::= c | C; C | \text{if}(b)\ \text{then} \ C \ \text{else} \ C | \text{while}(b)\ \text{do} \ C \\
x ::= \text{atomic}\{C\} | x ::= o.f(e)
\]

\[
P ::= C_1 \parallel \ldots \parallel C_m
\]

where \( b \) and \( e \) denote Boolean and integer expressions over local variables, left unspecified. The code of threads in a program \( P \) is given by sequential commands \( C_1,\ldots,C_m \), which can include primitive commands \( c \) from a set \( \text{PComm} \), sequential compositions, conditionals, loops, atomic blocks, object method invocations and a special \text{abort} command. Primitive commands are meant to execute atomically. We do not fix their set \( \text{PComm} \), but assume that it at least includes assignments to local and global variables, a \text{skip} command that does nothing, and a special \text{fault} command, which stops the execution of the program in an error state. Thus, \text{fault} encodes illegal computations, such as division by zero.
An **atomic block** $x := \text{atomic\{C\}}$ executes $C$ as a **transaction**, which the TM can **commit** or **abort**. The TM’s decision is returned in the local variable $x$, which gets assigned distinguished values **committed** or **aborted**. We forbid nested atomic blocks and, hence, nested transactions. Inside an atomic block (and only there), the program can invoke methods on transactional objects, as in $x := o.f(e)$. Here the expression $e$ gives the value of the method parameter, and $x$ gets assigned the return value after the method terminates. The transactional system may decide to abort a transaction initiated by $x := \text{atomic\{C\}}$ not only upon reaching the end of the atomic block, but also during the execution of a method on a transactional object. Once this happens, the execution of $C$ terminates. We do not allow programs in our language to abort a transaction explicitly. A typical pattern of using the transactional system is to execute a transaction repeatedly until it commits.

We assume that a thread cannot access global variables inside atomic blocks and cannot access local variables of other threads. Thus, we accordingly restrict the expressions used in the conditions of if and while commands. To restrict accesses by primitive commands, we partition the set $\text{PComm} - \{\text{fault}\}$ into $2m$ classes: $\text{PComm} - \{\text{fault}\} = \bigsqcup_{t=1}^{m} (\text{LPcomm}_t \cup \text{GPcomm}_t)$. The intention is that commands from $\text{LPcomm}_t$ can access only the local variables of thread $t$ ($\text{LVar}_t$); commands from $\text{GPcomm}_t$ can additionally access global variables ($\text{LVar}_t \cup \text{GVar}$). We formalize these restrictions in Section 2.4. We then require a thread $t$ to use only primitive commands from $\text{LPcomm}_t \cup \text{GPcomm}_t \cup \{\text{fault}\}$ and to use only those from $\text{LPcomm}_t \cup \{\text{fault}\}$ inside atomic blocks.

We note that, whereas transactional objects are managed by the transactional system, global variables are not. Thus, threads can communicate via the transactional system inside atomic blocks, and directly via global variables outside them. In the following, we define two variants of programming language semantics differing in the treatment of local variables upon a transaction abort: in one case, they are rolled back to the values they had when the transaction started, and in the other case, left as is. Thus, in the latter case, aborted transactions can leak information to the following non-transactional code.

### 2.2 Model of Computations

We model a program computation by a **trace**, which is a finite sequence of **actions**, each describing a single computation step.

**Definition 2.1** Let ActIonId be a set of action identifiers. A **primitive action** $\chi$ has the form $(a, t, c)$, where $a \in \text{ActIonId}$, $t \in \text{ThreadID}$ and $c \in \text{PComm}$ is a primitive command. A **TM interface action** $\psi$ has one of the following forms:
<table>
<thead>
<tr>
<th>Request actions</th>
<th>Matching response actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>((a,t,txbegin))</td>
<td>((a',t,\text{OK})) \lor ((a',t,\text{aborted}))</td>
</tr>
<tr>
<td>((a,t,txcommit))</td>
<td>((a',t,\text{committed})) \lor ((a',t,\text{aborted}))</td>
</tr>
<tr>
<td>((a,t,\text{call } o.f(n)))</td>
<td>((a',t,\text{ret}(n') o.f)) \lor ((a',t,\text{aborted}))</td>
</tr>
</tbody>
</table>

where \(a, a' \in \text{ActionId}\), \(t \in \text{ThreadID}\), \(o \in \text{Obj}\), \(f \in \text{Method}\) and \(n, n' \in \mathbb{Z}\). We use \(\varphi\) to range both over primitive and TM interface actions.

TM interface actions denote the control flow of a thread \(t\) crossing the boundary between the program and the TM: request actions correspond to the control being transferred from the program to the TM, and response actions, the other way around. A \(txbegin\) action is generated upon entering an atomic block. If this successfully starts a transaction, the TM responds with an \(\text{OK}\) action. A \(txcommit\) action is generated when a transaction tries to commit upon exiting an atomic block; if this is successful, the TM responds with a \(\text{committed}\) action. Actions \(\text{call}\) and \(\text{ret}\) denote, respectively, a call to and a return from an invocation of a method \(f\) on a transactional object \(o\) and are annotated with the method parameter \(n\) or return value \(n'\). The TM may abort a transaction at any point when it is in control; this is recorded by an \(\text{aborted}\) response action. We do not allow programs in our language to abort a transaction explicitly.

In addition to interface actions, we have actions of the form \((a,t,c)\), which denote the execution of a primitive command \(c\) by thread \(t\). To denote the evaluation of conditions in if and while statements, we assume that the sets \(\text{LPcomm}_t\) contain special primitive commands \(\text{assume}(b)\), where \(b\) is a Boolean expression over local variables of thread \(t\), defining the condition. We state their semantics formally in Section 2.4.2; informally, \(\text{assume}(b)\) does nothing if \(b\) holds in the current program state, and stops the computation otherwise. Thus, it allows the computation to proceed only if \(b\) holds. The \(\text{assume}\) commands are only used in defining the semantics of the programming language; hence, we forbid threads from using them directly.

We call a trace containing only TM interface actions a history. We use \(\tau\) to range over traces, and \(H, S\) to range over histories. We denote the set of all traces by \(\text{Traces}\), and the set of all histories by \(\text{History}\).

We denote irrelevant expressions by \(\_\) and use the following notation for traces: \(\varepsilon\) is an empty trace; \(\tau(i)\) is the \(i\)-th element of \(\tau\); \(|\tau|\) is the length of \(\tau\); \(\tau\mid_i\) is the prefix of \(\tau\) containing \(i\) actions; \(\text{history}(\tau)\) is the projection of \(\tau\) to TM interface actions; \(\tau\mid_t\) is the projection of \(\tau\) onto actions of thread \(t\); \(\tau\mid_{\sim t}\) is the projection of \(\tau\) onto actions of threads other than \(t\); \(\tau\mid_o\) is the projection of \(\tau\) onto call and ret actions on object \(o\); \(\tau_1\tau_2\) is the concatenation of \(\tau_1\) and \(\tau_2\); \(\tau_1\mid_{\sim\tau_2}\) is the subsequence of \(\tau_1\) comprised of all the actions in \(\tau_1\) which are not in \(\tau_2\). We say that an action \(\varphi\) is in \(\tau\), denoted by \(\varphi \in \tau\), if \(\tau = \varphi\varphi\). \(\psi \prec_\tau \psi'\) holds if \(\psi\) appears before \(\psi'\) in \(\tau\), i.e., \(\psi \prec_\tau \psi' \iff \tau = \psi\psi'\).
Programs in our programming language do not generate arbitrary traces, but only those satisfying certain conditions, summarized in the following definition.

**Definition 2.2** A trace $\tau$ is **well-formed** if:

(i) every action in $\tau$ has a unique identifier: if $\tau = \tau_1(a_1, \ldots) \tau_2(a_2, \ldots) \tau_3$ then $a_1 \neq a_2$;

(ii) no action follows a **fault**: if $\tau = \tau' \varphi$ then $\tau'$ does not contain a **fault** action;

(iii) request and response actions are properly matched: for every thread $t$, $\text{history}(\tau)|_t$ consists of alternating request and corresponding response actions, starting with a request action;

(iv) for every thread $t$, $\tau|_t$ does not contain a request action immediately followed by a primitive action;

(v) actions denoting the beginning and end of transactions are properly matched: for every thread $t$, in the projection of $\tau|_t$ to $\text{txbegin}$, committed and aborted actions, $\text{txbegin}$ alternates with committed or aborted, starting from $\text{txbegin}$;

(vi) call and ret actions occur only inside transactions: for every thread $t$, if $\tau|_t = \tau_1 \psi \tau_2$ for a call or ret action $\psi$, then $\tau_1 = \tau'_1(\ldots, \text{txbegin}) \tau''_1$ for some $\tau'_1$ and $\tau''_1$ such that $\tau''_1$ does not contain committed or aborted actions;

(vii) commands in $\tau$ do not access local variables of other threads: if $(_c, t, c) \in \tau$ then $c \in \text{LPcomm}_t \cup \text{GPcomm}_t \cup \{\text{fault}\}$;

(viii) commands in $\tau$ do not access global variables inside a transaction: if $\tau = \tau_1(\ldots, t, c) \tau_2$ for $c \in \text{GPcomm}_t$, then it is not the case that $\tau_1 = \tau'_1(\ldots, t, \text{txbegin}) \tau''_1$, where $\tau''_1$ does not contain committed or aborted actions.

A history is well-formed if it is well-formed as a trace. We denote the set of all well-formed traces by $\text{WfTraces}$, and the set of all well-formed histories by $\text{WfHistory}$.

**Definition 2.3** Two traces $\tau$ and $\tau'$ are **equivalent up to action identifiers**, denoted $\tau \equiv \tau'$, if $|\tau| = |\tau'|$ and for every $i = 1..|\tau|$, actions $\tau(i)$ and $\tau'(i)$ may differ only in their action identifiers.

**Definition 2.4** A transaction $T$ is a nonempty well-formed trace such that:

- it contains actions by the same thread,

- it begins with a $\text{txbegin}$ action, and

- only its last action can be a committed or an aborted action.

The **status** of a transaction $T$ is:
– **committed** if it ends with a **committed** action,
– **aborted** if it ends with an **aborted** action,
– **commit-pending** (abbreviated **copending**) if it ends with a **txcommit** action, and
– **live**, in all other cases.

A transaction $T$ is **completed** if it is either **committed** or **aborted**, **visible** if it contains a **txcommit** action (i.e. **commit-pending**, **committed** or **aborted**). An **aborted** transaction is **tm-aborted** if it contains a **txcommit** action, and **force-aborted** otherwise.

A transaction $T$ is in a trace $\tau$, written $T \in \tau$, if $\tau|_t = \tau_1 T \tau_2$ for some $t$, $\tau_1$ and $\tau_2$, where either $T$ is completed or $\tau_2$ is empty. We denote the set of all transactions in $\tau$ by $tx(\tau)$ and use self-explanatory notation for various subsets of transactions: $committed(\tau)$, aborted$(\tau)$, copending$(\tau)$, live$(\tau)$, completed$(\tau)$, visible$(\tau)$, tmaborted$(\tau)$, forceaborted$(\tau)$.

For $\varphi \in \tau$, the **transaction of** $\varphi$ **in** $\tau$, denoted $txof(\varphi, \tau)$, is the subsequence of $\tau$ comprised of all actions that are in the same transaction in $\tau$ as $\varphi$ (undefined if $\varphi$ does not belong to a transaction). An action $\varphi \in \tau$ is **transactional** if $\varphi \in T$ for some transaction $T \in \tau$, and **non-transactional** otherwise. We denote by $\tau|_{\text{trans}}$ and $\tau|_{\neg\text{trans}}$ the projections of $\tau$ to transactional and non-transactional actions. We denote by $\tau|_{\neg\text{abortedtx}}$ the projection of $\tau$ excluding aborted transactions. We denote by $\tau|_{\neg\text{live}}$ the projection of $\tau$ to actions by transactions that are not live, and $\tau|_{\text{com}}$ the projection to actions by committed transactions.

We specify the behavior of a TM implementation by the set of possible interactions it can have with programs: a **transactional memory** $\mathcal{T}$ is a set of well-formed histories that is prefix-closed and closed under renaming action identifiers, i.e., if a history $H$ is in $\mathcal{T}$ then any history equivalent to $H$ up to action identifiers is also in $\mathcal{T}$.

We require prefix-closure to take into account incomplete executions.

### 2.3 An Atomic Transactional Memory

We define the correctness of a TM implementation by relating its history set to that of an **abstract** implementation, whose behavior it has to simulate; in this context, we call the original implementation **concrete**. The abstract implementation is typically one in which **atomic** blocks actually execute atomically and methods called by aborted transactions have no effect. We now define such an abstract transactional memory $\mathcal{T}_{\text{atomic}}$. We first introduce the necessary ingredients, starting with a special class of **non-interleaved** histories.

**Definition 2.5** A well-formed history $H$ is **complete** if all transactions in it are completed. A well-formed history $H$ is **non-interleaved** if actions by any two transactions...
do not overlap: if \( H = H_1(\ldots, t, \text{txbegin}) H_2(\ldots, t', \text{txbegin}) H_3 \), where \( H_2 \) does not contain \text{txbegin} actions, then either \( H_2 \) contains a \((\ldots, t, \text{committed})\) or a \((\ldots, t, \text{aborted})\) action, or there are no actions by thread \( t \) in \( H_3 \).

Note that a non-interleaved history does not have to be complete. In fact, the history set \( T_{\text{atomic}} \) we are about to define contains only non-interleaved histories, but some of them are incomplete. This is because a concrete TM may produce histories with incomplete transactions, and the consistency condition requires these transactions to stay incomplete in the matching history of the abstract system. To check whether an incomplete history is in \( T_{\text{atomic}} \), we first complete it, as explained below, by aborting every transaction that is live, and committing or aborting every transaction that is commit-pending.

**Definition 2.6** \( H \) is a completing history for an interface action \( \psi \), if the following holds:

- if \( \psi \in \{ (\ldots, t, \text{call o.f}(n)), (\ldots, t, \text{txbegin}) \} \), then \( H = (\ldots, t, \text{aborted}) \);
- if \( \psi \in \{ (\ldots, t, \text{ret}(n) \text{ o.f}), (\ldots, t, \text{OK}) \} \), then \( H = (\ldots, t, \text{txcommit}) (\ldots, t, \text{aborted}) \);
- if \( \psi = (\ldots, t, \text{txcommit}) \), then \( H = (\ldots, t, \text{committed}) \) or \( H = (\ldots, t, \text{aborted}) \);
- otherwise, \( H = \varepsilon \).

**Definition 2.7** A history \( H_c \) is a non-interleaved completion of a non-interleaved history \( H \), if \( H_c \) is a non-interleaved complete history with unique action identifiers that can be constructed from \( H \) by adding a completing history for the last action of every thread right after this action. A history \( H_c \) is a non-interleaved \text{cp-completion} of a history \( H \) if \( H \) is a subsequence of \( H_c \), \( \text{copending}(H_c) = \emptyset \) and whenever \( \psi \in H_c \) and \( \psi \notin H \) we have \( H_c = (\ldots, t, \text{txcommit}) \psi \) and either \( \psi = (\ldots, t, \text{committed}) \) or \( \psi = (\ldots, t, \text{aborted}) \). We denote the set of non-interleaved completions of \( H \) by \( \text{nicomplete}(H) \) and the set of non-interleaved \text{cp-completions} of \( H \) by \( \text{nicomplete}_{\text{cp}}(H) \).

Informally, \( H' \in \text{nicomplete}(H) \) completes each commit-pending transaction in \( H \) by adding a committed or aborted action at its end, and \( H' \in \text{nicomplete}_{\text{cp}}(H) \) completes each commit-pending and live transaction in \( H \) by adding a completing history at its end.

To define \( T_{\text{atomic}} \), we also need to know the intended semantics of operations on transactional objects. We describe the semantics for an object \( o \in \text{Obj} \) by fixing all sequences of actions on \( o \) that are considered correct when executed by a sequential program. More precisely, a sequential specification of an object \( o \) is a set of well-formed histories \([o]\) such that:

- \([o]\) is prefix-closed;
– \([ o ]\) is closed under renaming action identifiers;

– each \( H \in [ o ]\) consists of alternating call and ret actions on \( o\), starting from a call action, where every ret is by the same thread as the preceding call; and

– \([ o ]\) is insensitive to thread identifiers: for any \( H \in [ o ]\), changing the thread identifier in call-ret pair of adjacent actions in \( H\) yields a history in \([ o ]\).

– \([ o ]\) accepts any call action on \( o\); if whenever \( H \in [ o ]\), we also have \( H' = H\psi \in [ o ]\), where \( H'\) is a well-formed history and \( \psi\) is a call action on \( o\).

For example, \([ o ]\) for a register object \( o\) would consist of histories where each read method invocation returns the value written by the latest preceding write method invocation (or the default value if there is none).

Using sequential specifications for all objects, we now define when a complete and non-interleaved history \( H\) respects the object semantics. Let \( H(i)\) be a call or ret action on an object \( o\). We say that \( H(i)\) is legal in \( H\) if \( H'\mid_o \in [ o]\), where \( H'\) is the history obtained from \( H\) by projecting \( H\mid_i\) on all actions by committed transactions and the transaction containing \( H(i)\). A complete and non-interleaved history \( H\) is legal if all call and ret actions in \( H\) are legal. We let \( T_{\text{atomic}}\) be the set of all non-interleaved histories that can be completed to a legal history:

\[
T_{\text{atomic}} = \{ H \mid (H \text{ is non-interleaved}) \land (\exists \text{ legal } H_c \in \text{nicompleteall}(H))\}.
\]

### 2.4 Semantics of the Programming Language

In this section, we define the semantics of our programming language, i.e., the set of traces that computations of programs produce. The semantics comes in two flavors:

– **Semantics without rollback.** When a transaction is aborted, local variables are not rolled back to their initial values, and the values written to them by the transaction can thus be observed by the following non-transactional code.

– **Semantics with rollback.** When a transaction is aborted, local variables are rolled back to the values they had at its start, and the values written to them by the transaction cannot be observed by the following non-transactional code.

The two semantics are needed to characterize STMS and TMS1, respectively. In the following, we subscript definitions used to define the semantics by RB and noRB, and use “\( X\)” to range over both.

A **state** of a program records the values of all its variables: \( s \in \text{State} = \text{Var} \rightarrow \mathbb{Z}\).

The semantics of a program \( P = C_1 \parallel \ldots \parallel C_m\) is given by the set of well-formed traces \([ P, T ]X(s) \subseteq \text{WfTraces}\) it produces when executed with a TM \( T\) from an initial state \( s\). We define this set in two stages. First, we define the set of traces \([ P ]X(s) \subseteq\)
WfTraces that a program can produce when executed from \( s \) with the behavior of the TM unrestricted, i.e., considering all possible values the TM can return to object method invocations and allowing transactions to commit or abort arbitrarily. We then compute the set of traces produced by \( P \) when executed with a TM \( T \) by selecting those traces that interact with the TM in a way consistent with \( T \):

\[
[P, T]_X(s) = \{ \tau \mid \tau \in [P]_X(s) \land \text{history}(\tau) \in T \}.
\] (2.1)

The set \([P]_X(s)\) is itself defined in two stages. First, we define a set \( \text{Tr}(P) \subseteq \text{WfTraces} \) of traces that resolves all issues regarding sequential control flow and interleaving. Intuitively, if one thinks of each thread \( C_t \) in \( P \) as a control-flow graph, then \( \text{Tr}(P) \) contains all possible interleavings of paths in the graphs of \( C_t, t \in \text{ThreadID} \), starting from their initial nodes. The set \( \text{Tr}(P) \) is a superset of all the traces that can actually be executed: e.g., if a thread executes the command

\[
x := 1; \text{if}(x = 1) y := 1 \text{ else } y := 2
\] (2.2)

where \( x, y \) are local variables, then \( \text{Tr}(P) \) will contain a trace where \( y := 2 \) is executed instead of \( y := 1 \). To filter out such nonsensical traces, we evaluate every trace to determine whether it is valid, i.e., whether its control flow is consistent with the effect of its actions on program variables. This is formalized by the function \( \text{eval}_X : \text{State} \times \text{WfTraces} \to P(\text{State}) \cup \{ \text{\#} \} \) that, given an initial state and a well-formed trace, produces the set of states resulting from executing the actions in the trace, an empty set if the trace is invalid, or a special error state \( \text{\#} \) if the trace contains a fault action. Thus,

\[
[P]_X(s) = \{ \tau \in \text{Tr}(P) \mid \text{eval}_X(s, \tau) \neq \emptyset \}.
\] (2.3)

It is the definition of the \( \text{eval}_X \) function that mandates that local variables be rolled back upon a transaction abort (when \( X = \text{RB} \)) or not (when \( X = \text{noRB} \)).

The rest of this section formally defines the trace set \( \text{Tr}(P) \) (Section 2.4.1) and the evaluation function \( \text{eval}_X \) (Section 2.4.3).

### 2.4.1 The Trace Set \( \text{Tr}(P) \)

The function \( \text{Tr}'(\cdot) \) in Figure 2.1 maps commands and programs to traces they may produce. Technically, \( \text{Tr}'(\cdot) \) may contain traces that are not well-formed, e.g., because they contain duplicate identifiers or continue beyond a fault command. This is resolved by intersecting the set \( \text{Tr}'(P) \) with the set of all well-formed traces to define \( \text{Tr}(P) \). We also take its prefix-closure to account for incomplete program computations, as well as those in which the scheduler preempts a thread forever.

\( \text{Tr}'(P) \) is defined as the set of all the interleavings of traces produced by the threads constituting \( P \). \( \text{Tr}'(c) \) returns a singleton set with the action corresponding to the
primitive command \( c \) (recall that primitive commands execute atomically). \( \text{Tr}'(C_1; C_2) \) concatenates all possible traces corresponding to \( C_1 \) with those corresponding to \( C_2 \).

The set of traces of a conditional considers cases where either branch is taken. We describe the parameter \( x \) of the loop is defined using the Kleene closure operator \( \ast \) indeed evaluates to \( \text{Ok} \) the method executes successfully and where the current transaction is aborted. The following stages of the semantics definition: the former when defining \( \llbracket P \rrbracket X(s) \) (2.3) by the semantics of \textit{assume}, and the latter when defining \( \llbracket P, T \rrbracket X(s) \) (2.1) by selecting the traces from \( \llbracket P \rrbracket X(s) \) that interact with the transactional memory correctly. The trace set of \( x := \text{atomic} \{ C \} \) contains those in which \( C \) is aborted in the middle of its

\[
\begin{align*}
\text{Tr}'(e) & = \{ (\ldots, t, e) \} \\
\text{Tr}'(C_1; C_2) & = \{ \tau_1 \tau_2 \mid \tau_1 \in \text{Tr}'(C_1) \land \tau_2 \in \text{Tr}'(C_2) \} \\
\text{Tr}'(\text{if}(b) \text{ then } C_1 \text{ else } C_2) & = \{ (\ldots, t, \text{assume}(b)) \tau_1 \mid \tau_1 \in \text{Tr}'(C_1) \} \cup \\
& \{ (\ldots, t, \text{assume}(-b)) \tau_2 \mid \tau_2 \in \text{Tr}'(C_2) \} \\
\text{Tr}'(\text{while}(b) \text{ do } C) & = \{ (\ldots, t, \text{assume}(b)) (\text{Tr}'(C))^\ast (\ldots, t, \text{assume}(-b)) \} \\
\text{Tr}'(x := \text{atomic} \{ C \}) & = \{ (\ldots, t, \text{txbegin}) (\ldots, t, \text{aborted}) (\ldots, x := \text{aborted}) \} \cup \\
& \{ (\ldots, t, \text{txbegin}) (\ldots, t, \text{OK}) \tau (\ldots, t, \text{aborted}) (\ldots, \text{false}) \mid \\
& \tau \in \text{Tr}'(C) \land (\ldots, t, \text{aborted}) \not\in \tau \} \cup \\
& \{ (\ldots, t, \text{txbegin}) (\ldots, t, \text{OK}) \tau (\ldots, t, \text{txcommit}) (\ldots, r) (\ldots, x := r) \mid \\
& \tau \in \text{Tr}'(C) \land (\ldots, t, \text{aborted}) \not\in \tau \land (r = \text{committed} \lor r = \text{aborted}) \} \\
\text{Tr}'(x := o.f(e)) & = \{ (\ldots, t, \text{assume}(e = n)) (\ldots, t, \text{call } o.f(n)) (\ldots, t, \text{ret}(n') o.f) \mid \\
& (\ldots, x := n') \mid n, n' \in \mathbb{Z} \} \cup \\
& \{ (\ldots, t, \text{assume}(e = n)) (\ldots, t, \text{call } o.f(n)) (\ldots, t, \text{aborted}) \mid n \in \mathbb{Z} \} \\
\text{Tr}'(C_1 \parallel \ldots \parallel C_m) & = \bigcup \{ \text{interleave}(\tau_1, \ldots, \tau_m) \mid \forall t. 1 \leq t \leq m \implies \tau_t \in \text{Tr}'(C_t) \} \\
\text{Tr}(P) & = \text{prefix}(\text{Tr}'(P)) \cap \text{WfTraces}
\end{align*}
\]

Figure 2.1: The definition of \( \text{Tr}(P) \). A trace \( \tau \in \text{interleave}(\tau_1, \ldots, \tau_m) \) if and only if every action in \( \tau \) is performed by some thread \( t \in \{1, \ldots, m\} \), and \( \tau_t = \tau_t \) for every \( t \in \{1, \ldots, m\} \). The function \( \text{prefix}(\cdot) \) prefix-closes a given set of traces.
execution (at an object operation or right after it begins), those in which $C$ executes until completion and then the transaction commits or aborts, and those in which $C$ is aborted by calling the \texttt{abort} command.

2.4.2 Semantics of Primitive Commands

To define evaluation, we assume a semantics of every command $c \in \mathbb{PComm} \, \setminus \{\text{fault}\}$, given by a function $[c]$ that defines how the program state is transformed by executing $c$. As we noted before, different classes of primitive commands are supposed to access only certain subsets of variables. To ensure that this is indeed the case, we define $[c]$ as a function of only those variables that $c$ is allowed to access. Namely, the semantics of $c \in \mathbb{LPcomm}_t$ is given by

$$[c] : (\mathbb{LVar}_t \to \mathbb{Z}) \to \mathcal{P}(\mathbb{LVar}_t \to \mathbb{Z}).$$

The semantics of $c \in \mathbb{GPcomm}_t$ is given by

$$[c] : ((\mathbb{LVar}_t \cup \mathbb{GVar}) \to \mathbb{Z}) \to \mathcal{P}((\mathbb{LVar}_t \cup \mathbb{GVar}) \to \mathbb{Z}).$$

Note that $c$ can be non-deterministic.

For a valuation $q$ of variables that $c$ is allowed to access, $[c](q)$ yields the set of their valuations that can be obtained by executing $c$ from a state with variable values $q$. For example, an assignment command $x := g$ has the following semantics:

$$[x := g](q) = \{ q[g \mapsto q(g)] \}.$$

We define the semantics of \texttt{assume} commands following the informal explanation given at the beginning of this section: for example,

$$[\text{assume}(x = n)](q) = \begin{cases} \{ q \}, & \text{if } q(x) = n; \\ \emptyset, & \text{otherwise}. \end{cases} \quad (2.4)$$

Thus, when the condition in \texttt{assume} does not hold for $q$, the command stops the computation by not producing any output.

We lift functions $[c]$ to full states by keeping the variables that $c$ is not allowed to access unmodified. For example, if $c \in \mathbb{LPcomm}_t$, then

$$[c](s) = \{ s|_{\mathbb{LVar} \setminus \mathbb{LVar}_t} \cup q \mid q \in [c](s|_{\mathbb{LVar}_t}) \},$$

where $s|_V$ is the restriction of $s$ to variables in $V$. Finally, we let

$$[\text{fault}](s) = \emptyset,$$
so that the only way a program can fault is by executing the fault command.

2.4.3 Evaluation of Traces

Using the semantics of primitive commands, we first define the evaluation of a single action on a given state:

\[
\text{eval} : \text{State} \times \text{Action} \rightarrow \mathcal{P}(\text{State}) \cup \{
\}
\]

\[
\text{eval}(s, (., t, c)) = \llbracket c \rrbracket(s);
\]

\[
\text{eval}(s, \psi) = \{s\}.
\]

Note that \text{eval} does not change the state \( s \) as a result of TM interface actions, since their return values are assigned to local variables by separate actions introduced when generating \( \text{Tr}(P) \). We now lift \text{eval} to traces, depending on whether local variables are rolled back upon abort or not.

Semantics without rollback.

In this case, the values of the local variables maintain the values that were assigned to them during the execution of a transaction even if the latter aborts. The set of states resulting from evaluating trace \( \tau \) from state \( s \) is computed by the function \( \text{eval}_{\text{noRB}}(s, \tau) \), which recursively evaluates every prefix of \( \tau \):

\[
\text{eval}_{\text{noRB}} : \text{State} \times \text{WfTraces} \rightarrow \mathcal{P}(\text{State}) \cup \{
\}
\]

\[
\text{eval}_{\text{noRB}}(s, \tau) = \begin{cases} 
\{s\}, & \tau = \varepsilon; \\
\n\varepsilon, & \tau = \tau' \varphi \text{ and } \exists s'. s' \in \text{eval}_{\text{noRB}}(s, \tau') \land \text{eval}(s', \varphi) = \varepsilon; \\
\{s'' | \exists s'. s' \in \text{eval}_{\text{noRB}}(s, \tau') \land s'' \in \text{eval}(s', \varphi)\}, & \tau = \tau' \varphi \text{ and } \neg \exists s'. s' \in \text{eval}_{\text{noRB}}(s, \tau') \land \text{eval}(s', \varphi) = \varepsilon.
\end{cases}
\]

Note that traces might evaluate to a set of states, a singleton set containing \( \varepsilon \), or the empty set. In the latter case, we say that the evaluation \emph{got stuck}.

Semantics with rollback.

When local variables are rolled back upon abort, the set of states resulting from evaluating trace \( \tau \) from state \( s \) while ignoring actions inside aborted transactions to model local variable rollback. However, ignoring the contents of aborted transactions completely poses a risk that we might consider traces including sequences of actions inside aborted transactions that yield an empty set of states. To mitigate this, \( \text{eval}_{\text{RB}}(s, \tau) \) recursively evaluates every prefix of \( \tau \), thus ensuring that sequences of actions inside aborted transaction are valid. Let \( \tau|_{\neg\text{abortact}} \) be the trace obtained from \( \tau \) by removing
all actions inside aborted transactions, so that every such transaction aborts immediately. We define $\text{eval}_{RB}$ as follows:

$$\text{eval}_{RB} : \text{State} \times \text{WfTraces} \rightarrow \mathcal{P}(\text{State}) \cup \{\emptyset\};$$

$$\text{eval}_{RB}(s, \tau) = \begin{cases} \emptyset, & \tau = \tau' \land \varphi \land \text{eval}_{RB}(s, \tau') = \emptyset; \\ \text{eval}_{\text{noRB}}(s, \tau|\neg \text{abortact}), & \text{otherwise}. \end{cases}$$

The restrictions on accesses to variables by commands from $\text{LPcomm}_t$ and $\text{GPcomm}_t$ imply:

**Proposition 2.1** Assume $\chi_1 = (\ldots, t_1, c_1)$ and $\chi_2 = (\ldots, t_2, c_2)$ are actions by different threads and

$$(c_1 \in \text{LPcomm}_{t_1} \land c_2 \in \text{LPcomm}_{t_2} \uplus \text{GPcomm}_{t_2}) \lor$$

$$(c_1 \in \text{LPcomm}_{t_1} \uplus \text{GPcomm}_{t_1} \land c_2 \in \text{LPcomm}_{t_2}).$$

Then $\text{eval}_X(s, \chi_1 \chi_2) = \text{eval}_X(s, \chi_2 \chi_1)$ for any state $s$.

The above definitions allow us to define $[P]_X(s)$ as the set of those traces from $\text{Tr}(P)$ that can be evaluated from $s$ without getting stuck, as formalized by (2.3). Note that this enables the semantics of $\text{assume}$ defined by (2.4) to filter out traces that make branching decisions inconsistent with the program state. For example, consider again the program (2.2). The set $\text{Tr}(P)$ includes traces where both branches are explored. However, due to the semantics of the $\text{assume}$ actions added to the traces according to Figure 2.1, only the trace executing $y := 1$ will result in a nonempty set of final states after the evaluation and, therefore, only this trace will be included into $[P]_X(s)$.

### 2.5 Observational Refinement

Observational refinement captures the intuitive expectations of a programmer: a concrete TM observationally refines an abstract one if every user-observable behavior of a program using the former can be reproduced if the program uses the latter. This allows the programmer to reason about the behavior of a program using the intuitive semantics formalized by the abstract TM.

Now we define the notion of *observational refinement*. Informally, a concrete TM $T_C$ observationally refines an abstract TM $T_A$, if replacing $T_C$ by $T_A$ in a program leaves all its original user-observable behaviors reproducible. The formal definition depends on which aspects of program behavior we consider observable. Informally, given a trace $\tau$ of a client program, we consider observable:

(i) the sequence of actions performed outside transactions in $\tau$; and

(ii) whether a $\tau$ ends with $\text{fault}$ or not.

We are interested in observing separate program actions and the order between them. In particular, this is desirable for checking the validity of linear-time temporal
properties over program traces. We formulate the notion of observational refinement as follows.

**Definition 2.8** Well-formed traces $\tau$ and $\tau'$ are **observationally equivalent**, written $\tau \sim \tau'$, if:

- $\tau = (\omega, \omega, \text{fault}) \iff \tau' = (\omega, \omega, \text{fault})$, and
- $\tau \neq (\omega, \omega, \text{fault}) \implies \tau|_{\neg \text{trans}} = \tau'|_{\neg \text{trans}}$.

We define observational refinement relation for a programming language in which local variables do get rolled back upon an abort ($X = \text{RB}$), and for a programming language in which local variables do not get rolled back upon an abort ($X = \text{noRB}$).

**Definition 2.9** A TM $T_C$ **observationally refines** a TM $T_A$, denoted $T_C \preceq_X T_A$, if for every program $P$, state $s$ and trace $\tau \in [P, T_C]_X(s)$ there exists a trace $\tau' \in [P, T_A]_X(s)$ such that $\tau' \sim \tau$.

We say that a TM $T_C$ **observationally refines** a TM $T_A$ **under the semantics without rollback** if $T_C \preceq_{\text{noRB}} T_A$, and $T_C$ **observationally refines** $T_A$ **under the semantics with rollback** if $T_C \preceq_{\text{RB}} T_A$. 

25
Chapter 3

Consistency Conditions and Main Results

In this chapter, we define the **Strong TMS (STMS)** correctness condition and formalize **TMS1** \([11]\) in our setting. As we shall see, STMS is a stronger variant of TMS1 as it places stricter restrictions on the TM than TMS1.

A crucial building block in the definitions of STMS and TMS1 is the following notion of the **real-time order**, which captures the order between non-overlapping transactions in a history.

**Definition 3.1** Let \(\psi = (\_, t, \_)\) and \(\psi' = (\_, t', \_\) be two actions in a history \(H\). Then \(\psi\) is **before** \(\psi'\) in the real-time order in \(H\), denoted by \(\psi \prec_H \psi'\), if \(H = H_1\psi H_2\psi' H_3\) and either

(i) \(t = t'\) or

(ii) \((\_, t', \text{txbegin}) \in H_2\psi'\) and either \((\_, t, \text{committed}) \in \psi H_2\) or \((\_, t, \text{aborted}) \in \psi H_2\).

A transaction \(T\) is **before** an action \(\psi'\) in the real-time order in \(H\), denoted by \(T \prec_H \psi'\), if \(\psi \prec_H \psi'\) for every \(\psi \in T\). A transaction \(T\) is **before** a transaction \(T'\) in the real-time order in \(H\), denoted by \(T \prec_H T'\), if \(T \prec_H T'(1)\).

**Definition 3.2** A history \(S\) **preserves the real-time order of all actions** in a history \(H\), denoted by \(H \sqsubseteq_{RT} S\), if:

\[\forall \psi, \psi', (\psi \in S \iff (\psi \in H) \land (\psi \prec_H \psi' \implies \psi \prec_S \psi').\]

### 3.1 The Strong TMS Relation

The consistency conditions we define relate a concrete TM \(T_C\) and an abstract TM \(T_A\). Our intention is for STMS to be equivalent to the observational refinement between the two TMs under the semantics without rollback. For this, STMS should match histories
in $\mathcal{T}_C$ with those in $\mathcal{T}_A$, so that a trace $\tau$ of a program $P$ with the history $H \in \mathcal{T}_C$ could be transformed into an observationally equivalent trace $\tau'$ of $P$ with a matching history $S \in \mathcal{T}_A$. STMS does the matching between histories in two ways, which can respectively be used to transform $\tau$ into $\tau'$ in the cases when $\tau$ contains a fault or not (cf. Definition 2.8). We start by defining the STMS requirements that allow performing the transformation in the latter case.

Consider first a complete history $H \in \mathcal{T}_C$. In this case, STMS requires $\mathcal{T}_A$ to contain a history $S$ such that $H \sqsubseteq_{RT} S$. Thus, $S$ has to contain the same transactions as $H$. This is necessary for observational refinement: since all transactions in $H$ are completed, the non-transactional code following the corresponding atomic blocks in $P$ is aware of the return values obtained inside these transactions, even those that aborted. (Recall that we assume the semantics without rollback, and thus the values written to local variables during the execution of a transaction are not reset to their initial values if the transaction aborts.) Thus, to convert the above $\tau$ into $\tau'$ while preserving non-transactional actions, the history $S$ has to match the return values obtained inside the transactions of $H$. The relative position of actions by different transactions may in general differ in $H$ and $S$, but we require that $S$ preserve the real-time order between transactions in $H$. This is also necessary for observational refinement. As illustrated in Figure 3.1, if $T_1 \prec_H T_2$, then in between $T_1$ completing and $T_2$ starting in a trace $\tau$ with the history $H$, threads $t_1$ and $t_2$ may execute non-transactional code, and can thus communicate using global variables, such as $g$. Preserving the real-time order of $H$ in $S$ ensures that the corresponding non-transactional actions can be preserved when transforming $\tau$ into $\tau'$.

We now consider the general case of a history $H \in \mathcal{T}_C$, which requires us to deal with live and commit-pending transactions in $H$. In this case, STMS requires $\mathcal{T}_A$ to contain a history $S$ such that $H^c \sqsubseteq_{RT} S$, for some history $H^c$ that can be obtained from $H$ via the following transformation: all live transactions and any number of commit-pending transactions are discarded, and the remaining commit-pending transactions are completed with $\text{txcommit}$ actions. The intuition behind this is that, since a live transaction has not made an attempt to commit, its actions should not affect the other transactions in the history. A commit-pending transaction may or may not have effectively committed, and thus, its actions may or may not affect the other transactions (Section 2.3).

To perform the above transformations on $H$, we use the following auxiliary operations.

Let $\text{removecp}(H)$ be the set of histories obtained from $H$ by removing any number of the commit-pending transactions in $H$: we have $H' \in \text{removecp}(H)$ if and only if:

- $H'$ is a subsequence of $H$;
- $\text{tx}(H) \setminus \text{copending}(H) = \text{tx}(H') \setminus \text{copending}(H')$; and
- $\text{copending}(H') \subseteq \text{copending}(H)$.
Definition 3.3 A well-formed history $H^c$ is a \textit{cp-completion} of a history $H$ if

- $H$ is a subsequence of $H^c$,
- every action in $H^c$ but not in $H$ is either ($\_\_\_\_\_\_,\_\_\_\_\_\_$ committed) or ($\_\_\_\_\_\_,\_\_\_\_\_\_$ aborted), and
- every transaction in $H^c$ is complete.

We denote the set of cp-completions of $H$ by $\text{cpcomplete}(H)$.

Given the above definitions, the STMS relation between $T_C$ and $T_A$ requires

\[ \forall H \in T_C. \exists H' \in \text{removecp}(H). \exists H^c \in \text{cpcomplete}(H'\mid \neg \text{live}). \exists S \in T_A. H^c \subseteq RT S. \] (3.1)

We now turn to the other part of the STMS definition, which allows transforming a trace $\tau$ of $P$ using $T_C$ with a fault into a trace $\tau'$ of $P$ using $T_A$ that also has a fault. This allows us to convert a trace $\tau$ with a given live transaction into a trace $\tau'$ with the same live transaction. In particular, this means any fault in $\tau$ can be reproduced in $\tau'$. STMS allows us to pick a different history $S$ for each live transaction in $H$.

The history $S_\psi$ used to justify a response action $\psi$ includes the transaction of $\psi$ and a subset of transactions from $H$ whose actions justify the response $\psi$. Additionally, some aborted transactions from $H$ may be included into $S_\psi$ as committed: from the perspective of the transaction of $\psi$ they took effect. This creates a “virtual world” that describes the view of the transaction of $\psi$ on the TM state, which may be different from views of other transactions.

We now introduce operations needed to define this part of STMS. First, the following notion of a \textit{possible past} of a history $H = H_1\psi$ defines all sets of transactions from $H$ that can form $S_\psi$.

A transaction $T \in \text{tx}(H)$ is \textit{maximal}, if it is not followed by another transaction in the real-time order: $\neg T' \in \text{tx}(H). T \prec_H T'$. We denote the set of all maximal transactions of $H$ by $\text{maxtx}(H)$.

Definition 3.4 A well-formed history $H_\psi = H'_1\psi$ is a \textit{possible past} of a well-formed history $H = H_1\psi$, where $\psi$ is a response action that it is not a committed or aborted action, if:

(i) $H'_1$ is a subsequence of $H_1$;

(ii) $H_\psi$ is comprised of the transaction of $\psi$ and some of the visible and force-aborted transactions in $H$:

\[ \text{tx}(H_\psi) \subseteq \{\text{txof}(\psi, H)\} \cup \text{visible}(H) \cup \text{forceaborted}(H). \]

(iii) for every transaction $T \in H_\psi$, the history $H_\psi$ includes all the transactions preceding $T$ in the real-time order in $H$:

\[ \forall T \in \text{tx}(H_\psi), \forall T' \in \text{tx}(H). T' \prec_H T \implies T' \in \text{tx}(H_\psi). \]
Figure 3.1: Transactions $T_1$, $T_3$, $T_4$ and $T_5$ form one possible past of the history $H$ of the trace shown. Allowed status of transactions in $H$ is denoted as follows: committed – $C$, aborted – $A$, commit-pending – $CP$, live – $L$. The transaction $T_5$ executes only primitive actions after $\psi$ in the trace. $\chi_1$ and $\chi_2$ are non-transactional actions by threads $t_1$ and $t_2$, respectively, and $g$ and $g'$ are global variables.

(iv) A force-aborted transaction $T$ may be included into $H_\psi$ only if it is not maximal:

$$\text{forceaborted}(H_\psi) \cap \text{maxtx}(H_\psi) = \emptyset.$$  

We denote the set of possible pasts of $H$ by $\text{STMSpast}(H)$.

Note that every history $H_\psi \in \text{STMSpast}(H_\psi)$ includes the (live) transaction $\text{txof}(\psi, H)$ of $\psi$ in $H$ but no other live transaction.

We explain the definition using the history $H$ of the trace shown in Figure 3.1: one of its possible pasts $H_\psi$ consists of the transactions $T_1$, $T_3$, $T_4$ and $T_5$. According to (ii), the transaction of $\psi$ ($T_5$ in Figure 3.1) is always included into any possible past, and live transactions are excluded: since they have not made an attempt to commit, they should not have an effect on $\psi$. We are allowed to select which other transactions to include into $S_\psi$ subject to (iii): if we include a transaction $T$ then, we also have to include all transactions preceding it in the real-time order. For example, since $T_4$ and $T_5$ are included in $H_\psi$, so are $T_1$ and $T_3$. This condition is necessary for STMS to imply observational refinement. For example, $T_3$ has to be included into $H_\psi$ because, in a program producing $H$, in between $T_3$ aborting and $T_5$ starting, thread $t_2$ could have communicated to thread $t_3$ some facts about the behavior of $T_3$, e.g., using a global variable $g$, as illustrated in Figure 3.1. When executing $\psi$, the code in $T_5$ may thus expect that $T_3$ did not take effect; hence, the result of $\psi$ has to reflect this, so that the code behavior is preserved when replacing the concrete TM by an abstract one in observational refinement. The view on the TM behavior of $T_5$ has to be consistent with that of $T_3$.

To construct the desired $S_\psi$ from $H_\psi$, we change the status of all maximal aborted transactions in $H_\psi$ to committed, as as formalized in Definition 3.7: the transaction of $\psi$ is allowed to observe the effect of these transactions on the TM state. Hence, condition (iv) allows including force-aborted transactions only when they are not maximal, i.e., only when their inclusion is forced by condition (iii). Since these transactions have not made an attempt to commit, they should not have an effect on $\psi$. 

30
We now introduce the operation necessary to change the status of aborted transactions into committed.

**Definition 3.5** We let \( \text{maxcom}(H) \) denote the history obtained from \( H \) by making all maximal aborted transactions in \( H \) committed: \( |\text{maxcom}(H)| = |H| \) and

\[
\text{maxcom}(H)(i) = \begin{cases} 
(a,t,\text{committed}) & \text{if } (H(i) = (a,t,\text{aborted}) \land \text{txof}(H(i), H) \in \text{maxtx}(H)) \\
H(i) & \text{else}
\end{cases}
\]

**Definition 3.6** A well-formed history \( H^c \) is a commit justification of a well-formed history \( H_\psi \), where \( \psi \) is a response action and \( \text{txof}(\psi, H) \) is live, if

- \( H \) is a subsequence of \( H^c \),
- every action in \( H^c \) but not in \( H \) is \((-, -, \text{committed})\),
- every transaction in \( H^c \) except that of \( \psi \) is complete, and
- \( \text{txof}(\psi, H^c) = \text{txof}(\psi, H) \).

We denote the set of commit justifications of \( H_\psi \) by \( \text{justify}(H_\psi) \).

**Definition 3.7** A well-formed history \( H^c_\psi \) is a completed possible past of a well-formed history \( H = H_1^\psi \), if \( H^c_\psi \) is a commit justification of a history obtained from a possible past \( H_1^\psi \) of \( H \) by replacing the aborted actions of all maximal aborted transactions by committed actions. The set of completed possible pasts of \( H \) is denoted \( \text{cSTMSpast}(H) \):

\[
\text{cSTMSpast}(H_1^\psi) = \{ H^c_\psi \mid \exists H_1', H_1''^\psi. H_1'^\psi \in \text{STMSpast}(H_1^\psi) \land H_1''^\psi = \text{maxcom}(H_1') \land H_\psi^c \in \text{justify}(H_1''^\psi) \}
\]

For example, one completed possible past of the history in Figure 3.1 consists of the transactions \( T_1, T_3, T_4 \) and \( T_5 \), with the status of \( T_4 \) changed to committed if it was previously aborted or commit-pending. Since our language does not allow accessing global variables inside transactions, there is no way for the code in \( T_5 \) to find out about the status of \( T_4 \) from thread \( t_2 \), and hence, this code will not notice if the status of \( T_4 \) is changed to committed when replacing the concrete TM by an abstract one in observational refinement. Note that a history \( H \) has a commit justification only if \( H \) is of the form \( H = H_1^\psi \) where all the transactions in \( H_1^\psi \), except possibly that of \( \psi \), are commit-pending or completed. Also, \( \text{cSTMSpast}(H_1^\psi) \neq \emptyset \) only if \( \psi \) is a response action. We can now give a complete definition of STMS, strengthening (3.1).

**Definition 3.8** A history \( H \) is in the Strong TMS (STMS) relation with TM \( \mathcal{T} \), denoted by \( H \sqsubseteq_{\text{stms}} \mathcal{T} \), if:
(i) \( \exists H' \in \text{removecp}(H). \exists H^c \in \text{cpcomplete}(H'|-\text{live}). \exists S \in \mathcal{T}. H^c \sqsubseteq_{RT} S; \) and 

(ii) for every response action \( \psi \) such that it is not a committed or aborted action and \( H = H_1 \psi H_2 \), we have \( \exists H^c_\psi \in \text{cTMSpast}(H_1 \psi). \exists S_\psi \in \mathcal{T}. H^c_\psi \sqsubseteq_{RT} S_\psi. \)

A TM \( \mathcal{T}_C \) is in the Strong TMS (STMS) relation with a TM \( \mathcal{T}_A \), denoted by \( \mathcal{T}_C \sqsubseteq_{\text{stms}} \mathcal{T}_A \), if

\[
\forall H \in \mathcal{T}_C. H \sqsubseteq_{\text{stms}} \mathcal{T}_A.
\]

We formulate STMS so that it is not restricted to a particular abstract TM \( \mathcal{T}_A \), similarly to other work using observational refinement to study consistency conditions [12, 13]. Most TM consistency conditions are formulated with \( \mathcal{T}_{\text{atomic}} \) (defined in Chapter 2) as the abstract TM. An exception is the consistency condition last-use opacity [37], based on specific TM implementations, whose formulation requires an abstract TM which is not as restrictive as \( \mathcal{T}_{\text{atomic}} \). This generality, not allowed by the other TM consistency conditions, has two benefits. First, our formulation can be used to compare two TM implementations, e.g., an optimized and an unoptimized one. Second, dealing with the general definition forces us to explicitly state the closure properties required from the abstract TM, rather than having them follow implicitly from its atomic behavior.

### 3.2 The TMS1 Relation

TMS1 [11] was originally formulated in an operational manner, using I/O automata; here we present a more abstract definition appropriate for our goals. TMS1 is a weaker consistency condition than STMS. Similar to STMS, TMS1 requires to justify each response action inside a transaction in a history by an abstract history. However, unlike STMS, it requires to justify the behavior of only the committed transactions by a single history of the abstract TM.

**Definition 3.9** A completed visible possible past of a well-formed history \( H \) is a history obtained from a completed possible past \( H' \) of \( H \) by removing all the aborted transactions in \( H' \). The set of completed visible possible pasts of \( H \) is denoted \( \text{cTMSpast}(H) \):

\[
\text{cTMSpast}(H) = \{(H'|-\text{abortedtx}) \mid H' \in \text{cSTMSpast}(H)\}.
\]

**Definition 3.10** A history \( H \) is in the TMS1 relation with TM \( \mathcal{T} \), denoted by \( H \sqsubseteq_{\text{tms1}} \mathcal{T} \), if:

(i) \( \exists H' \in \text{removecp}(H). \exists H^c \in \text{cpcomplete}(H'|-\text{live}). \exists S \in \mathcal{T}. H^c \sqsubseteq_{\text{RT}} S; \) and 

(ii) for every response action \( \psi \) such that it is not a committed or aborted action and \( H = H_1 \psi H_2 \), we have \( \exists H^c_\psi \in \text{cTMSpast}(H_1 \psi). \exists S_\psi \in \mathcal{T}. H^c_\psi \sqsubseteq_{\text{RT}} S_\psi. \)
A TM \( T_C \) is in the TMS1 relation with a TM \( T_A \), denoted by \( T_C \preceq_{\text{tms1}} T_A \), if

\[
\forall H \in T_C. H \preceq_{\text{tms1}} T_A.
\]

As clear from Definition 3.10, STMS and TMS1 relation differ in the way they treat aborted transactions. For example, one completed possible past of the history in Figure 3.1 consists of the transactions \( T_1, T_3, T_4 \) and \( T_5 \). However, if either of \( T_4 \) or \( T_5 \) are included in a completed visible possible past of the history, then \( T_3 \) cannot be included.

### 3.3 Main Results

The main results of this thesis are that the Strong TMS relation is equivalent to observational refinement under the semantics without rollback and the TMS1 relation is equivalent to observational refinement under the semantics with rollback. These results hold for abstract TMs that enjoy certain natural closure properties, whose formulation relies on the following notions.

A history \( H_a \) is an immediate abort extension of a history \( H \) if \( H \) is a subsequence of \( H_a \), and whenever \( \psi \in H_a \) and \( \psi \notin H \) we have:

(i) \( \psi = (\_, \_, \text{txbegin}) \) or \( \psi = (\_, \_, \text{aborted}) \);

(ii) if \( \psi = (\_, t, \text{txbegin}) \) then \( H_a = H'_a \psi_(\_, t, \text{aborted})_\_ \), where \( H'_a \in \{\varepsilon, (\_, \_, \text{committed}), (\_, \_, \text{aborted})\} \); and

(iii) if \( \psi = (\_, \_, \text{aborted}) \) then there exists \( \psi' \notin H \) such that \( H_a = \psi' \psi_\_ \).

We denote by \( \text{addab}(H) \) the set of all immediate abort extensions of \( H \). Informally, a history \( H_a \in \text{addab}(H) \) is an extension of \( H \) with transactions that abort immediately after their invocation. Note that the added transactions are placed either right before other transactions begin or right after they complete.

The required closure properties are formulated as follows:

**CLP1** A TM \( T \) is closed under immediate aborts if whenever \( H \in T \) and \( \text{aborted}(H) = \emptyset \), we also have \( H' \in T \) for any well-formed history \( H' \in \text{addab}(H) \).

**CLP2** A TM \( T \) is closed under removing live and aborted transactions if whenever \( H \in T \), we also have \( H' \in T \) for any history \( H' \) which is a subsequence of \( H \) such that \( \text{committed}(H') = \text{committed}(H) \), \( \text{copending}(H') = \text{copending}(H) \), \( \text{live}(H') \subseteq \text{live}(H) \) and \( \text{aborted}(H') \subseteq \text{aborted}(H) \).

**CLP3** A TM \( T \) is closed under completing commit-pending transactions if whenever \( H \in T \), we have \( \text{cpcomplete}(H) \cap T \neq \emptyset \).
These properties are satisfied by the expected TM specification $T_{atomic}$ defined in Section 2.3. $T_{atomic}$ satisfies CLP1 as the added transactions are placed either right before other transactions begin or right after they complete, thus keeping the history non-interleaved. Moreover, since the added transactions abort immediately, it preserves the legality of the history. $T_{atomic}$ satisfies CLP2 since removing a live or aborted transaction from a non-interleaved history keeps it non-interleaved and also preserves the legality of the history. $T_{atomic}$ satisfies CLP3 since non-interleaved completion of a history keeps it non-interleaved, and the legality is also preserved as we are not removing any committed transaction.

We say that a TM $T_C$ is in the real-time relation with a TM $T_A$, denoted by $T_C \sqsubseteq_{RT} T_A$, if for every history $H \in T_C$, there exists a history $S \in T_A$ such that $H \sqsubseteq_{RT} S$.

The following theorem states that the real-time relation implies observational refinement.

**Theorem 3.1** $T_C \sqsubseteq_{RT} T_A \implies T_C \preceq_X T_A$.

The next theorem states that the Strong TMS relation implies observational refinement under the semantics without rollback.

**Theorem 3.2** Let $T_C$ and $T_A$ be transactional memories.

(i) $T_C \sqsubseteq_{stms} T_A \implies T_C \preceq_{noRB} T_A$.

(ii) If $T_A$ satisfies CLP2 and CLP3, then $T_C \preceq_{noRB} T_A \implies T_C \sqsubseteq_{stms} T_A$.

The next theorem portrays the connection between the TMS1 relation and observational refinement under the semantics with rollback.

**Theorem 3.3** Let $T_C$ and $T_A$ be transactional memories.

(i) If $T_A$ satisfies CLP1, then $T_C \sqsubseteq_{tms1} T_A \implies T_C \preceq_{RB} T_A$.

(ii) If $T_A$ satisfies CLP2 and CLP3, then $T_C \preceq_{RB} T_A \implies T_C \sqsubseteq_{tms1} T_A$. 
Chapter 4

Proofs of Sufficiency for Observational Refinement

4.1 Sufficiency of the Real-Time Relation

Let us fix a program $P$ and a state $s$. According to Definition 2.8, to prove Theorems 3.2 and 3.3, we need to transform a trace $\tau \in [P]_X(s)$ with a history $H \in T_C$ into an observationally equivalent trace $\tau' \in [P]_X(s)$ with a history $S \in T_A$. The histories $H$ and $S$ may differ in two ways, corresponding to the two items in Definitions 3.8 and 3.10. We thus need to perform two kinds of transformations to go from $\tau$ to $\tau'$. The following lemma is used in the transformation corresponding to part (i) of Definitions 3.8 and 3.10, which matches histories using the real-time order relation.

We say that traces $\tau$ and $\tau'$ are strongly equivalent, denoted $\tau \approx \tau'$, if

$$\tau|_{\bot \text{trans}} = \tau'|_{\bot \text{trans}} \land \forall t \in \text{ThreadID}. \tau|_t = \tau'|_t.$$  

Lemma 4.1 (Rearrangement)

$$\forall H, S \in \text{WfHistory}. H \sqsubseteq_{RT} S \implies \exists \forall \tau_H \in \text{WfTraces}. \text{history}(\tau_H) = H \implies \exists \tau_S \in \text{WfTraces}. \text{history}(\tau_S) = S \land \tau_H \approx \tau_S.$$  

**Proof:** Consider $H, S \in \text{WfHistory}$ and $\tau_H \in \text{WfTraces}$ such that $H \sqsubseteq_{RT} S$ and $\text{history}(\tau_H) = H$. Note that $|H| = |S|$. To obtain the desired trace $\tau_S$, we inductively construct a sequence of traces $\tau^i \in \text{WfTraces}$, $i = 0..|S|$, with histories $H^i = \text{history}(\tau^i) \in \text{WfHistory}$ such that

$$H^i|_{i} = S|_{i}; \quad H^i \sqsubseteq_{RT} S; \quad \tau_H \approx \tau^i.$$  

(4.1)

We then let $\tau_S = \tau^{|S|}$, so that $\tau_H \approx \tau^{|S|}$ and

$$\text{history}(\tau^{|S|}) = H^{|S|} = H^{|S|}|_{|S|} = S|_{|S|} = S,$$  

We then let $\tau_S = \tau^{|S|}$, so that $\tau_H \approx \tau^{|S|}$ and

$$\text{history}(\tau^{|S|}) = H^{|S|} = H^{|S|}|_{|S|} = S|_{|S|} = S.$$  

35
as required. Note that the condition $H^i \sqsubseteq_{RT} S$ in (4.1) is not used to establish the required properties of $\tau S$; we add it so that the induction goes through.

We start the construction of the sequence of traces $\tau^i$ with $\tau^0 = \tau_H$, so that $H^0 = H$ and all the requirements in (4.1) hold trivially. Assume a trace $\tau^i \in \text{WfTraces}$ satisfying (4.1) has been constructed. We show that there is a history $H^{i+1} \in \text{WfHistory}$ and a trace $\tau^{i+1} \in \text{WfTraces}$ such that

$$\text{history}(\tau^{i+1}) = H^{i+1}; \quad H^{i+1}|_{i+1} = S|_{i+1}; \quad H^{i+1} \sqsubseteq_{RT} S; \quad \tau^i \approx \tau^{i+1}.$$ 

Let $S = S_1\psi S_2$, where $|S_1| = i$. By (4.1), $\text{history}(\tau^i)|_i = H^i|_i = S|_i = S_1$. Thus, for some traces $\tau_1$ and $\tau_2$, we have $\tau^i = \tau_1\tau_2$, where $\tau_1$ is the minimal prefix of $\tau^i$ such that $\text{history}(\tau_1) = S_1$; see Figure 4.1(a). We also have $H^i \sqsubseteq_{RT} S$, and, hence, $S$ is a permutation of $H^i$ preserving the real-time order (Definition 3.2). Since $\text{history}(\tau_1) = S_1$ and $\text{history}(\tau^i) = S_1\psi S_2$, we have, for some traces $\tau_3$ and $\tau_4$,

$$\tau_2 = \tau_3\psi\tau_4, \quad \tau^i = \tau_1\tau_2 = \tau_1\tau_3\psi\tau_4.$$ 

Let $\psi = (\_ , t , \_ )$. We note that $\text{history}(\tau_3|_{\_}) = \varepsilon$, since $\sqsubseteq_{RT}$ preserves the order of actions by the same thread and $\text{history}(\tau_1) = S_1$. We consider two cases, depending on whether $\psi = (\_ , t , \text{txbegin})$ or not.

**Case I:** $\psi \neq (\_ , t , \text{txbegin})$. Let $\tau^{i+1} = \tau_1 (\tau_3|_{t}) \psi (\tau_3|_{\_}) \tau_4$ and $H^{i+1} = \text{history}(\tau^{i+1})$; see Figure 4.1(b). Intuitively, $\tau^{i+1}$ is obtained from $\tau^i = \tau_1\tau_3\psi\tau_4$ by moving all the actions in $\tau_3$ performed by thread $t$, together with $\psi$, to the position right after $\tau_1$.

Since $\text{history}(\tau_1) = S_1$, $\text{history}(\tau_3|_{t}) = \varepsilon$ and $|S_1| = i$, we get:

$$H^{i+1}|_{i+1} = (\text{history}(\tau_1 (\tau_3|_{t}) \psi (\tau_3|_{\_}) \tau_4))|_{i+1} = S_1\psi = S|_{i+1},$$
as required. We also have:

\[ \tau^{t+1}\mid t = (\tau_1, (\tau_3\mid t) \psi (\tau_3\mid t) \tau_4)\mid t = (\tau_1\mid t) (\tau_3\mid t) \psi (\tau_4\mid t) = (\tau_1\tau_3\psi\tau_4)| \mid t = \tau^t\mid t; \]

\[ \tau^{i+1}\mid t = (\tau_1, (\tau_3\mid t) \psi (\tau_3\mid t) \tau_4)| \mid t = (\tau_1\mid t) (\tau_3\mid t) (\tau_4\mid t) = (\tau_1\tau_3\psi\tau_4)| \mid t = \tau^i\mid t. \]

Hence, for any thread \( t' \), we have \( \tau^t|\mid t' = \tau^{i+1}|\mid t' \) and \( H^{i+1}|\mid t' = S|\mid t' \). If a committed or an aborted action precedes a txbegin action in \( H^{i+1} \), but not in \( H^i \), then the precedence also holds in \( S \). Thus, \( H^{i+1} \subseteq RT \ S \).

Since \( \psi \neq (\_t, t, \text{txbegin}) \) and \( \text{history}(\tau_3|t) = \varepsilon \), all the actions performed by \( t \) in the subtrace \( \tau_3 \) of \( \tau^i \) are transactional. Hence,

\[ \tau^{i+1}\mid \text{trans} = (\tau_1, (\tau_3\mid t) \psi (\tau_3\mid t) \tau_4)| \mid \text{trans} = (\tau_1\tau_3\psi\tau_4)| \mid \text{trans} = \tau^i| \mid \text{trans} \]

and therefore \( \tau^i \approx \tau^{i+1} \).

**Case II:** \( \psi = (\_t, t, \text{txbegin}) \). Assume that the subtrace \( \tau_3 \) of \( \tau^i \) contains a committed or aborted action \( \psi' \). Since \( \text{history}(\tau_4) = S_1 \), the action \( \psi' \) would be in \( S_2 \). This would mean that the real-time order between \( \psi' \) and \( \psi \) in \( H^i \) is not preserved in \( S \), contradicting our assumption that \( H^i \subseteq RT \ S \). Thus, for any thread \( t' \neq t \), \( \tau_3|\mid t' \) consists of some number of non-transactional actions followed by some number of transactional ones, and \( \tau_3|\mid t \) does not contain any transactional actions. Motivated by these observations, we let

\[ \tau^{i+1} = \tau_1 (\tau_3|\mid \text{trans}) \psi (\tau_3|\mid \text{trans}) \tau_4 \]

and \( H^{i+1} = \text{history}(\tau^{i+1}) \); see Figure 4.1(c). Intuitively, \( \tau^{i+1} \) is obtained from \( \tau^i = \tau_1\tau_3\psi\tau_4 \) by moving all transactional actions in \( \tau_3 \) to the position right before \( \tau_4 \).

Since \( \text{history}(\tau_1) = S_1 \), \( \text{history}(\tau_3|\mid \text{trans}) = \varepsilon \) and \( |S_1| = i \), we get:

\[ H^{i+1}|\mid i+1 = (\text{history}(\tau_1, (\tau_3|\mid \text{trans}) \psi (\tau_3|\mid \text{trans}) \tau_4))|\mid i+1 = S_1\psi = S|\mid i+1, \]

as required.

Since for every thread \( t' \neq t \), \( \tau_3|\mid t' \) consists of non-transactional actions followed by transactional ones,

\[ ((\tau_3|\mid \text{trans}) \psi (\tau_3|\mid \text{trans}))|\mid \psi = (\tau_3\psi)|\mid \psi. \]

Since \( \tau_3|\mid t \) does not contain any transactional actions, \( \tau_3|\mid t = (\tau_3|\mid \text{trans})|\mid t \) and, hence,

\[ ((\tau_3|\mid \text{trans}) \psi (\tau_3|\mid \text{trans}))|\mid t = (\tau_3\psi)|\mid t. \]

Thus, for any \( t'' \) we have

\[ \tau^{i+1}|\mid t'' = (\tau_1, (\tau_3|\mid \text{trans}) \psi (\tau_3|\mid \text{trans}) \tau_4)|\mid t'' = (\tau_1\tau_3\psi\tau_4)|\mid t'' = \tau^i|\mid t'' \]

and \( H^{i+1}|\mid t'' = H^i|\mid t'' = S|\mid t'' \). If a committed or an aborted action precedes a txbegin
action in $H^{i+1}$, then it does also in $H^i$. Hence, $H^{i+1} \subseteq_{RT} S$. Finally,

$$\tau^{i+1}|_{-\text{trans}} = (\tau_1 (\tau_3|_{-\text{trans}}) \psi (\tau_3|_{\text{trans}}) \tau_4)|_{-\text{trans}} = (\tau_1 \tau_3 \psi \tau_4)|_{-\text{trans}} = \tau^i|_{-\text{trans}}.$$ 

and hence, $\tau^i \approx \tau^{i+1}$.  

We also rely on the following lemma, which states that, if two traces are strongly equivalent, then one of the traces is valid if so is the other. The proof of the lemma relies on the restrictions on accesses to variables in Definition 2.2.

**Lemma 4.2**

$$\forall \tau_H, \tau_S \in \text{WfTraces}, \forall s. \tau_H \approx \tau_S \land \text{eval}_X(s, \tau_H) \neq \emptyset \implies \text{eval}_X(s, \tau_H) = \text{eval}_X(s, \tau_S).$$

**Proof:** The proof goes by case analysis depending on whether $\tau_H$ contains a fault action or not.

**Case I.** Let us assume first that $\tau_H$, and hence $\tau_S$, does not contain a fault action. We inductively construct a sequence of traces $\tau^i \in \text{WfTraces}$, $i = 0,|\tau_S|$ such that

$$\tau^i|_{i_i} = \tau_S|_{i_i}; \quad \tau^i \approx \tau_S; \quad \text{eval}_X(s, \tau^i) = \text{eval}_X(s, \tau_H) \neq \emptyset. \quad (4.2)$$

Then for $i = |\tau_S|$ we get $\tau^i = \tau_S$, which implies the required result.

To construct the sequence of traces $\tau^i$, we let $\tau_0 = \tau_H$, so that all the requirements in (4.2) hold trivially. Assume now that a trace $\tau^i$ satisfying (4.2) has been constructed. Let $\tau_S = \tau_1 \varphi \tau_2$, where $|\tau_1| = i$, and $\varphi = (.,t,.)$. By assumption, $\tau^i|_{i_i} = \tau_S|_{i_i}$ and $\tau^i \approx \tau_S$, so that $\tau^i|_{i+1} = \tau_S|_{i+1}$. Hence, for some traces $\tau'_2$ and $\tau''_2$, we get $\tau^i = \tau_1 \tau'_2 \varphi \tau''_2$, where $\tau'_2$ does not contain any actions by thread $t$. Let $\tau^{i+1} = \tau_1 \varphi \tau'_2 \varphi \tau''_2$; then $\tau^{i+1}|_{i+1} = \tau_S|_{i+1}$. We now show that $\tau^{i+1} \approx \tau^i$ and $\text{eval}_X(s, \tau^{i+1}) = \text{eval}_X(s, \tau^i)$, which completes the proof of this case.

First note that $\tau^i|_{t'} = \tau^{i+1}|_{t'}$ for any thread $t'$ since $\tau'_2$ does not contain any actions by thread $t$. Next, consider the case where the action $\varphi$ in $\tau^i$ is non-transactional. Since $\tau^i|_t = \tau_S|_t$, the corresponding action $\varphi$ in $\tau_S$ is also non-transactional. Assume that some action $\varphi' = (.,t',.)$ in the subtrace $\tau'_2$ of $\tau^i$ is non-transactional as well, where $t' \neq t$. Let $\varphi'$ be the $j$-th action by thread $t'$ in $\tau^i$. Since $\tau^i|_{t'} = \tau_S|_{t'}$, $\varphi'$ is also the $j$-th action by thread $t'$ in $\tau_S$ and is non-transactional in this trace. But then $\varphi'$ has to be in the subtrace $\tau_2$ of $\tau_S$ and thus follow $\varphi$, contradicting $\tau^i|_{-\text{trans}} = \tau_S|_{-\text{trans}}$. Hence, if the action $\varphi$ in $\tau^i$ is non-transactional, then the subtrace $\tau'_2$ of $\tau^i$ does not contain any non-transactional actions. This implies $\tau^{i+1}|_{-\text{trans}} = \tau^i|_{-\text{trans}}$ and thus $\tau^{i+1} \approx \tau^i$.

Since $\tau^i$ is well-formed, by Definition 2.2, for any action $(.,t',c)$ in it we have that $c \in \text{LPcomm}_t \cup \text{GPcomm}_t$ and, if $c \in \text{GPcomm}_t$, then the action is non-transactional. Given this and the properties of $\tau'_2$ established above, by applying Proposition 2.1 repeatedly we get that $\text{eval}_X(s', \tau'_2) = \text{eval}_X(s', \tau'^i_2 \varphi)$ for any state $s'$. Hence,

$$\text{eval}_X(s, \tau^{i+1}) = \text{eval}_X(s, \tau_1 \tau'_2 \varphi \tau''_2) = \text{eval}_X(s, \tau_1 \tau'_2 \varphi \tau''_2) = \text{eval}_X(s, \tau^i).$$
Case II. We now consider the case when $\tau_H = \tau'_H (\omega, t, \text{fault})$. By assumption, $\tau_H \approx \tau_S$. Hence, $\tau_H|_t = \tau_S|_t$ and, because $\tau_S$ is well-formed, $\tau_S = \tau'_S (\omega, t, \text{fault})$ and neither $\tau'_H$ nor $\tau'_S$ contains a fault. Then $\tau'_H \approx \tau'_S$. By assumption, we also have $\text{eval}_X(s, \tau_H) \neq \emptyset$ and, hence, $\text{eval}_X(s, \tau'_H) \neq \emptyset$ and $\text{eval}_X(s, \tau_H) = \text{fault}$. Applying Case I, we get $\text{eval}_X(s, \tau'_S) = \text{eval}_X(s, \tau'_H) \neq \emptyset$. By the definition of $\text{eval}_X$, this implies $\text{eval}_X(s, \tau_S) = \text{eval}_X(s, \tau_H)$.

The next lemma shows that a trace $\tau_H \in [P]_X(s)$ with a history $H$ can be transformed into a strongly equivalent trace $\tau_S$ for every history $S \in [P]_X(s)$ such that $H \subseteq \text{RT}(S)$.

**Lemma 4.3**

\[
\forall P. \forall s. \forall H, S \in \text{WfHistory}. H \subseteq \text{RT}(S) \implies (\forall \tau_H \in [P]_X(s). \text{history}(\tau_H) = H \implies \exists \tau_S \in [P]_X(s). \text{history}(\tau_S) = S \land \tau_H \approx \tau_S).
\]

**Proof:** Let $P = C_1 \parallel \ldots \parallel C_m$. Consider $s$ and $H, S \in \text{WfHistory}$ and $\tau_H \in [P]_X(s)$ such that $\text{history}(\tau_H) = H$. By Lemma 4.1, there exists $\tau_S \in \text{WfTraces}$ such that $\text{history}(\tau_S) = S$ and $\tau_H \approx \tau_S$.

Since $\tau_H \in [P]_X(s)$, for some $\tau'$ we have $\tau_H \tau' \in \text{Tr}'(P)$. This implies $(\tau_H \tau')|_t \in \text{Tr}'_t(C_i)$ for any thread $t$. Since $\tau_H \approx \tau_S$, we have $\tau_S|_t = \tau_H|_t$, and so $(\tau_S \tau')|_t \in \text{Tr}'_t(C_i)$. Then by the definition of $\text{Tr}(P)$ in Figure 2.1, we get $\tau_S \in \text{Tr}(P)$.

Since $\tau_H \in [P]_X(s)$, we also have $\text{eval}_X(s, \tau_H) \neq \emptyset$. Together with $\tau_H \approx \tau_S$, by Lemma 4.2 this implies $\text{eval}_X(s, \tau_S) \neq \emptyset$. But we have also established $\tau_S \in \text{Tr}(P)$, so that $\tau_S \in [P]_X(s)$.

Strong equivalence implies observational equivalence (Definition 2.8) and, hence, Theorem 3.1 is an easy corollary of Lemma 4.3.

4.2 The Live Transaction Insensitivity Lemma

Let us fix a program $P = C_1 \parallel \ldots \parallel C_m$ and a state $s$. As part of the sufficiency proofs, we need to transform a trace $\tau \in [P]_X(s)$ with a history $H \in \mathcal{T}_C$ into an observationally equivalent trace $\tau' \in [P]_X(s)$ with a history $S \in \mathcal{T}_A$. In Section 4.1 we proved a lemma necessary for this transformation in the case when $H$ and $S$ are related as in part (i) of Definitions 3.8 and 3.10. We now deal with the other case, when $H$ and $S$ are related as in part (ii). The main subtlety of this case lies in the fact that part (ii) allows justifying the behaviour of a live transaction in $H$ by a history $S$ that contains only a subset of transactions in $H$, with the committed/aborted status of some of these transactions changed; this is formalized by the use of $\text{cSTMSt Past}$ and $\text{cTMSPast}$ in Definition 3.8(ii) and Definition 3.10(ii). This makes it challenging to show that a fault inside a live transaction of $\tau$ can be reproduced in $\tau'$, as required by Definition 2.8(i). The following lemma describes the first and foremost step of
Lemma 4.4 (Live transaction insensitivity) Let \( \tau = \tau_1 \psi_2 \in [P]_X(s) \) be a trace such that \( \psi \) is a response action by thread \( t_0 \) that is not a committed or aborted action and \( \tau_2 \) is a sequence of primitive actions by thread \( t_0 \). For any \( H^c_\psi \in \text{cSTMSpast}(\text{history}(\tau)) \) there exists \( \tau_\psi \in [P]_X(s) \) such that \( \text{history}(\tau_\psi) = H^c_\psi \) and \( \tau_\psi | t_0 = \tau | t_0 \).

Proof: We first show how to construct \( \tau_\psi \) and then prove that it satisfies the required properties. We illustrate the idea of its construction using the trace \( \tau \) in Figure 3.1. Let \( \text{history}(\tau) = H_1 \psi \). Since \( H^c_\psi \in \text{cSTMSpast}(H) \), by Definition 3.7 there exist histories \( H'_1 \) and \( H''_1 \) such that

\[
H'_1 \psi \in \text{STMSpast}(H_1 \psi) \land H''_1 = \text{maxcom}(H'_1) \land H^c_\psi \in \text{justify}(H''_1 \psi).
\]

In the following, we use \( H^{cc} \) to denote the subsequence of \( H'_\psi \) comprised of the (committed) actions that were added as part of the commit justification of \( H'_1 \psi \), i.e., \( H^{cc} = H'_\psi | -H''_\psi \).

Recall that, for the \( \tau \) in Figure 3.1, one possible \( H'_1 \psi \) consists of the transactions \( T_1, T_3, T_4 \) and \( T_5 \). Then \( H''_1 \) is obtained from \( H'_1 \) by changing the last action of \( T_4 \) to committed if it was aborted; \( H^c_\psi \) is obtained by completing \( T_4 \) with a committed action if it was commit-pending. The trickiness of the proof comes from the fact that just mirroring these transformations on \( \tau \) may not yield a trace of the program \( P \); for example, if \( T_4 \) aborted, the code in thread \( t_2 \) following \( T_4 \) may rely on this fact, communicated to it by the TM via a local variable. Fortunately, we show that it is possible to construct the required trace by erasing certain suffixes of every thread and therefore getting rid of the actions that could be sensitive to the changes of transaction status, such as those following \( T_4 \). This erasure has to be performed carefully, since threads can communicate via global variables: for example, the value written by the assignment to \( g' \) in the code following \( T_4 \) may later be read by \( t_1 \), and, hence, when erasing the the former, the latter action has to be erased as well.

We now explain how to truncate \( \tau \) consistently. Let \( \psi^b \) be the last \( \text{txbegin} \) action in \( H'_1 \psi \); then for some traces \( \tau_1^b \) and \( \tau_2^b \) we have \( \tau = \tau_1^b \psi^b \tau_2^b \). For the trace \( \tau \) in Figure 3.1, \( \psi^b \) is the \( \text{txbegin} \) action of \( T_4 \). Our idea is, for every thread other than \( t_0 \), to erase all its actions that follow the last of its transactions included into \( H'_1 \psi \) or its last non-transactional action preceding \( \psi^b \), whichever is later.

Formally, for every thread \( t \), let \( \tau'_t \) denote the prefix of \( \tau_t \) that ends with the last TM interface action of \( t \) in \( H'_1 \psi \), or \( \varepsilon \) if no such action exists. For example, in Figure 3.1,
\(\tau^I_t\) and \(\tau^I_t\) end with the last TM interface actions of \(T_1\) and \(T_4\), respectively.

Similarly, let \(\tau^N_t\) denote the prefix of \(\tau|_t\) that ends with the last non-transactional action of \(t\) in \(\tau^b\), or \(\varepsilon\) if no such action exists. For example, in Figure 3.1, \(\tau^N_t\) and \(\tau^N_t\) respectively end with the actions \(\chi_1\) and \(\chi_2\), directly preceding \(T_2\) and \(T_4\).

Let \(\tau_{t_0} = \tau|_{t_0}\) and for each \(t \neq t_0\) let \(\tau_t\) be the longer trace of \(|\tau^N_t|\) and \(|\tau^I_t|\), i.e., \(\tau^I_t\), if \(|\tau^N_t| < |\tau^I_t|\), and \(\tau^N_t\), otherwise.

We define the truncated trace \(\tau'\) to be the subsequence of \(\tau\) such that \(\tau'|_t = \tau_t\) for each \(t\). Thus, for the \(\tau\) in Figure 3.1, in the corresponding trace \(\tau'\) the actions of \(t_1\) end with \(\chi_1\) and those of \(t_2\) with the last action of \(T_4\); note that this erases both operations on \(g\'). To construct \(\tau_\psi\) from \(\tau'\), we mirror the transformations of \(H'_t\) into \(H''_t\) and \(H_c\).

Let \(\tau''\) be defined by \(|\tau''| = |\tau'|\) and

\[
\tau''(i) = (\text{if } (\tau'(i) = (a, t, \text{ aborted})) \land \text{txof}(\tau'(i), \text{history}(\tau')) \in \text{maxtx}(\text{history}(\tau')))
\text{ then } (a, t, \text{ committed}) \text{ else } \tau'(i).
\]

Let \(\tau_\psi\) be an interweaving of \(\tau''\) and \(H^{cc}\) such that the order of the indices of the actions in \(\tau_\psi\) is same as in \(H_c\). Formally, we pick a trace \(\tau_\psi\) such that the following holds:

- \(|\tau_\psi| = |\tau''| + |H^{cc}|,\)
- \(|\tau_\psi| - |\tau''| = H^{cc},\)
- \(|\tau_\psi| - H^{cc} = \tau'',\) and
- for any action identifier \(a\) and \(a'\), it holds that

\[
(a, \_ \_ \_) < H_c (a', \_ \_ \_) \implies (a, \_ \_ \_) < \tau_\psi (a', \_ \_ \_).
\]

Claim I: \(\tau_\psi\) is well-formed. Most of the properties required by Definition 2.2 for well-formedness are inherited from the well-formedness of \(\tau\). The latter is a trace of a program, and thus well-formed by definition (see Figure 2.1). \(\tau'\) is a subsequence of \(\tau\) such that for every \(t\), \(\tau'|_t\) is a prefix of \(\tau|_t\). It is easy to verify that such a subsequence of a well-formed trace is also well-formed. \(\tau''\) is obtained by replacing \text{aborted} actions with \text{committed} actions, while preserving the action and thread identifier. This does not affect well-formedness as Definition 2.2 allow both \text{aborted} and \text{committed} actions to complete a transaction.

Note that, by construction, the actions in \(\tau'\) are a subset of the actions in \(\tau\), and that the transformation of \(\tau'\) into \(\tau''\) did not change the identifiers of the actions in the traces, i.e.,

\[
\{a \mid (a, \_ \_ \_) \in \tau\} = \{a \mid (a, \_ \_ \_) \in \tau'\} \land \{a \mid (a, \_ \_ \_) \in \tau'\} \subseteq \{a \mid (a, \_ \_ \_) \in \tau\}.
\]

According to our assumption, the identifiers of actions in \(H^{cc}\) are different from these
in $\tau$, and thus, by (4.4), different from the identifiers of actions in $\tau'':$

$$\{a \mid (a, \ldots) \in \tau''\} \cap \{a \mid (a, \ldots) \in H^{cc}\} = \emptyset. \quad (4.5)$$

Moreover, as every action in $\tau$ has a unique identifier, every action in $\tau''$ has a unique identifier, which further implies that action identifiers in $\tau_\psi$ are unique. This holds because the action identifiers of $\tau''$ and $H^{cc}$ are disjoint (4.5). Since $\tau''$ is well-formed, and $\tau_\psi$, with unique action identifiers, is an interweaving of $\tau''$ and $H^{cc}$, which contains only committed actions added for commit justification of $H''_\psi$, $\tau_\psi$ is well-formed.

**Claim II:** $\tau_\psi |_{t_0} = \tau |_{t_0}$. This holds because $\tau' |_{t_0} = \tau |_{t_0}$, and any aborted transaction in $\text{history}(\tau |_{t_0})$ precedes the live transaction $\text{txof}(\psi, H_1 \psi)$ in the real-time order and, hence, is not maximal. Furthermore, $H^{cc}$ does not contain any action of $t$.

**Claim III:** $\text{history}(\tau_\psi) = H^{cc}_\psi$. By the choice of $\tau'_I$ for $t \neq t_0$, every transaction in $(H'_1 \psi) |_{t_0}$ is also in $\tau'_I$. Hence, $H'_1 \psi$ is a subsequence of $\text{history}(\tau')$. We now show that every transaction in $\text{history}(\tau')$ is in $H'_1 \psi$. We consider three cases, depending on the thread $t$ the transaction is by.

- $t = t_0$. Let $T = \text{txof}(\psi, H_1 \psi) \in H'_1 \psi$. Then by Definition 3.4(iii) we get

  $$\forall T', T' \prec_{H'_1 \psi} T \iff T' \prec T. \quad (4.6)$$

  Since any transaction $T'$ in $\text{history}(\tau' |_{t_0})$ is either $T$ or is such that $T' \prec (H_1 \psi) |_{t_0} T$, this implies the required.

- $t \neq t_0$ is such that $\tau' |_{t_0} = \tau'_I \neq \varepsilon$. Let $\psi'_I$ be the last action in $\tau'_I$ and let $T = \text{txof}(\psi'_I, H_1 \psi)$. By the choice of $\tau'_I$ we have $T \in H'_1 \psi$; then by Definition 3.4(iii) we get (4.6). Since any transaction $T'$ in $\text{history}(\tau' |_{t_0})$ is either $T$ or is such that $T' \prec (H_1 \psi) |_{t_0} T$, this implies the required.

- $t \neq t_0$ is such that $\tau'_I = \tau'_I \neq \varepsilon$. Let $\chi'_I$ be the last action in $\tau'_I$ and let $T = \text{txof}(\psi'_I, H_1 \psi) \in H'_1 \psi$. Then by Definition 3.4(iii) we get (4.6). Since $\chi'_I$ comes before $\psi'_I$ in $H_1 \psi$, any transaction $T'$ in $\tau' |_{t_0}$ is such that $T' \prec_{H_1 \psi} T$, which together with (4.6) implies the required.
We have thus shown that history(τ′) = H′ψ. Then, by the definition of τ″ and H′′ψ, we have history(τ″) = H′′ψ. By Claim I, τ″ is well-formed, and thus so is its history.

By construction of τψ, its history contains in addition to the actions in H′′ψ, only the same interface actions in H′′ψ. From (4.3) and from the totality of the <. order, we get that for every interface actions ψ′ and ψ″, it holds that

\[ ψ′ <_{H′ψ} ψ″ ⇐⇒ ψ′ <_τ ψ″. \]  (4.7)

We conclude that history(τψ) = H′′ψ.

Claim IV: τ′ ∈ ℙX(s). We start by analyzing how the trace τ|t is truncated to τt for every thread t ≠ t0. Let us assume a case split on the relative positions of τtN, τtI and ψb in τ. There are three cases, shown in Figure 4.2. Either τt = τtN (a, thread t1 in Figure 3.1) or τt = τtI (b, c). If τt = τtN, then ψb has to come in τ after the end of τtN. If τt = τtI, then either ψb comes after the end of τtI (b) or ψb is the last action of τtI or precedes this action (c, thread t2 in Figure 3.1).

By the choice of τtN, in (a) and (b) the fragment of τ in between the end of τtN and ψb can contain only those actions by t that are transactional (T2 in Figure 3.1). By the choice of τtI and ψb, in (c) the fragment of τ in between ψb and the end of τtI cannot contain a txbegin action by t; hence, by the it can contain only those actions by t that are transactional. Furthermore, these have to come from a single transaction, started either by ψb or before it (T3 in Figure 3.1). Finally, by the choice of ψb the actions of t0 following ψb are transactional and come from the transaction of ψ, also started either by ψb or before it (T5 in Figure 3.1).

Given the aforementioned analysis and Claim II, the transformation from τ to τ′ can be viewed as a sequence of two: (i) erase all actions following ψb, except those in some of transactions that were already ongoing at this time; (ii) erase some suffixes of threads containing only transactional actions. Since transactional actions do not access global variables, according to the semantics of Section 2.4, they are not affected by the actions of other threads. Furthermore, [ℙX(s)] includes incomplete program computations. This allows us to conclude that τ′ ∈ ℙX(s).

Claim V: τψ ∈ ℙX(s). Consider a transaction T by a thread t whose status is changed when switching from τ′ to τ″. Then t ≠ t0 and T must be the last transaction in τ′. We again consider cases (a-c). In case (a), we have history(T) ≺_{H′ψ} txof(ψb, H′ψ). Hence, the status of T is not changed when switching from τ′ to τ″. In cases (b) and (c), T does not have any non-transactional actions following it. Since τ″ is well-formed, T is also visible. Since, the definition of [ℙX(s)] allows committing or aborting transactions arbitrarily, we conclude that τ″ ∈ ℙX(s). For the same reason, we get τψ ∈ ℙX(s).
4.3 Proof of Theorem 3.2(i): Sufficiency of the Strong TMS Relation

Assume \( T_C \subseteq \text{stms} T_A \). Consider \( \tau \in [P, T_C]_{\text{noRB}}(s) \) and let \( H = \text{history}(\tau) \).

Assume first that \( \tau \) does not contain a fault action. Since \( T_C \subseteq \text{stms} T_A \), there exist \( H' \in \text{removecp}(H) \), \( H^c \in \text{justify}(H'|_{\text{live}}) \) and \( S \in T_A \) such that \( H^c \subseteq RT S \).

Then \( H^c \) is obtained by an interweaving of \( H'|_{\text{live}} \) and a sequence of commit actions \( H^{cc} \) for some \( H^{cc} \). Let \( \tau_0 \) be a trace obtained from \( \tau \) by discarding some committing transactions and all live transactions. Let \( \tau^c \) be the trace obtained from \( \tau \) by interweaving the actions of \( \tau_0 \) and \( H^{cc} \) in the same way \( H^c \) is obtained; thus, \( \text{history}(\tau^c) = H^c \).

It is easy to see that \( \tau^c \in [P]_{\text{noRB}}(s) \). Besides, \( \tau^c|_{\text{trans}} = \tau|_{\text{trans}} \). Since \( \tau^c \in [P]_{\text{noRB}}(s) \) and \( H^c \subseteq RT S \), by Lemma 4.3 there exists \( \tau' \in [P]_{\text{noRB}}(s) \) such that \( \text{history}(\tau') = S \) and \( \tau^c \approx \tau' \), which implies that \( \tau'|_{\text{trans}} = \tau^c|_{\text{trans}} = \tau|_{\text{trans}} \) and \( \tau' \) does not contain a fault. Hence, \( \tau' \in [P, T_A]_{\text{noRB}}(s) \) and \( \tau \sim \tau' \).

Now assume that \( \tau \) contains a fault action and let \( \tau = \tau_1 \psi \tau_2 \chi \), where \( \chi = (\_, t_0, \text{fault}) \) is transactional and \( \psi \) is the last TM interface action by thread \( t_0 \). Then \( \tau_2|t_0 \) consists of transactional actions and thus does not contain accesses to global variables. Let \( \tau_3 = \tau_1 \psi (\tau_2|t_0) \chi \); then \( \tau_3 \in [P, T_C]_{\text{noRB}}(s) \). By our assumption, \( T_C \subseteq \text{stms} T_A \). Then there exists \( H^c_\psi \in \text{cSTMSpast}(\text{history}(\tau_3)) \) and \( S \in T_A \) such that \( H^c_\psi \subseteq RT S \). By Lemma 4.4, for some \( \tau_4 \) we have \( \tau_4 \in [P]_{\text{noRB}}(s) \), \( \text{history}(\tau_4) = H^c_\psi \) and \( \tau_4|t_0 = \tau_3|t_0 = \_\chi \). By Lemma 4.3, there exists \( \tau_5 \in [P, T_A]_{\text{noRB}}(s) \) such that \( \tau_4 \approx \tau_5 \) and, hence, \( \tau_5 = \_\chi \) and \( \tau \sim \tau_5 \), as needed.

4.4 Proof of Theorem 3.3(i): Sufficiency of the TMS1 Relation

To prove Theorem 3.3(i), we cannot straightforwardly apply Lemma 4.3: Definition 3.10(i) matches only histories of committed transactions, but the histories of traces produced by a program \( P \) in Lemma 4.3 also contain aborted transactions. The following lemma allows us to use CLP1 and add empty aborted transactions into the abstract history while preserving the real-time order of all actions. Furthermore, the following proposition shows that, under the semantics with rollback, the set of traces produced by a program is closed under making all aborted transactions empty.

**Lemma 4.5** Let \( H, S \in \text{WfHistory} \) be such that \( H|_{\text{abortact}} = H \) and \( H|_{\text{abortdtx}} \subseteq RT S \). There exists \( S' \in \text{WfHistory} \) such that \( S' \in \text{addab}(S) \) and \( H \subseteq RT S' \).

**Proof:** Let \( n \) be the number of aborted transactions in \( H \). To construct the desired
$S'$, we inductively construct a sequence of histories $S_i$, $i = 0..n$ such that

$$S_i \in WfHistory; \ |abort(S_i)| = i; \ S_i \in addab(S);$$

$$\{\psi \mid \psi \in H \mid abort(S_i)\} \subseteq \{\psi \mid \psi \in S_i\} \subseteq \{\psi \mid \psi \in H\};$$

$$\forall \psi_1, \psi_2 \in S_i: \ \psi_1 \prec H \ \psi_2 \implies \psi_1 \prec S_i \ \psi_2.$$  \hspace{1cm} (4.8)

We then let $S' = S_n$, so that $H \subseteq RT S'$.

For $i = 0$, we take $S_0 = S$, and all the requirements in (4.8) hold vacuously. Assume a history $S_i$ satisfying (4.8) was constructed; we get $S_{i+1}$ from $S_i$ by the following construction. Let $H = H_1 \psi_b H_2 \psi_a H_3$, where $\psi_b = (\_, t, txbegin), \psi_a = (\_, t, aborted), \psi_b \notin S_i, H_2 | t = \varepsilon$ and

$$\neg \exists \psi'. \ \psi' = (\_, \_, txbegin) \in H_1 \land txof(\psi', H) \in aborted(H) \land \psi' \notin S_i.$$ 

That is, out of all aborted transactions in $H$ that are not in $S_i$, $\psi_b \psi_a$ is the one with the earliest $txbegin$.

If $H_1$ does not contain a committed or an aborted action, we let $S_{i+1} = \psi_b \psi_a S_i$. Then $S_{i+1} \in WfHistory$. We only need to show that for any $\psi' \in S_i$ we have $\psi' \prec H \ \psi_b \implies \psi' \prec S_{i+1} \ \psi_b$ and $\psi_a \prec H \ \psi' \implies \psi_a \prec S_{i+1} \ \psi'$. The latter holds by the construction of $S_{i+1}$. To show the former, observe that, since $H_1$ does not contain a committed or aborted action, and hence it cannot contain actions by thread $t$.

The rest of the proof deals with the case when $H_1$ contains a committed or an aborted action. Let $\psi$ be the last committed or aborted action in $S_i$ that is also in $H_1$ and let $S_i = S' \psi S''$. We then let $S_{i+1} = S' \psi_b \psi_a S''$. We again need to show that for any $\psi' \in S_i$ we have $\psi' \prec H \ \psi_b \implies \psi' \prec S_{i+1} \ \psi_b$ and $\psi_a \prec H \ \psi' \implies \psi_a \prec S_{i+1} \ \psi'$.

Assume $\psi' \prec H \ \psi_b$ for some $\psi' \in S_i$; then $\psi' \in H_1$. By the choice of $\psi_b$ and $\psi_a$, all the committed and aborted actions in $H_1$ are in $S_i$, and by the choice of $\psi$, all such actions are in $S'\psi$. Hence, if $\psi'$ is a committed or an aborted action, then $\psi' \in S'\psi$ and, hence, $\psi' \prec S_{i+1} \ \psi_b$.

If $\psi'$ is by thread $t$, then it is either a committed or an aborted action (and, hence, $\psi' \prec S_{i+1} \ \psi_b$) or it precedes such an action $\psi'' \in S_i$ by $t$ in $H_1$:

$$\psi' \prec H \ \psi''$$

Then $\psi' \prec S_{i+1} \ \psi''$ and $\psi'' \prec S_{i+1} \ \psi_b$, which implies $\psi' \prec S_{i+1} \ \psi_b$.

Now assume $\psi_a \prec H \ \psi'$ for some $\psi' \in S_i$; then $\psi' \in H_3$. If $\psi'$ is a $txbegin$ action, then $\psi \prec H \ \psi'$. Hence, $\psi \prec S_i \ \psi'$, i.e., $\psi' \in S''$, which implies $\psi_a \prec S_{i+1} \ \psi'$. If $\psi'$ is by thread $t$, then it is either a $txbegin$ action (and, hence, $\psi_a \prec S_{i+1} \ \psi'$) or it follows such an action $\psi'' \in S_i$ by thread $t$ in $H_3$: $\psi'' \prec H \ \psi'$. Then $\psi'' \prec S_{i+1} \ \psi'$ and $\psi_a \prec S_{i+1} \ \psi''$, which implies $\psi_a \prec S_{i+1} \ \psi'$.

Finally, it is easy to show that $H_1 | T = (S' \psi) | T$, implying that $S_{i+1} \in WfHistory$.  

Recall that we assume the semantics with rollback, and thus the values written to local variables during the execution of a transaction are reset to their initial values if the transaction aborts. Hence, applying $\tau \in abortact$ to a trace preserves its validity.

**Proposition 4.6** \(\forall P. \forall s. \forall \tau. \tau \in [P]_{RB}(s) \implies \tau | abortact \in [P]_{RB}(s).\)
Proof: [of Theorem 3.3(i)] Assume $T_C \subseteq_{\text{tms1}} T_A$. Consider $\tau \in [P, T_C]_{RB}(s)$ and let $H = \text{history}(\tau)$.

Assume first that $\tau$ does not contain a fault action. Since $T_C \subseteq_{\text{tms1}} T_A$, there exist $H' \in \text{removecp}(H)$, $H^c = \text{justify}(H'|_{\text{live}})$ and $S \in T_A$ such that $H^c|_{\text{com}} \subseteq_{\text{RT}} S$. Then $H^c$ is obtained by an interleaving of $H'|_{\text{live}}$ and a sequence of commit actions $H^c_{\text{cc}}$ for some $H^c_{\text{cc}}$. Let $\tau_0$ be a trace obtained from $\tau$ by discarding some commit-pending transactions and all live transactions. Let $\tau^c$ be the trace obtained from $\tau$ by interleaving the actions of $\tau_0$ and $H^c_{\text{cc}}$ in the same way $H^c$ is obtained; thus, $\text{history}(\tau^c) = H^c$. It is easy to see that $\tau^c \in [P]_{RB}(s)$. Besides, $\tau^c|_{\text{trans}} = \tau|_{\text{trans}}$. Let $\tau^{na} = \tau^c|_{\text{~abortact}}$.

By Proposition 4.6, we get $\tau^{na} \in [P]_{RB}(s)$. Since $(H^c|_{\text{~abortact}})|_{\text{~abortacted}} = H^c|_{\text{com}} \subseteq_{\text{RT}} S$, by Lemma 4.5, for some history $S'$ we have $H^c_{\text{~abortact}} \subseteq_{\text{RT}} S'$ and $S' \in \text{addab}(S)$. Since $S \in T_A$ and $T_A$ is closed under immediate aborts (CLP1), we have $S' \in T_A$. We have $\tau^{na} \in [P]_{RB}(s)$ and $\text{history}(\tau^{na}) = H^c_{\text{~abortact}}$: hence, by Lemma 4.3, there exists a trace $\tau' \in [P]_{RB}(s)$ such that $\text{history}(\tau') = S' \in T_A$ and $\tau^{na} \approx \tau'$, which implies that

$$\tau'|_{\text{~trans}} = \tau^{na}|_{\text{~trans}} = \tau^c|_{\text{~trans}} = \tau|_{\text{~trans}}$$

and $\tau'$ does not contain a fault. Hence, $\tau' \in [P, T_A]_{RB}(s)$ and $\tau \sim \tau'$.

Now assume that $\tau$ contains a fault action and let $\tau = \tau_1 \psi \tau_2 \chi$, where $\chi = (\_t_0, \_\text{fault})$ is transactional and $\psi$ is the last TM interface action by thread $t_0$. Then $\tau_2|_{t_0}$ consists of transactional actions and thus does not contain accesses to global variables. Let $\tau_3 = \tau_1 \psi(\tau_2|_{t_0}) \chi$; then $\tau_3 \in [P, T_C]_{RB}(s)$. By our assumption, $T_C \subseteq_{\text{tms1}} T_A$. Then there exists $H^c_{\psi} \in c\text{STMSpast}(\text{history}(\tau_3))$ and $S \in T_A$ such that $H^c_{\psi} \subseteq_{\text{RT}} S$. By Definition 3.9, there exists a history $H^c_{\psi} \in c\text{STMSpast}(\text{history}(\tau_3))$ such that $H^c_{\psi} = H^c_{\psi}|_{\text{~abortacted}}$. By Lemma 4.4, for some $\tau_4$ we have $\tau_4 \in [P]_{RB}(s)$, $\text{history}(\tau_4) = H^c_{\psi}$ and $\tau_4|_{t_0} = \tau_3|_{t_0} = \_\chi$. By Proposition 4.6, $\tau_4|_{\text{~abortact}} \in [P]_{RB}(s)$. Using Lemma 4.5, we get $S' \in \text{WFHistory}$ such that $\text{history}(\tau_4|_{\text{~abortact}}) \subseteq_{\text{RT}} S'$ and $S' \in \text{addab}(S)$. Since $S \in T_A$ and $T_A$ is closed under immediate aborts (CLP1), we get $S' \in T_A$. Hence, by Lemma 4.3, there exists $\tau_5 \in [P, T_A]_{RB}(s)$ such that $\tau_4 \approx \tau_5$ and, hence, $\tau_5 = \_\chi$ and $\tau \sim \tau'$.

\[ \square \]
Chapter 5

Proofs of Necessity for Observational Refinement

In this chapter, we prove that the STMS and TMS1 are necessary for obtaining observational refinement. The former for the set of observations obtained with the semantics that does not roll back the values of local variables when a transaction aborts (the noRB semantics) and the latter for those obtained with the semantics with rollback (the RB semantics). We use the same proof strategy in both cases. Thus, although below we focus our informal explanation on establishing the necessity result for Strong TMS, the explanation is also relevant for TMS1.

Given transactional memories \( T_C \) and \( T_A \) such that \( T_A \) observationally refines \( T_C \), we find for every history \( H \in T_C \), a subset of the histories in \( T_A \) which attests that the conditions in Definition 3.8 for \( H \subseteq_{stms} T_A \) to hold are met.

More specifically, recall that Definition 3.8 places conditions on the completed transactions in \( H \) and on certain of its prefixes: For the completed transactions, we find a history \( S \in T_A \) such the condition in Definition 3.8(i) holds for a certain cp-completion of \( H \), and for every (relevant) prefix \( H' \psi \) of \( H \), we find a (different) history \( S \in T_A \) which attests that the condition in Definition 3.8(ii) is met for \( H' \psi \).

Technically, we find such histories by constructing programs which can produce particular traces only when executed with a transactional memory which contains \( H \), e.g., when the program is executed with \( T_C \). The programs are constructed in such a way that when they detect that the interaction with the transactional memory is not as dictated by \( H \), they make an observable action that they would not have done otherwise. The assumption about observational refinement ensures that a history \( S_1 \) which is “close enough” to \( H \) exists in \( T_A \). The minor differences are then reconciled using the assumed closure properties: We find a completion \( H^c \) of \( H \) and a history \( S \in T_A \) such that \( H^c \subseteq_{RT} S \), where \( S \) is a history that can be obtained from \( S_1 \) by applying the assumed closure conditions.

The crux of the proof is in the encoding of the information pertaining to the *history* of every executed trace in the program’s observable behavior. Every trace induces a
particular observation, i.e., the subsequence $\tau|\neg\text{trans}$ of non-transactional actions. By assumption, $T_C \preceq_{\text{noRB}} T_A$. Thus, for every trace $\tau$ of $P$ when executed with $T_C$ there should be a trace $\tau'$ of $P$ when it runs with $T_A$ which produces the same observations, i.e., $\tau|\neg\text{trans} = \tau'|\neg\text{trans}$ if $\tau$ does not fault, and $\tau'$ faults if $\tau$ does.

In the following section, Lemma 5.1 handles the conditions pertaining to completed transactions in $H$ using the first kind of observations, and Lemma 5.2 handles the conditions pertaining to the (relevant) prefixes of $H$ using the second kind. This approach is taken because the lemma pertains to histories containing live transactions, and faults are the only means using which the internal behavior of a live transaction can be made known to the client program. Later, in Section 5.2, Lemma 5.3 and Lemma 5.4 play the roles as Lemma 5.1 and Lemma 5.3 in the proof of necessity of the TMS1 Relation.

5.1 Proof of Theorem 3.2(ii): Necessity of the Strong TMS Relation

Lemma 5.1 Let $T_C$ and $T_A$ be TMs such that $T_C \preceq_{\text{noRB}} T_A$ and $T_A$ satisfies CLP2 and CLP3. Then

$$\forall H \in T_C. \exists H' \in \text{removecp}(H). \exists H^c \in \text{cpcomplete}(H'|\neg\text{live}). \exists S \in T_A. H^c \sqsubseteq_{\text{RT}} S.$$  

Proof: The proof works in several stages. For every history $H \in T_C$, we construct a program $P_H$ and define a special initial state $s$ (Stage I). Threads in $P_H$ perform the sequence of transactions specified by $H$, record the return values they obtain from the TM, and monitor whether the real-time order between actions includes that in $H$. In particular, the program $P_H$ is such that for some trace $\tau \in \llbracket P_H \rrbracket_{\text{noRB}}(s)$ we have $\text{history} (\tau) = H$ (Stages II and III). Since $H \in T_C$ and $T_C \preceq_{\text{noRB}} T_A$, we get that there exists a trace $\tau' \in \llbracket P_H \rrbracket_{\text{noRB}}(s)$ such that $\tau \sim \tau'$ (Stage IV). Let $S_1 = \text{history} (\tau') \in T_A$. From $\tau \sim \tau'$ and the structure of $P_H$, we infer a certain correspondence between the histories $H$ and $S_1$. This allows us to transform them into the required histories $H^c$ and $S$ by ensuring that the $H^c$ is a cp-completion of $H$, $S \in T_A$, and $H^c \sqsubseteq_{\text{RT}} S$ (Stage V).

Stage I: Constructing $P_H$. Consider a history $H \in T_C$. We construct $P_H$ in Figure 5.3 using the constants defined in Figure 5.1. For ease of reference, Figure 5.2 lists the variables used in $P_H$ and their intended meaning.

Given that the history $H$ is well-formed, we construct the command $C'_H$ of every thread $t$ in $P_H$ as a sequence of atomic blocks, corresponding to $\text{txbegin}$, $\text{committed}$ and $\text{aborted}$ actions in $H|_t$. These blocks perform the sequence of method invocations determined by $\text{call}$ and $\text{ret}$ actions in $H|_t$. Namely, the command $C'_H$ is comprised of an alternating sequence of commands $GP^t_i$ and $CP^t_i$ ($i = 1..k^t$), constructed according
<table>
<thead>
<tr>
<th>Shorthand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>u</td>
<td>An arbitrary integer which does not appear in $H$.</td>
</tr>
<tr>
<td>m</td>
<td>The largest thread identifier occurring in $H$.</td>
</tr>
<tr>
<td>$k^t$</td>
<td>The number of transactions started by thread $t$ in $H$, i.e., the number of $(_-,t,\text{txbegin})$ actions in $H$.</td>
</tr>
<tr>
<td>$H^t_i$</td>
<td>$H</td>
</tr>
<tr>
<td>$c^t_i$</td>
<td>The outcome of the i-th transaction of thread $t$, i.e., $c^t_i = \text{committed}$ or $c^t_i = \text{aborted}$. If the i-th transaction is not completed, then $c^t_i = u$.</td>
</tr>
<tr>
<td>$q^t_i$</td>
<td>The number of call actions of thread $t$ in its i-th transaction, i.e., in $H^t_i$.</td>
</tr>
<tr>
<td>$(<em>-,t,\text{call } o^t</em>{i,j}.f^t_{i,j}(n^t_{i,j}))$</td>
<td>The j-th call action of thread $t$ in its i-th transaction.</td>
</tr>
<tr>
<td>$(<em>-,t,\text{ret } r^t</em>{i,j}) o^t_{i,j}.f^t_{i,j}$</td>
<td>The j-th ret action of thread $t$ in its i-th transaction. If the response to $(<em>-,t,\text{call } o^t</em>{i,j}.f^t_{i,j}(n^t_{i,j}))$ is an aborted action, we let $r^t_{i,j} = u$. Similarly, if there is no response to $(<em>-,t,\text{call } o^t</em>{i,j}.f^t_{i,j}(n^t_{i,j}))$, i.e., the transaction is live and $(<em>-,t,\text{call } o^t</em>{i,j}.f^t_{i,j}(n^t_{i,j}))$ is its last action, then we also let $r^t_{i,j} = u$.</td>
</tr>
<tr>
<td>lasttx$(t,i,t')$</td>
<td>The number of transactions of thread $t'$ in $H$ that either committed or aborted before the i-th transaction of thread $t$ started, i.e., the number of $(<em>-,t',\text{committed})$ and $(</em>-,t',\text{aborted})$ actions preceding the i-th $(_-,t,\text{txbegin})$ action in $H$.</td>
</tr>
</tbody>
</table>

Figure 5.1: Constants derived from the history $H$ that are used to construct $P_H$ (see Figure 5.3).

The command $CP^t_H$ records the return value of the j-th method invocation in the i-th transaction by thread $t$ in a dedicated variable $y^t_{i,j}$, local to $t$. The return status of the i-th transaction is recorded in a dedicated local variable $w^t_i$. 

49
<table>
<thead>
<tr>
<th>Shorthand</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y_{i,j}^t$</td>
<td>The local variable which records the return value of the $j$-th object method invocation in the $i$-th transaction of thread $t$.</td>
</tr>
<tr>
<td>$w_i^t$</td>
<td>The local variable which records whether the $i$-th transaction of thread $t$ committed or aborted.</td>
</tr>
<tr>
<td>$g_i^t$</td>
<td>The global variable written only by thread $t$ and signals that the $i$-th transaction of thread $t$ ended: $g_i^t$ is set to 1 only after $t$’s $i$-th transaction committed or aborted.</td>
</tr>
<tr>
<td>$z_{i,j}^t$</td>
<td>The local variable of thread $t$ which records whether the last transaction of thread $j$ which precedes the $i$-th transaction of $t$ in $H$ has signaled its termination before $t$’s $i$-th transaction begins.</td>
</tr>
<tr>
<td>$a_i^t$</td>
<td>The local variable of thread $t$ which records whether the $i$-th transaction of thread $t$ was (i) not immediately aborted and (ii) and got all the expected responses up to the penultimate method call.</td>
</tr>
<tr>
<td>$b_i^t$</td>
<td>The local variable of thread $t$ which records whether the $i$-th transaction of thread $t$ was forced-aborted during the last method invocation.</td>
</tr>
</tbody>
</table>

Figure 5.2: The variables used in $P_H$ and their intended meaning (see Figure 5.3).

The values of $y_{i,j}^t$ and $w_i^t$ are checked after the $i$-th transaction terminates, and, if there is a mismatch with $H|_t$, the thread $t$ enters a non-terminating loop. Technically, we make the deviation observable by the command “loop” which prevents the thread from continuing its execution by getting into an infinite loop. Intuitively, threads get into an infinite loop every time the transactional memory returns an unexpected response. Note that if the thread gets stuck in an infinite loop, it cannot produce any non-transactional (and hence observable) actions afterwards. The correspondence between actual returned values and the expected ones is checked again using the variables $a_i^t$ and $b_i^t$ after the transaction terminates. We explain the need for the double check later on.

To check whether an execution of $P_H$ complies with the real-time order in $H$, we exploit the ability of threads to communicate via global variables outside transactions.
For each transaction in $H$, we introduce a global variable $g_t^i$, which is initially 0 and is set to 1 by $t$ right after its $i$-th transaction completes. We also add a dummy variable $g_0^t$ for every thread $t$; we always execute the program from a state in which all $g_0^t$ are initialized to 1. Before starting the $i$-th transaction, each thread checks whether all transactions preceding it in the real-time order in $H$ have finished by reading the corresponding $g$-variables and records the result in a dedicated local variable $z^i_{t,t'}$. If there is a mismatch with $H$, the thread enters an infinite loop.

**Stage II: Constructing $\tau$.** We now construct a particular trace $\tau$ of $P_H$, by first constructing a trace $\tau^i$ for every sequential command $C^i_H$ and then interleaving the traces $\tau^1, \ldots, \tau^m$ in a particular way. The construction of $\tau$ in this stage is merely syntactic and concerns only possible interleaving of sequences of actions. In Stage III, we show that $\tau$ can indeed be executed.

Consider the set of traces $\text{Tr}(C^i_H)t$ of the sequential command $C^i_H$, and let $\tau^i \in \text{Tr}(C^i_H)t$ be the maximal trace which does not include actions coming from loop command such that $H|_t = \text{history}(\tau^i)$. Intuitively, $\tau^i$ is obtained when the program interacts with the object system is exactly $H$. More formally, the trace $\tau^i$ exists, since by construction of $C^i_H$ and the definition of the trace set of a sequential command (Figure 2.1), there is a trace in $\text{Tr}(C^i_H)t$ for every possible parameter and return value of object method invocations and atomic blocks in $C^i_H$; in particular, $\text{Tr}(C^i_H)t$ contains a trace where the parameters and return values of object method invocations and the return values of transactions are as in $H|_t$.

We now partition every $\tau^i$ into $|H|_t + 1$ subsequences that we later interleave to create $\tau$:

$$\tau^i = \tau^i_1 \ldots \tau^i_{|H|_t} \tau^i_{|H|_t + 1}.$$  \hspace{1cm} (5.1)

Formally, for every $i = 1..|H|_t$ there is exactly one interface action $\psi^i_t$ in $\tau^i_t$, according to the conditions in Figure 5.4.

Note that this defines $\tau^i_t$, $i = 1..|H|_t$, uniquely. Therefore, $\tau^i_{|H|_t + 1}$ is also defined uniquely as containing the rest of the actions in the trace. The fact that each subsequence $\tau^i_t$, except possibly the last one, includes exactly one interface action allows us in the following to interleave the subsequences in the order induced by $H$. The above definition also ensures that, if $\tau^i_t$ ends with $\psi^i_t = (\_ t, \text{txbegin})$, then it contains all the actions that are used to read the global signaling variables that precede $\psi^i_t$ in $\tau^i$. Furthermore, if $\tau^i_t$ contains a committed or aborted action, then it ends with the primitive (non-transactional) action that signals the end of the corresponding transaction as well as the actions that performs the check that the transaction performed the expected method calls returned (and not, e.g., got aborted earlier) and that it received the expected responses.
\[ P_H = C_1^H \parallel \ldots \parallel C_m^H \]
\[ C_H^t = GP_1^t; CP_1^t; GP_2^t; CP_2^t; \ldots; GP_k^t; CP_k^t \]
\[ GP_i^t = z_{i,1}^t := g_{lasttx(t,i,1)}^t;\]
\[ \quad \text{if}(z_{i,1}^t \neq 1) \text{ then loop}; \]
\[ \ldots \]
\[ z_{i,m}^t := g_{lasttx(t,i,m)}^t;\]
\[ \quad \text{if}(z_{i,m}^t \neq 1) \text{ then loop}; \]
\[ CP_i^t = w_i^t := \text{atomic}\{
\]
\[ \quad y_{i,1}^t := o_{i,1}^t.f_{i,1}^t(n_{i,1}^t);\]
\[ \quad \text{if}(y_{i,1}^t \neq r_{i,q_1}^t) \text{ then loop}; \]
\[ \ldots \]
\[ y_{i,q_1-1}^t := o_{i,q_1-1}^t.f_{i,q_1-1}^t(n_{i,q_1-1}^t);\]
\[ \quad \text{if}(y_{i,q_1-1}^t \neq r_{i,q_1-1}^t) \text{ then loop}; \]
\[ a_i^t = 1; \]
\[ y_{i,q_1}^t := o_{i,q_1}^t.f_{i,q_1}^t(n_{i,q_1}^t);\]
\[ \quad \text{if}(y_{i,q_1}^t \neq r_{i,q_1}^t) \text{ then loop}; \]
\[ b_i^t = 1; \]
\[ \text{lastCommand}_i^t \}
\[ \quad \text{if}(w_i^t \neq c_i^t) \text{ then loop}; \]
\[ \quad \text{if}(\neg \text{lastCheck}_i^t) \text{ then loop}; \]
\[ g_i^t := 1 \]

\[ \text{lastCommand}_i^t = \begin{cases} 
\text{loop}, & H_i^t \in \text{live}(H) \cup \text{forceaborted}(H) \\
\text{skip}, & \text{otherwise.} 
\end{cases} \]

\[ \text{lastCheck}_i^t = \begin{cases} 
b_i^t = 1, & H_i^t \in \text{committed}(H) \cup \text{tmaborted}(H) \\
b_i^t \neq 1, & H_i^t \in \text{forceaborted}(H) \text{ and } |H_i^t| = 2 \\
a_i^t = 1 \text{ and } b_i^t \neq 1, & \text{otherwise} 
\end{cases} \]

Figure 5.3: The construction of \( P_H \) for Lemma 5.1, Lemma 5.2 (the case of \( t \neq t_0 \)), Lemma 5.3 and Lemma 5.4 (the case of \( t \neq t_0 \)). For conciseness, we use an extension of the programming language with conditionals without an “else” clause. The \text{loop} command can be implemented by \text{while(true) do skip}. 

52
\[ \tau'_i = \begin{cases} 
\psi'_i, & \text{if } \psi'_i = (\_t, \text{OK}); \\
\psi'_i (\_t, y_i^0 := n), & \text{if } \psi'_i = (\_t, \text{ret}(n) \ o.f); \\
\psi'_i (\_t, g_i^0 := 1), & \text{if } \psi'_i \in \{ (\_t, \text{committed}), (\_t, \text{aborted}) \}; \\
\psi'_i, & \text{otherwise.} 
\end{cases} \]

Figure 5.4: The construction of \( \tau'_i \) for Lemma 5.1, Lemma 5.2 (the case of \( t \neq t_0 \)), Lemma 5.3 and Lemma 5.4 (the case of \( t \neq t_0 \)).

The desired trace is constructed by interleaving the subsequences of the traces \( \tau^1, \ldots, \tau^m \) according to the order induced by \( H \). Formally,

\[
\tau = \tau^{\tau^1}_j, \ldots, \tau^{\tau^{|H|}}_j \cdot \tau^{\tau_{\text{sfz}}} \quad \text{where} \quad \tau^{\tau_{\text{sfz}}} = \tau^{\tau^{|H|}+1}_j, \ldots, \tau^{\tau^{|H|}+1}_j,
\]

\[
H(i) = (\_t, j), \quad \text{and} \quad j_i = |H|_i |t|, \quad (5.2)
\]

i.e., \( t^i \) is the thread that performs the \( i \)-th action in \( H \) and \( j_i \) is the number of actions that thread \( t^i \) performed in \( H_1 \ldots H_i \). By construction, we get that history(\( \tau \)) = \( H \). The trace \( \tau^{\tau_{\text{sfz}}} \) consists of the rest of the actions made by the threads, if such actions exists. Note that by construction, \( \tau^{\tau_{\text{sfz}}} \) is comprised only of primitive actions.

**Stage III: Defining \( s \).** Since \( \tau^i \in \text{Tr}(C^i_t) t \), we have \( \tau \in \text{Tr}(P_H) \). Let \( s \) be the state where all the local variables are set to \( u \) and for all \( t \), \( g_i^t = 0 \) for \( i \neq 0 \) and \( g^0 = 1 \). Recall that \( H \) is the history of \( t \) and that \( t \) does not perform any loop command. Thus, it is possible to evaluate \( \tau \) till completion because (i) its history is in the transactional memory, and (ii) it does not get stuck (enters a non-terminating loop). It is easy to verify that eval\( \text{noRB}(s, \tau) \neq 0 \), and thus we get that \( \tau \in [P_H]_{\text{noRB}}(s, \mathcal{T}_C) \).

**Stage IV: Obtaining \( \tau' \) and analyzing the relation of its history with \( H \).**

By assumption, history(\( \tau \)) = \( H \in \mathcal{T}_C \). Since \( \tau \in [P_H]_{\text{noRB}}(s, \mathcal{T}_C) \) and \( \mathcal{T}_C \preceq_{\text{noRB}} \mathcal{T}_A \), by Definition 2.8 there exists a trace \( \tau' \in [P_H]_{\text{noRB}}(s, \mathcal{T}_A) \) such that \( \tau'|_{\text{trans}} = \tau|_{\text{trans}} \) and \( S_1 = \text{history}(\tau') \in \mathcal{T}_A \).

Consider a thread \( t \), and let \( T \) be the \( i \)-th transaction in \( \tau|_t \) and \( T' \) be the \( i \)-th transaction in \( \tau'|_t \), if such a transaction exists. These transactions arise from executing the same commands, and \( T' \) might not exist only if the commands did not execute in \( \tau' \). We now analyze the relationship between \( T \) and \( T' \). The construction in Figure 5.3 ensures the properties listed below. Informally, this is because the conditional checks are done after every object action and after every transaction, and discrepancy is exposed by taking a different path of the execution which leads to a non-terminating loop.

(i) If \( T \) is committed or tm-aborted, then \( T' \) is committed or aborted, respectively, and \( T \equiv T' \). This is because \( T \) is followed in \( \tau \) by non-transactional actions that

53
check the return status of the transaction. By our assumption, $\tau'|_{\neg\text{trans}} = \tau|_{\neg\text{trans}}$
thus had $T'$ not completed or had it not executed at all, we would have seen the discrepancy because it would not execute the sequence of non transaction actions which follow $T$. In addition, both $T$ and $T'$ performed the same sequence of method invocations and got the same responses. The former is assured by construction. The latter is checked twice: after every method call the program goes into an infinite loop if the return value is not the expected one, and after the transaction completes the program checks that $b^t_i = 1$. The checks outside of the transaction also ensure that a completed $T$ is committed (or aborted) if and only if so is $T'$.

(ii) If $T$ is force-aborted, then $T'$ is also force aborted and $T \equiv T'$. The assignments to $a^t_i$ and $b^t_i$ ensure that if $T$ is committed or tm-aborted, then $a^t_i = 1$ and $b^t_i = 1$ when the transaction ends. On the other hand, if $T$ is force-aborted, then check $b^t_i \neq 1$ in $\text{lastCheck}^t_i$ would ensure that $T'$ is also force-aborted. The check $b^t_i = 1$ ensures that the transaction was aborted in its last method call and that it performed the same sequence of invocations and responses as $T$.

(iii) If $T$ was force aborted immediately after it started, i.e., $|T| = 2$, then so was $T'$. As there are no method invocations in $T$, checking that $b^t_i \neq 1$ ensures that $T'$ is also force aborted. We need a special case for these kind of transactions because here $a^t_i \neq 1$, while in all other cases of force-aborted transactions we require that $a^t_i = 1$ to ensure that $T \equiv T'$ (see case ii).

(iv) Similarly, if $T$ is live, then the non-terminating loop before the end of the atomic block ensures that $T'$ is live or aborted. Note that only a prefix of a live transaction in $\tau$ may be executed in $\tau'$.

(v) If $T$ is commit-pending, then $T'$ may have any status or may not exist at all.

(vi) The real-time order between transactions in $S_1$ is implied by the real-time order between the corresponding transactions $H$. Let $T''$ be the $j$-th transaction of some thread $t''$ in $H$, for some $j$, and $T'''$ the $j$-th transaction of $t''$ in $S_1$. If $T \prec_H T''$ then $T' \prec_{S_1} T'''$. This holds because $T'''$ cannot start before $T'$ signals its termination.

Stage V: Constructing $H^c$ and $S$. From the above analysis, it follows that $H$ and $S_1$ might differ only by the behavior of their maximal transactions (see Section 3.1). We show how to construct a cp-completion $H^c$ of $H$ and a history $S$, which reconcile these differences by applying certain transformations on $H$ and on $S_1$. The transformations on the former are allowed by the definition of completion and on the latter by the assumed closure properties. For clarity, we describe how we can transform the histories for every maximal transaction, and omit the formal construction.
Recall that in Definition 3.8(i) we construct $H^c$ out of $H$ in the following way:

$$\exists H' \in \text{removecp}(H). \exists H^c \in \text{cpcomplete}(H'|_{\neg \text{live}}). \exists S \in \cal T. H^c \subseteq_{RT} S.$$  

Our first step in obtaining $H^c$ is to remove from $H$ transactions that do not correspond to any transaction in $S_1$. More specifically, we remove the last transaction of every thread which performed more transactions in $H$ than in $S_1$. By point (i), this means that the removed transactions are not completed, and thus can be projected away from $H$ in the completion process by applying either $\text{removecp}(\cdot)$ or $\cdot|_{\neg \text{live}}$. (Note that by the construction of $P_H$ and by our choice of $\tau'$, it is not possible that $S_1$, the history of $\tau'$, have more transactions than $H$.)

Second, we continue to construct $H^c$ and $S$ simultaneously, by transforming pairs of corresponding maximal transactions. Figure 5.5 summarizes the transformations that we use and explains the reasons why these transformations are allowed.

Let $t$ be an arbitrary thread which performs the same number of transactions in $H$ and in $S_1$. Let $T$ be its last transaction in $H$ and $T'$ its last transaction in $S_1$.

**(c-c)** If both $T$ and $T'$ are committed, then we leave them as is in $H^c$ and in $S$: by (i), we get that $T$ and $T'$ are equivalent, i.e., $T \equiv T'$. Thus, we leave them as is.

**(a-a)** Similar to the above case, if both $T$ and $T'$ are aborted, then we leave them as is in $H^c$ and in $S$: by (i,ii,iii), we get that $T$ and $T'$ are equivalent, i.e., $T \equiv T'$.

**(l-l)** If both $T$ and $T'$ are live, then we do not include them in $H^c$ and $S$. In this case, by (iv), $T'$ can be live, aborted or may not exist. Since $\cal T_A$ is closed under removing live and aborted transactions (CLP2), we can remove $T'$ from $S_1$, if it exists, to get $S \in \cal T_A$.

**(l-fa)** Because $\cal T_A$ is closed under removing aborted transactions (CLP2), then this

<table>
<thead>
<tr>
<th>Case</th>
<th>$T$ in $H$</th>
<th>$T'$ in $S_1$</th>
<th>Transformation: $H$ to $H^c$</th>
<th>Transformation: $S_1$ to $S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(c-c)</td>
<td>committed</td>
<td>committed</td>
<td>$-$</td>
<td>$-$</td>
</tr>
<tr>
<td>(a-a)</td>
<td>aborted</td>
<td>aborted</td>
<td>$-$</td>
<td>$-$</td>
</tr>
<tr>
<td>(l-l)</td>
<td>live</td>
<td>live</td>
<td>remove $T$</td>
<td>remove $T'$ (CLP2)</td>
</tr>
<tr>
<td>(l-fa)</td>
<td>live</td>
<td>forceaborted</td>
<td>remove $T$</td>
<td>remove $T'$ (CLP2)</td>
</tr>
<tr>
<td>(cp-cp)</td>
<td>copending</td>
<td>copending</td>
<td>complete $T$</td>
<td>complete $T'$ (CLP3)</td>
</tr>
<tr>
<td>(cp-c)</td>
<td>copending</td>
<td>committed</td>
<td>commit $T$</td>
<td>$-$</td>
</tr>
<tr>
<td>(cp-tma)</td>
<td>copending</td>
<td>tmaborted</td>
<td>abort $T$</td>
<td>$-$</td>
</tr>
<tr>
<td>(cp-l)</td>
<td>copending</td>
<td>live</td>
<td>remove $T$</td>
<td>remove $T'$ (CLP2)</td>
</tr>
<tr>
<td>(cp-fa)</td>
<td>copending</td>
<td>forceaborted</td>
<td>remove $T$</td>
<td>remove $T'$ (CLP2)</td>
</tr>
</tbody>
</table>

Figure 5.5: Summary of operations used to generate $H^c$ from $H = \text{history}(\tau)$ and $S$ from $S_1 = \text{history}(\tau')$ in Lemma 5.1. $T$ is a maximal transaction in $H$ and $T'$ is the corresponding transaction in $S$. Case cp/cp - requires the check inside. Check constant is not uninitialized - not aborted!
case is similar to the previous case (1-l), and again we do not include $T$ and $T'$ in $H$ and $S_1$.

**(cp-cp)** If both $T$ and $T'$ are commit-pending, then we complete $T$ in $H$ and $T'$ in $S_1$ by adding a committed or aborted action: Since $T_A$ satisfies CLP3, there must exist a cp-completion $S$ of $S_1$ in $T_A$. If $T'$ is committed in $S$, then we commit $T$ in $H$ by adding a committed action in $H$ while constructing $H^c$, and if $T'$ is aborted in $S$, then we abort $T$ in $H$ by adding an aborted action in $H$ when we construct $H^c$. Note that completing the commit-pending transaction $T$ in $H$ for constructing $H^c$ is justified since $H^c \in \text{cocomplete}(H'|\neg \text{live})$.

**(cp-c)** If $T$ is commit-pending and $T'$ is committed, we commit $T$ in $H$ by adding a committed action and keep $T'$ as is in $S$. Completing the commit-pending transaction $T$ in $H$ for constructing $H^c$ is justified since $H^c \in \text{cocomplete}(H'|\neg \text{live})$.

**(cp-tma)** If $T$ is commit-pending and $T'$ is tm-aborted, we remove $T$ from $H$ and $T'$ from $S_1$. Note that Definition 3.8(i) allows removing some of the commit-pending transactions by using $\text{removecp}(H)$. Furthermore, as $S_1 \in T_A$ and $T_A$ satisfies CLP2, removing the (aborted) transaction $T'$ from $S_1$ keeps $S$ in $T_A$.

**(cp-l)** This case is similar to the previous case (cp-tma), and we do not include $T$ and $T'$ in $H^c$ and $S$, respectively.

**(cp-fa)** This case is also similar to the case of (cp-tma), and we do not include $T$ and $T'$ in $H^c$ and $S$, respectively.

---

**Lemma 5.2** Let $T_C$ and $T_A$ be TMs such that $T_C \preceq_{\text{noRB}} T_A$ and $T_A$ satisfies CLP2 and CLP3. Let $H = H' \psi \in T_C$, where $\psi$ is a response action that is not a committed or aborted action. There exist $H^c \in \text{cSTMSpast}(H)$ and $S \in T_A$ such that $H^c \sqsubseteq_{RT} S$.

**Proof:** Let $\psi = (\cdot, t_0, \cdot)$. Assume that $\psi$ appear in $k_\psi$ transaction of $t_0$ in $H$. We construct a program $P_H = C^1_H \parallel \ldots \parallel C^m_H$ where, as in Lemma 5.1, every thread $t = 1..m$ performs the sequence of transactions specified in $H_t$. In fact, the construction of the $C^t_H$ for thread $t \neq t_0$ is same as in Lemma 5.1. The construction of $C^{t_0}_H$, however, is done somewhat differently. In particular, $C^{t_0}_H$ contains a single fault command. The latter is used to observe the behavior of $t_0$’s last transaction during its execution.

The proof itself is comprised of the same stages which we used in Lemma 5.1. As in Lemma 5.1, $P_H$ monitors certain properties of the TM behavior, e.g., checking that the return values obtained from methods of transactional objects in committed transactions correspond to those in $H$ and that the real-time order between actions includes that in $H$. The reaction of the threads when detecting a mismatch, is the same as in Lemma 5.1; they enter a non-terminating loop. However, thread $t_0$ has a
special reaction when it detects that the execution had no mismatch from its own local perspective: it executes a fault.

In the following, we reuse the notation introduced in the proof of Lemma 5.1 (Figure 5.1 and Figure 5.2).

**Stage I: Construction of the monitoring program.** We use the same program \( P_H \) as in the previous lemma, with one exception: we add a fault right before the end of the transaction of \( \psi \). More formally, we change the definition of \( \text{lastCommand} \) to be as before, except that

\[
\text{lastCommand}_i^t = \text{fault}, \quad \text{where } i = k_\psi.
\]

This construction is motivated by the fact that faulting is the only observation Definition 2.8 allows us to make about the behavior of the live transaction of \( \psi \).

**Stage II: Constructing \( \tau \).** We construct \( \tau \) in the same way as in the previous lemma, except that we place the fault command right after \( \psi \). Formally, we partition every \( \tau^t \) into subsequences \( \tau^t_i \) as defined by (5.1), but with the conditions in Figure 5.4 changed so that the case of \( t = t_0 \) and \( \psi_{t_0}^i = \psi \) is treated specially: in this case

\[
\tau^t_i = \psi_\sim(t, \text{fault}).
\]

Then we let \( \tau \) be defined by (5.2), so that \( \text{history}(\tau) = H \).

**Stage III: Defining \( s \).** We define \( s \) to be the same initial state chosen as in Lemma 5.1.

**Stage IV: Obtaining \( \tau' \).** As before, we have \( \tau \in [P_H]_{RB}(s, T_C) \). Since \( \tau \) ends with a fault by \( t_0 \) and \( T_C \preceq \text{noRB} \) \( T_A \), by Definition 2.8 there exists \( \tau' \in [P_H]_{\text{noRB}}(s, T_A) \) that also ends with a fault by \( t_0 \). Let \( S_1 = \text{history}(\tau') \in T_A \).

Since \( t_0 \) faults in \( \tau' \), the checks inside this thread ensure that

\[
H|_{t_0} \equiv S_1|_{t_0}. \tag{5.3}
\]

Consider a thread \( t \neq t_0 \), and let \( T \) be the \( i \)-th transaction in \( \tau|_t \) and \( T' \) be the \( i \)-th transaction in \( \tau'|_t \), if such a transaction exists. These transactions arise from executing the same commands, and \( T' \) might not exist if the commands did not execute in \( \tau' \).

Let \( T_0 = \text{txof}(\psi, H) \). Note that if \( T \) and \( T' \) are executed by the thread \( t_0 \), then \( T = T' \), because of the fault. We now analyze the relationship between \( T \) and \( T' \). The construction of \( CP^t_i \) ensures the following:

(i) If \( T \prec_H T_0 \) and \( T \) is completed, then so is \( T' \). This is because \( T \) is followed in \( \tau \) by non-transactional actions that check the return status of the transaction. By our assumption, both \( \tau \) and \( \tau' \) end with a fault by \( t_0 \) inside the transaction \( T_0 \), thus had \( T' \) not completed or aborted, we would have seen the discrepancy.

In addition, both \( T \) and \( T' \) performed the same sequence of method invocations
and got the same responses. The former is assured by construction. The latter is checked twice: after every method call the program goes into an infinite loop if the return value is not the expected one, and after the transaction completes the program checks that $b' = 1$. The checks outside of the transaction also ensure that a completed $T$ is committed (or aborted) if and only if so is $T'$.

(ii) If $T \not\prec_H T_0$, then $T'$ may have any status or may not exist at all. This is because the only observation we have is the fault by thread $t_0$, and $t_0$ can observe only the transactions that completed before $T_0$, using non-transactional actions. Since we do not allow non-transactional actions inside transactions, $t_0$ cannot observe the execution of transactions that begin after txbegin of $T_0$ or completed after txbegin of $T_0$.

(iii) Every transaction $\hat{T}$ which precedes $T$ in $H$ has a corresponding transaction $\hat{T}'$ in $\tau'$ which precedes $T'$, i.e., if $\hat{T}$ is the $j$-th transaction of some thread $\hat{t}$ in $H$, then $\hat{T}$ has at least $j$ transaction in $\tau'$; otherwise, it would not have signaled its termination using a global variable, a variable which $T'$ must read before it starts executing.

(iv) If both $T$ and $T'$ performed more that $p$ method calls than the responses they got for the first $p$ calls match; otherwise, $T'$ would have gotten stuck before its $p + 1$ invocation.

(v) If $T$ is live, then $T'$ may have any status. Note that this is a special case of (ii).

(vi) If $T \prec_H T_0$ and $T$ is force-aborted, then the non-terminating loop before the end of the atomic block ensures that $T'$ is live or force-aborted.

(vii) If $T$ is commit-pending, then $T'$ may have any status or may not exist at all. Note that this is a special case of (ii).

(viii) The real time order between transactions in $S_1$ is implied by the realtime order between the corresponding transactions $H$. Specifically, Let $T''$ be the $j$-th transaction of some thread $t''$ in $H$, for some $j$, and $T'''$ the $j$-th transaction of $t''$ in $S_1$. If $T \prec_H T''$ then $T' \prec_{S_1} T'''$. This holds because $T'''$ cannot start before $T'$ signals its termination.

Stage V: Constructing $H^c$ and $S$. From the above analysis it follows that $H$ and $S_1$ might differ only by the behavior of the transactions that did not complete before the txbegin of $T_0$. Note that this include not only the live and commit-pending transactions in $H$ by threads $t \neq t_0$, but also all completed transactions that completed after the txbegin of $T_0$.

We show how we can find a completed possible past $H^c$ of $H$ and a history $S$ which reconcile these differences by applying certain transformations on $H$ and on $S_1$. The transformations on the former are allowed by the definition of completed possible past
and on the latter by the assumed closure properties: As in the proof of Lemma 5.1, we describe how we can transform the histories for every maximal transaction, and omit the formal construction.

Let \( t \) be an arbitrary thread which performs \( i \) transactions in \( S_1 \). Let \( T \) be its \( i \)-th transaction in \( H \) and \( T' \) its \( i \)-th transaction in \( S_1 \). We first note that if \( i = 0 \) then by property (i), \( T \) must be either live or commit-pending when \( T_0 \) starts and by property (iii), it must be maximal in \( H \). Thus, it can be removed according to Definition 3.4(ii,iii). As we handled the case where \( i = 0 \), in the following we assume that \( T' \) exists.

Note that if \( T \prec_H T_0 \), then \( T' \) exists and, by (i), has the same status as \( T' \). In this case, we keep both \( T \) and \( T' \) as is in \( H \) and \( S_1 \). This is justified by the possible past definition (Definition 3.4(iii)), which includes all the transactions preceding \( T_0 \) in the real-time order in \( H \).

If \( T \not\prec_H T_0 \), by (ii), \( T' \) can have any status. If \( T' \) is force-aborted, live or does not exist, by (ii), then we remove \( T \) from \( H \). This is justified by the possible past (Definition 3.4(ii)), which allows removing some of the visible transactions in \( H \). Since \( T_A \) is closed under removing live and aborted transactions (CLP2), we can remove \( T' \) from \( S_1 \), if it exists, to get \( S \in T_A \). For other cases, we continue by case analysis according to the status of \( T \) and \( T' \).

(c-c) If both \( T \) and \( T' \) are committed, then we leave them as is in \( H^c \) and in \( S \): by (i), we get that \( T \) and \( T' \) are equivalent, i.e., \( T \equiv T' \). Thus, we leave them as is. This is justified by the possible past (Definition 3.4(ii)), which allows including some of the visible transactions in \( H \).

(a-a) Similar to the above case, if both \( T \) and \( T' \) are aborted, then we leave them as is in both \( H^c \) and \( S \): by (i), we get that \( T \) and \( T' \) are equivalent, i.e., \( T \equiv T' \).

(a-c) If \( T \) is aborted and \( T' \) is committed, then we include \( T \) as a committed transaction by replacing the aborted action of \( T \) by a committed action. This is justified by the completed possible past definition (Definition 3.7), which allows replacing the aborted actions of all maximal aborted transactions by committed actions.

(c-a) If \( T \) is committed and \( T' \) is aborted, then we remove \( T \) from \( H \) and \( T' \) from \( S_1 \). This is justified as Definition 3.4(ii) allows removing some of the visible transactions of \( H \). Furthermore, as \( S_1 \in T_A \) and \( T_A \) satisfies CLP2, removing the transaction \( T' \) from \( S_1 \) keeps \( S \) in \( T_A \).

(cp-cp) If both \( T \) and \( T' \) are commit-pending, then we complete \( T \) in \( H \) and \( T' \) in \( S_1 \) by adding a committed or aborted action. Since \( T_A \) satisfies CLP3, there must exist a cp-completion \( S \) of \( S_1 \) in \( T_A \). If \( T' \) is committed in \( S \), then we commit \( T \) in \( H \) by adding a committed action in \( H \) while constructing \( H^c \), and if \( T' \) is aborted in all the cp-completions of \( S_1 \), then we remove \( T \) from \( H \) and \( T' \) from \( S_1 \). Note
that completing the commit-pending transaction $T$ in $H$ for constructing $H^c$ is justified since $H^c$ is a commit justification of a history obtained from a possible past of $H$.

**(cp-c)** If $T$ is commit-pending and $T'$ is committed, we commit $T$ in $H$ by adding a committed action and keep $T'$ as is in $S$. This is justified since $H^c$ is a commit justification of a history obtained from a possible past of $H$.

**(cp-tma)** If $T$ is commit-pending and $T'$ is tm-aborted, we remove $T$ from $H$ and $T'$ from $S_1$. Note that Definition 3.4(ii) allows removing some of the commit-pending transactions of $H$. Furthermore, as $S_1 \in T_A$ and $T_A$ satisfies CLP2, removing the transaction $T'$ from $S_1$ keeps $S$ in $T_A$.

**Proof:** [of Theorem 3.2(ii)] Assume $T_C \preceq_{\text{noRB}} T_A$, and $T_A$ satisfies CLP2 and CLP3.

From Lemma 5.1, we get that for every history $H \in T_C$ there exists histories $H' \in \text{removecp}(H), H^c \in \text{cpcomplete}(H' \mid \neg \text{live})$ and $S \in T_A$ such that $H^c \sqsubseteq_{\text{RT}} S$. This gives part (i) of STMS relation (Definition 3.8).

From Lemma 5.2, we get that for every history $H = H' \psi$ in $T_C$, where $\psi$ is a response action that is not a committed or aborted action, there exists a history $H^\psi \in \text{cSTMSpast}(H)$ and a history $S^\psi \in T_A$ such that $H^\psi \sqsubseteq_{\text{RT}} S^\psi$, giving part (ii) of STMS relation (Definition 3.8), and hence completing the proof.

5.2 **Proof of Theorem 3.3(ii): Necessity of the TMS1 Relation**

**Lemma 5.3** Let $T_C$ and $T_A$ be TMs such that $T_C \preceq_{\text{RB}} T_A$, and $T_A$ satisfies CLP2 and CLP3. Then

$$\forall H \in T_C. \exists H' \in \text{removecp}(H), H^c \in \text{cpcomplete}(H' \mid \neg \text{live}), S \in T_A. H^c \sqsubseteq_{\text{RT}} S.$$ 

**Proof:** Similarly to the proof of Lemma 5.1, for every history $H \in T_C$ we construct a program $P_H$ where every thread performs the sequence of transactions specified by $H$. The construction of the program $P_H$, a particular trace $\tau$ of $P_H$ and a trace $\tau'$ for every sequential command $C_H$ is the same as in the proof of Lemma 5.1, except for the assignment to variables $a_i$ and $b_i$, and the lastCheck command, which are omitted. As we are assuming the semantics with rollback, we cannot gain any information regarding the execution of aborted transactions done outside the atomic block from these variables as their values is rolled back. Since $\tau' \in \text{Tr}(C_H) \tau$, we have $\tau \in \text{Tr}(P_H)$. Let $s$ be the state where all the local variables are set to $u$ and for all $t$, $g_i^t = 0$ for $i \neq 0$ and $g_0^t = 1$. By the construction of $\tau$, we have $\text{eval}_{\text{RB}}(s, \tau) \neq \emptyset$. This is because $\tau$ does not get stuck (enters a non-terminating loop), and it is possible to evaluate $\tau$ till completion.
Then, since $\text{history}(\tau) = H \in T_C$, we have $\tau \in [P_H]_{\text{RB}}(s, T_C)$. Since $\tau \in [P_H]_{\text{RB}}(s, T_C)$ and $T_C \preceq_{\text{RB}} T_A$, by Definition 2.8 there exists a trace $\tau' \in [P_H]_{\text{RB}}(s, T_A)$ such that $\tau'\neg\text{-trans} = \tau\neg\text{-trans}$ and $S_1 = \text{history}(\tau') \in T_A$.

Consider a thread $t$, and let $T$ be the $i$-th transaction in $\tau|_t$ and $T'$ be the $i$-th transaction in $\tau'|_t$, if such a transaction exists. These transactions arise from executing the same commands, and $T'$ might not exist only if the commands did not execute in $\tau'$. We now analyze the relationship between $T$ and $T'$: The analysis is similar to the one we did in Stage IV in the proof of Lemma 5.2, but provides slightly weaker properties because we consider the semantics with rollback: Hence, we cannot observe the values of local variables changed inside aborted transactions, specifically $a^t_i$ and $b^t_i$. Luckily, we can manage because there is no need to reason about responses inside aborted transactions as Definition 3.10(i) considers only committed and some of commit-pending transactions in $H$. In fact, it does not even matter if $T$ is tm-aborted and $T'$ is force aborted, or vice versa. All we care about is that they are both aborted.

(1) $T$ and $T'$ are related as specified in properties (iii),(iv), and (vi) in the proof of Lemma 5.2 and for the same reasons. Note that the proof that these properties hold does not involve reasoning about the values of the $a^t_i$ and $b^t_i$ variables.

(2) If $T$ is committed, then $T$ and $T'$ are related as specified in property (i). Note that the proof that the property holds in this case mentioned $a^t_i$ and $b^t_i$, but only as an alternative way to prove the property.

(3) If $T$ is aborted, then so is $T'$. Recall that $P_H$ checks that the status of transaction when they complete. Thus any discrepancy would become observable.

From the above analysis, it follows that $H$ and $S_1$ might differ only by the behavior of their aborted or maximal transactions. We show how we can find a completion of $H^c$ and a history $S$ which reconcile these differences by applying certain transformations on $H$ and on $S_1$. Note that to show that the conditions in Definition 3.10(i) hole, we need to construct $H^c$ out of $H$ such that the following holds:

$$\exists H' \in \text{removecp}(H). \exists H^c \in \text{cpcomplete}(H'|_\text{live}). \exists S \in T. H^c|_\text{com} \subseteq_{\text{RT}} S.$$  

We begin by constructing $H^c$ and $\hat{S}$ in the same way we constructed $H^c$ and $S$ in Stage IV of the proof of Lemma 5.2. This is possible because:

- By property (1), which ensures that the relation between commit-pending transactions and commit transactions in $H$ and in $S_1$ are as in Lemma 5.2.

- By properties (2) and (3), committed and aborted transactions in $H$ correspond to committed, respectively, aborted transactions in $S_1$.

Note, however, that the constructed history $\hat{S}$ might contain aborted transactions, which might differ from the corresponding ones in the $H^c$, the chosen completion of $H$. However, $S$ is required by Definition 3.10(i) to contain only committed transactions, and those match in $H^c$ and $\hat{S}$. Let $S = \hat{S}|_\text{com}$. $S \in T_A$ because $S \in T_A$ and $T_A$ is
closed under removing live and aborted transactions (CLP2). Since for committed and commit-pending transactions, \( H^c \) and \( S_1 \) have the same responses, and the construction for committed and commit-pending transactions is same as in Lemma 5.1, we get that \( H^c|_{\text{com}} \sqsubseteq_{RT} S \).

**Lemma 5.4** Let \( T_C \) and \( T_A \) be TMs such that \( T_C \preceq_{RB} T_A \) and \( T_A \) satisfies CLP2 and CLP3. Let \( H = H'\psi \in T_C \), where \( \psi \) is a response action that is not a committed or aborted action. There exist \( H^c \in \text{cTMSpast}(H) \) and \( S \in T_A \) such that \( H^c \sqsubseteq_{RT} S \).

**Proof:** Recall that by Definition 3.9, to construct \( H^c \) out of \( H'\psi \) we need to find some \( H^c_0 \) such that \( H^c = H^c_0|_{\text{com}} \) and \( H^c_0 \in \text{cSTMSpast}(H) \).

In Lemma 5.2, we were able to show the existence of some \( S_0 \) in \( T_A \) such that \( H^c_0 \sqsubseteq_{RT} S \). There, however, we relied on the observations obtained by semantics without rollback. In this proof, we assume the semantics with rollback, and thus we cannot observe the behavior of (non faulty) aborted transactions. Luckily, as in Lemma 5.3, we can overcome this limitation because the condition we need to prove does not concern the contents of aborted transactions. However, instead of projecting away the aborted transaction all together using \( \cdot|_{\text{com}} \), we remove their contents using \( \cdot|_{\text{not abortedtx}} \).

The construction of \( H^c_0 \) and \( S_1 \) through the history \( S_1 \) of a trace \( \tau' \) which faults when \( P_H \) is executed using \( T_A \) is virtually identical to the one we used in Lemma 5.2: Stages I-III go through as before. Properties (ii)-(viii) listed in stage IV and regarding the relation between \( H \) and of \( S_1 \), still hold. Property (i) holds with the expected exception: Aborted transactions in \( H^c_0 \) and in \( S_1 \) might behave differently. Thus, instead of (5.3) we have

\[
(H|_{\text{not abortedtx}})|_{t_0} \equiv (S_1|_{\text{not abortedtx}})|_{t_0}.
\]

The construction of \( H^c \) and \( S \) is done as in Stage V. Specifically, we apply the same case analysis as in Lemma 5.2 and perform the same changes to \( H \) and \( S \). As before, the only point that might be worth discussion concerns aborted transactions. History \( H \) and \( S_1 \) might contain pairs of corresponding aborted transactions (i.e., both are the \( n \)-th transactions of some thread) which might differ in the actions that they take. This discrepancy is immaterial, as the condition is oblivious to the internal actions of such transactions. (Recall the use of \( \cdot|_{\text{not abortedtx}} \).) Thus, unsurprisingly, we can only establish

\[
\forall t. (H^c_0|_{\text{not abortedtx}})|_t \equiv (S_1|_{\text{not abortedtx}})|_t.
\]

Given the resulting histories, \( H^c_0 \) and \( \hat{S} \), and considering the fact that the real-time order between corresponding transaction in \( S \) is implied by that of \( H^c_0 \) (property (viii)), and that carving out the contents of transactions does not affect their real time order, by choosing \( H = H^c_0|_{\text{not abortedtx}} \) and \( S = \hat{S}|_{\text{not abortedtx}} \), we get \( H^c \sqsubseteq_{RT} S \). The former
transformation is justified by the definition of $cTMS_{past}(H)$ and the latter by CLP2, which ensures that because $T_A$ is close under the removal of aborted transactions, we have that $S \in T_A$.

**Proof:** [of Theorem 3.3(ii)] Assume $T_C \leq_{RB} T_A$, and $T_A$ satisfies CLP2 and CLP3.

From Lemma 5.3, we get that for every history $H \in T_C$ there exist histories $H' \in \text{removecp}(H), H^c \in \text{cocomplete}(H'|\text{live})$ and $S \in T_A$ such that $H^c|_{\text{com}} \sqsubseteq_{RT} S$. This gives part (i) of TMS1 relation (Definition 3.10).

From Lemma 5.4, we get that for every history $H = H'\psi$ in $T_C$, where $\psi$ is a response action that is not a committed or aborted action, there exists a history $H^\psi \in cTMS_{past}(H)$ and a history $S^\psi \in T_A$ such that $H^\psi \sqsubseteq_{RT} S^\psi$, giving part (ii) of TMS1 relation (Definition 3.10), and hence completing the proof.
Chapter 6

Other Consistency Conditions

In this chapter, we define other TM consistency conditions in our setting, and relate them to STMS and TMS1.

6.1 Opacity

In this section, we show that the original notion of opacity [14, 16] can be obtained from $\sqsubseteq_{RT}$ by instantiating Definition 3.2 with the abstract transactional system $T_{\text{atomic}}$ (defined in Chapter 2), in which atomic blocks execute atomically and methods called by aborted transactions have no effect. We capture the original notion of opacity [14, 16] in our framework.

Definition 6.1 A history $H_c$ is a suffix completion of a history $H$, if it is a complete history, $H$ is a prefix of $H_c$, and $H_c$ can be constructed from $H$ by appending to it a completing history for the last action of every thread. We denote the set of suffix-completions of $H$ by $\text{comp}(H)$.

Definition 6.2 A TM $T_C$ is opaque if for every history $H \in T_C$, there exists a history $H_c \in \text{comp}(H)$ and a complete, non-interleaved and legal history $S_c$ such that $H_c \sqsubseteq_{RT} S_c$.

The main difference between the original notion of opacity (Definition 6.2) and the one obtained by instantiating Definition 3.2 with $T_{\text{atomic}}$ is that Definition 6.2 first completes a history from $T_C$ and then finds its match according to the real-time relation. In contrast, our criterion $H_c \sqsubseteq_{RT} S_c$ first finds the match and then completes the matching history. For technical reasons, Definition 6.2 also uses a slightly different completion, putting completing histories at the end to avoid creating new real-time orderings. Fortunately, completion and matching commute, and thus the two formulations of opacity are equivalent.

Theorem 6.1 A TM $T_C$ is opaque if and only if $T_C \sqsubseteq_{RT} T_{\text{atomic}}$.
Proof: Only if. Assume that \( \mathcal{T}_C \) is an opaque TM and let \( H \in \mathcal{T}_C \). By the definition of opacity, there is a complete history \( H_c \in \text{comp}(H) \) and a complete, non-interleaved and legal history \( S_c \) such that \( H_c \sqsubseteq_{\text{RT}} S_c \).

Let \( S \) be the history produced by removing all the actions in \( S_c \) added while completing \( H \) to \( H_c \). It is easy to see that \( S \) is non-interleaved and that \( H|_t = S|_t \) for every thread \( t \). Hence, the last action of every thread \( t \) in \( H \) is the same as its last action in \( S \). Furthermore, the actions used to suffix-complete \( H \) to \( H_c \) can be used to show that \( S_c \) is a non-interleaved completion of \( S \). Hence, \( S_c \in \text{nicompleteall}(S) \).

Since \( S_c \) is legal, we have \( S \in \mathcal{T}_\text{atomic} \). Furthermore, the real-time order between actions in \( H_c \) is preserved in \( S_c \), and hence, the real-time order in \( H \) is preserved in \( S \), so that \( H \sqsubseteq_{\text{RT}} S \). The history \( H \in \mathcal{T}_C \) was chosen arbitrarily, hence, \( \mathcal{T}_C \sqsubseteq_{\text{RT}} \mathcal{T}_\text{atomic} \).

If. Assume \( \mathcal{T}_C \sqsubseteq_{\text{RT}} \mathcal{T}_\text{atomic} \) and let \( H \in \mathcal{T}_C \). Hence, there is a history \( S \in \mathcal{T}_\text{atomic} \) such that \( H \sqsubseteq_{\text{RT}} S \). Since \( S \in \mathcal{T}_\text{atomic} \), it is non-interleaved and there exists a complete, non-interleaved and legal history \( S_c \in \text{nicompleteall}(S) \).

Since \( H \sqsubseteq_{\text{RT}} S \), for every thread \( t \) we have \( H|_t = S|_t \), and, hence, the last action of every thread \( t \) in \( H \) is the same as its last action in \( S \). Therefore, the same actions used to create \( S_c \) as a non-interleaved completion of \( S \) can be used to suffix-complete \( H \) to a history \( H_c \in \text{comp}(H) \).

It is easy to see that \( H_c|_t = S_c|_t \) for every thread \( t \). Note that no \texttt{txbegin} actions are added to \( H \) to produce \( H_c \) and \texttt{aborted} or \texttt{committed} actions are added to the end of the history. Hence, no new real-time orders are introduced when suffix-completing \( H \) to \( H_c \). Since the real-time order in \( H \) is preserved in \( S \) and the real-time order in \( S \) is preserved in \( S_c \), we get that the real-time order in \( H_c \) is preserved in \( S_c \). Hence, \( H_c \sqsubseteq_{\text{RT}} S_c \). The history \( H \in \mathcal{T}_C \) was chosen arbitrarily. Thus, from \( H_c \in \text{comp}(H) \) and \( H_c \sqsubseteq_{\text{RT}} S_c \) it follows that \( \mathcal{T}_C \) is opaque. \( \blacksquare \)

It is clear from Theorem 6.1 and Definition 3.8 that every opaque history is also STMS. However, there are histories which are STMS but not opaque. For example, Figure 6.1(a) shows a history that is STMS but not opaque. Opacity requires a suffix completion \( H_c \) of the history to be in real-time relation with a non-interleaved, complete and legal history \( S_c \) (Definition 6.2). The only suffix completion \( H_c \) of the history shown in Figure 6.1(a) is in which the transaction \( T_3 \) is aborted. It is easy to see that there cannot exist a non-interleaved, complete and legal history \( S_c \) such that \( H_c \sqsubseteq_{\text{RT}} S_c \). This is because the operations on the object \( x \) require \( T_1 \) to precede \( T_2 \) and \( T_2 \) to precede \( T_3 \) in \( S_c \), which invalidates the legality of \( S_c \) because of operations on the object \( y \). However, this history is STMS as we can remove the live transaction \( T_3 \) to get \( H^c \) and find a non-interleaved and a complete history \( S = T_1T_2 \), which is legal, such that \( H^c \sqsubseteq_{\text{RT}} S_c \) (Definition 3.8 (i)). Also, for every response action \( \psi \) that is not a \texttt{committed} or \texttt{aborted} action, we can find a completed possible past \( H^c \psi \) of the history and a non-interleaved history \( S^c \psi \) such that \( H^c \psi \sqsubseteq_{\text{RT}} S^c \psi \) (Definition 3.8 (ii)), and \( S^c \psi \) has a non-interleaved completion which is legal. The non-trivial response actions in
this history are the responses for the reads by $T_3$. For both these response actions, the completed possible past contains $T_2$ and a prefix $T_3'$ of $T_3$ (till the response actions), and $S_\psi = T_2T_3'$.

Now we show that every history that satisfies STMS and contains only completed transactions is also opaque. This shows that the difference between STMS and opacity is in the way these conditions handle live transactions.

**Lemma 6.2** Let $H$ be a complete history such that $H \subseteq_{\text{stms}} T$, for a TM $T$. Then there exists a history $S \in T$ such that $H \subseteq_{\text{RT}} S$.

**Proof:** By Definition 3.8(i), there exists a history $H' \in \text{removecp}(H)$, a history $H' \in \text{cpcomplete}(H'|_{-\text{live}})$ and a history $S \in T$ such that $H' \subseteq_{\text{RT}} S$.

Since $H$ is complete, $\text{removecp}(H) = \{H\}$ and $H|_{-\text{live}} = H$. Therefore, $H = H'$ and $\text{cpcomplete}(H'|_{-\text{live}}) = \text{cpcomplete}(H|_{-\text{live}}) = \text{cpcomplete}(H) = \{H\}$, giving $H = H'$, and hence $H \subseteq_{\text{RT}} S$, completing the proof.

The following lemma shows that if a suffix completion of a history $H$ is STMS, then $H$ is opaque.

**Lemma 6.3** Let $H$ be a well-formed history. If there exists a history $H_c \in \text{comp}(H)$ such that $H_c \subseteq_{\text{stms}} T$, for a TM $T$, then there exists a history $S \in T$ such that $H \subseteq_{\text{RT}} S$.

**Proof:** Let $H_c \in \text{comp}(H)$ such that $H_c \subseteq_{\text{stms}} T$. Since $H_c$ is a complete history and $H_c \subseteq_{\text{stms}} T$, using Lemma 6.2, we get a history $S_c \in T$ such that $H_c \subseteq_{\text{RT}} S_c$.

Let $H_c = HH'$. Let $S$ be the history produced by removing all the actions in $S_c$ added while completing $H$ to $H_c$, i.e., $S$ is the subsequence of $S_c$ obtained by removing all actions from $S_c$ that are in $H'$. We will show that $H \subseteq_{\text{RT}} S$.

Since $H_c \subseteq_{\text{RT}} S_c$, for every thread $t$, $H_c|_t = S_c|_t$, which further implies, by the construction of $S$, $H|_t = S|_t$, for every thread $t$. Furthermore, the real-time order between actions in $H_c$ is preserved in $S_c$, and hence, the real-time order in $H$ is preserved in $S$. This is true since for any actions $\psi$ and $\psi'$ in $H$ and $S$, if $\psi \prec_H \psi'$ then $\text{txof}(\psi, H)$ is complete and thus $\text{txof}(\psi, H) = \text{txof}(\psi, H_c) = \text{txof}(\psi, S_c) = \text{txof}(\psi, S)$, which implies $\psi \prec_H \psi' \implies \psi \prec_{H_c} \psi' \implies \psi \prec_{S_c} \psi' \implies \psi \prec_{S} \psi'$.

Therefore, $H \subseteq_{\text{RT}} S$, completing the proof.

Theorem 6.4 is an easy corollary of Lemma 6.3. It depends on the following closure property.

**CLP4** A TM $T$ is **closed under completing transactions** if whenever $H \in T$, we have $\text{comp}(H) \cap T \neq \emptyset$.

**Theorem 6.4** Let $T$ be a TM that satisfies CLP4. $T$ is STMS $\iff$ $T$ is opaque.
Figure 6.1: Examples of STMS, VWC, Opaque, DU-opaque and TMS2 histories.

6.2 Deferred-update (DU) opacity

In this section, we give the definition of DU-opacity [4] in our setting, and then show that it is strictly stronger than STMS.

For any action $\psi$, let $H \mid \psi$ denote the prefix of $H$ up to the action $\psi$.

**Definition 6.3** A history $S'$ is the local serialization of an action $\psi$ with respect to histories $H$ and $S$ if:

- $S'$ is a subsequence of $S \mid \psi$,
- $S'$ contains $\text{txof}(\psi, S \mid \psi)$, and
- $T \in \text{tx}(S') \implies T \in \text{visible}(H \mid \psi) \cup \text{txof}(\psi, H \mid \psi)$.

We denote the local serialization of an action $\psi$ with respect to histories $H$ and $S$ by $S'_{H \mid \psi}$.
Informally, $S'_\psi$ is the subsequence of $S|_\psi$ derived by removing from $S|_\psi$ the actions of all transactions $T' \in \text{tx}(H) \setminus \{T\}$, where $T = \text{txof}(\psi, H)$, such that $T'$ is not visible in $H|_\psi$.

**Definition 6.4** A TM $T_C$ is **du-opaque** if for every history $H \in T_C$ there exists a history $H_c \in \text{comp}(H)$ and a complete, non-interleaved and legal history $S_c$ such that
- $H_c \sqsubseteq_{RT} S_c$, and
- for every response action $\psi$ in $H$ that is not a committed or aborted action, a non-interleaved completion of $S_c', S_c$, in which the response of every txcommit action is same as in $H$, if it exists, is legal.

Informally, for every history $H$ of a du-opaque TM, there is a legal non-interleaved history $S$ that is equivalent to $H$, respects the real-time ordering of transactions in $H$ and every action is legal in its local serialization with respect to $H$ and $S$. The second condition reflects the implementation’s deferred-update semantics, i.e., the legality of an action in a serialization does not depend on transactions that start committing after the response.

It is clear from Definition 6.4 that every du-opaque TM is also opaque. From our discussion in Section 6.1, we can conclude that every du-opaque TM is also STMS.

Figure 6.1(b) shows a history that is opaque, but not du-opaque. It is easy to see that the history is opaque. Since the history is complete, its suffix completion $H_c$ is the history itself. The only possible complete, non-interleaved and legal histories $S$ such that $H_c \sqsubseteq_{RT} S$ are $S_1 = T_1T_3T_2$ or $S_2 = T_3T_1T_2$ or $S_3 = T_3T_2T_1$. Consider the response of the read operation by $T_2$. In the prefix of the history until this response action, $T_1$ is commit-pending and $T_3$ is live. Therefore, the local serialization of this response action with respect to this history and every $S \in \{S_1, S_2, S_3\}$ cannot not contain $T_3$. Since the response of every txcommit action in a completion of local serialization must match that in $H$ for it to be du-opaque, $T_1$ must be aborted in completions of all the possible local serializations, leaving the response action not legal.

### 6.3 Virtual World Consistency (VWC)

VWC [26] is a relaxation of opacity, allowing every aborted transaction to be justified by a separate abstract history. In this section, we introduce a weaker notion of VWC and discuss its relation with STMS and the original definition of VWC. We also prove that it is sufficient for a weaker notion of observational refinement under the semantics with rollback.

We call the original definition of VWC [26], **Original VWC** (OVWC).

**Definition 6.5** A history $H_\psi = H'_1\psi$ is a **ovwc possible past** of a history $H = H_1\psi$, where $\psi$ is an aborted action, if:

---

69
(i) $H'_1$ is a subsequence of $H_1$;
(ii) $H_\psi$ is comprised of the transaction of $\psi$ and some of the committed transactions in $H$:
\[ \text{tx}(H_\psi) \subseteq \{ \text{txof}(\psi, H) \} \cup \text{committed}(H). \]
(iii) for a thread $t \in \text{ThreadID}$, for every transaction $T \in H_\psi | t$, out of all transactions preceding $T$ in the real-time order in $H | t$, the history $H_\psi | t$ includes exactly the committed ones preceding $T$:
\[ \forall t \in \text{ThreadID}. \forall T \in \text{tx}(H_\psi | t). \forall T'. T' \prec_{H_\psi | t} T \iff T' \prec_{H | t} T \wedge T' \in \text{committed}(H). \]

We denote the set of ovwc possible pasts of $H$ by $\text{OVWCpast}(H)$.

**Definition 6.6** A history $H$ is in the **Original VWC (OVWC) relation** with TM $T$, denoted $H \sqsubseteq_{\text{ovwc}} T$, if:
(i) $\exists S \in T. H | \text{com} \sqsubseteq_{\text{RT}} S$
(ii) for every aborted action $\psi$ such that $H = H_1 \psi H_2$: $\exists H'_\psi \in \text{OVWCpast}(H_1 \psi). \exists S_\psi \in T. H'_\psi \sqsubseteq_{\text{RT}} S_\psi$.
A TM $T_C$ is in the **OVWC relation** with a TM $T_A$, denoted by $T_C \sqsubseteq_{\text{ovwc}} T_A$, if $\forall H \in T_C. H \sqsubseteq_{\text{ovwc}} T_A$. A TM $T_C$ is **OVWC** if $T_C \sqsubseteq_{\text{ovwc}} T_{\text{atomic}}$.

Note that the original definition is weaker than STMS, but incomparable with TMS1. The main difference between OVWC and STMS is that OVWC requires that a whole transaction be explained by an abstract history, while STMS allows each response action to be explained by a different abstract history. OVWC is based on causal past and considers only some of the committed transactions for abstract history of an aborted transaction (the ones in its causal past), while STMS considers all the transactions in the past. For example, in Figure 3.1, for STMS, the abstract history for transaction $T_4$ must contain the transaction $T_1$, while VWC does not mandate this inclusion.

Moreover, STMS allows a transaction to read from a commit-pending transaction, even if it gets aborted later. Thus, STMS allows including concurrent aborted transactions for abstract history while OVWC does not. In Figure 3.1, STMS allows the abstract history for transaction $T_5$ to contain transactions $T_2$ and $T_4$, while OVWC does not allow including $T_2$ and $T_4$, if these transactions abort.

Since OVWC is based on causal past, it assumes unique writes, i.e., no two transactions write the same value on the same object to detect causality. STMS does not make this assumption.

Now we introduce our notion of VWC. The main difference between OVWC and our notion of VWC is that in the original definition, every aborted transaction is explained by an abstract history, while in our definition every response in an aborted transaction is explained by an abstract history. Therefore, by our definition two reads by an aborted transaction can be justified by two different abstract histories while the
original definition restricts different reads by same transaction to be justified by the same abstract history.

Furthermore, the original definition does not have any transformations, i.e., an aborted transaction cannot be used in the justification of another aborted transaction. By our definition, an aborted transaction, transformed into a committed transaction, can be used in explaining another transaction.

**Definition 6.7** A history $H$ is in the **sequential consistency relation** with a history $S$, denoted $H \sqsubseteq_{\text{seq}} S$, if for every thread $t \in \text{ThreadID}$, $H|_t = S|_t$.

**Definition 6.8** A history $H_\psi = H'_1 \psi$ is a **vwc possible past** of a history $H = H'_1$, where $\psi$ is a response action, if:

(i) $H'_1$ is a subsequence of $H$;

(ii) $H_\psi$ is comprised of the transaction of $\psi$ and some of the visible transactions in $H$:

$$\text{tx}(H_\psi) \subseteq \{\text{txof}(\psi, H)\} \cup \text{visible}(H).$$

(iii) for a thread $t \in \text{ThreadID}$, for every transaction $T \in H_\psi|_t$, out of all transactions preceding $T$ in the real-time order in $H|_t$, the history $H_\psi|_t$ includes exactly the committed ones preceding $T$:

$$\forall t \in \text{ThreadID}. \forall T \in \text{tx}(H_\psi|_t). \forall T' \in \text{tx}(H|_t). T' \prec_H T \implies (T' \in \text{tx}(H_\psi) \iff T' \in \text{committed}(H)).$$

We denote the set of vwc possible pasts of $H$ by $\text{VWCpast}(H)$.

**Definition 6.9** A history $H_\psi^c$ is a **completed vwc past** of a history $H = H'_1$, if $H_\psi^c$ is a commit justification of a history obtained from a vwc possible past $H'_1 \psi$ of $H$ by replacing all the aborted actions in $H'_1$ by committed actions. The set of completed vwc pasts of $H$ is denoted $\text{cVWCpast}(H)$:

$$\text{cVWCpast}(H_1) = \{H_\psi^c | \exists H'_1. H'_1 \psi \in \text{VWCpast}(H_1) \land H_\psi^c \in \text{justify}(\text{com}(H'_1)(\psi))\},$$

where $|\text{com}(H'_1)| = |H'_1|$ and $\text{com}(H'_1)(i) = (\text{if } (H'_1)(i) = (a, t, \text{aborted}) \text{ then } (a, t, \text{committed}) \text{ else } H'_1(i))$.

**Definition 6.10** A history $H$ is in the **VWC relation** with TM $T$, denoted $H \sqsubseteq_{\text{vwc}} T$, if:

(i) $\exists H' \in \text{removecp}(H). \exists H^c \in \text{cpcomplete}(H'|_{\text{live}}). \exists S \in T. H^c|_{\text{com}} \sqsubseteq_{\text{vwc}} S$;

(ii) for every response action $\psi$ such that such that it is not a committed or aborted action and $H = H_1 \psi H_2$: $\exists H_\psi^v \in \text{cVWCpast}(H_1). \exists S_\psi \in T. H_\psi^v \sqsubseteq_{\text{seq}} S_\psi$. 

71
A TM $T_C$ is in the **VWC relation** with a TM $T_A$, denoted by $T_C \sqsubseteq_{\text{vwc}} T_A$, if $\forall H \in T_C. H \sqsubseteq_{\text{vwc}} T_A$. A TM $T_C$ is VWC if $T_C \sqsubseteq_{\text{vwc}} T_{\text{atomic}}$.

Our notion of VWC is weaker than both TMS1 (and hence STMS) and OVWC. Figure 6.1(c) shows a history that is VWC, but not TMS1. TMS1 mandates the inclusion of $T_2$ in the completed possible past for the response of read operation on $x$ by $T_1$. It also mandates the preservation of the real-time order between transactions $T_2$ and $T_1$, which makes this response not legal in the abstract history. VWC, however, allows the exclusion of $T_2$ to justify the response, which makes the abstract history legal and this history VWC.

### 6.3.1 Sufficiency of VWC

VWC, like STMS, allows every operation in a live or aborted transaction to be justified by a separate abstract history. However, it places different constraints on the choice of abstract histories, which do not take into account the real-time order between actions. Because of this, VWC does not imply observational refinement under the semantics with rollback for our programming language: taking into account the real-time order is necessary when threads can communicate via global variables outside transactions. However, this does not rule out the viability of VWC and related notions as a consistency condition for TM. We show that VWC implies *weak observational refinement* for a programming language in which local variables are rolled back on transaction abort.

**Definition 6.11** A TM $T_C$ weakly observationally refines a TM $T_A$, denoted $T_C \preceq_{\text{wRB}} T_A$, if for every program $P$, state $s$ and trace $\tau \in \llbracket P, T_C \rrbracket_{\text{RB}}(s)$ there exists a trace $\tau' \in \llbracket P, T_A \rrbracket_{\text{RB}}(s)$ such that $\tau = (\cdot, \cdot, \text{fault}) \iff \tau' = (\cdot, \cdot, \text{fault})$.

Let us fix a program $P = C_1 \| \ldots \| C_m$, state $s$ and TMs $T_C$ and $T_A$ such that $T_C \sqsubseteq_{\text{vwc}} T_A$. We need to show that a trace from $\llbracket P, T_C \rrbracket_{\text{RB}}(s)$ with a fault inside a live transaction can be transformed into a trace with the fault from $\llbracket P, T_A \rrbracket_{\text{RB}}(s)$.

The following lemma describes the first and foremost step of this transformation: it converts a trace $\tau \in \llbracket P \rrbracket_{\text{RB}}(s)$ with a live transaction into another trace from $\llbracket P \rrbracket_{\text{RB}}(s)$ that contains the same live transaction, but whose history of non-aborted transactions is a given element of $c_{\text{VWCpast}}(\text{history}(\tau))$. In other words, this establishes that the live transaction cannot notice the changes of the status of other transactions done by $c_{\text{VWCpast}}$.

**Lemma 6.5** Let $\tau = \tau_1 \psi \tau_2 \in \llbracket P \rrbracket_{\text{RB}}(s)$ be such that $\psi$ is a response action by thread $t_0$ that is not a committed or aborted action and $\tau_2$ contains only primitive actions by thread $t_0$. Consider $H_\psi \in c_{\text{VWCpast}}(\text{history}(\tau))$. There exists $\tau_\psi \in \llbracket P \rrbracket_{\text{RB}}(s)$ such that $\text{history}(\tau_\psi)|_{\text{abortedtx}} = H_\psi$ and $\tau_\psi|_{t_0} = \tau|_{t_0}$.
Proof: We first show how to construct $\tau_\psi$ and then prove that it satisfies the required properties. Let $\text{history}(\tau) = H_1 \psi$. Since $H_1^c \psi \in \text{cVWCpast}(H)$, by Definition 6.9 there exist histories $H_1', H_1''$, and $H^{cc}$ such that

$$H_1' \psi \in \text{VWCpast}(H_1 \psi) \land H_1'' = \text{com}(H_1') \land H_1^c \psi = H_1'' \psi H^{cc} \in \text{justify}(H_1'' \psi).$$

We show that it is possible to construct the required trace by erasing certain suffixes of every thread and therefore getting rid of the actions that could be sensitive to the changes of transaction status. We now explain how to truncate $\tau$ consistently.

Our idea is, for every thread other than $t_0$, to erase all its actions that follow the last of its transactions included into $H_1' \psi$. Formally, $\tau_{t_0} = \tau|_{t_0}$ and for every thread $t \neq t_0$, let $\tau_t$ denote the prefix of $\tau_t$ that ends with the last TM interface action of $t$ in $H_1' \psi$, or $\varepsilon$ if no such action exists. We then let the truncated trace $\tau'$ be the subsequence of $\tau$ such that $\tau'|_t = \tau_t$ for every $t$.

To construct $\tau_\psi$ from $\tau'$, we mirror the transformations of $H_1'$ into $H_1''$ and $H_1^c$. Let $\tau''$ be defined by $|\tau''| = |\tau'|$ and

$$\tau''(i) = (\text{if } (\tau'(i) = (a, t, \text{aborted}) \land \tau'(i) \in H_1')) \text{ then } (a, t, \text{committed}) \text{ else } \tau'(i)).$$

Then we let $\tau_\psi = \tau'' H^{cc}$.

We first prove that $\tau_\psi|_{t_0} = \tau|_{t_0}$. Let $T = \text{txof}(\psi, H_1 \psi)$; then by Definition 6.8(ii), $T \in H_1' \psi$. Hence, by Definition 6.8(iii) we have

$$\forall T' \in \text{tx}(H_1' \psi)|_{t_0}. T' \prec_{H_1' \psi} T \iff T' \prec_{H_1 \psi} T \land T' \in \text{committed}(H_1 \psi), \quad (6.1)$$

so that $(H_1' \psi)|_{t_0}$ does not contain aborted transactions and $\tau''|_{t_0} = \tau'|_{t_0} = \tau|_{t_0}$. Besides, $H^{cc}|_{t_0} = \varepsilon$ and, hence, $\tau_\psi|_{t_0} = \tau''|_{t_0} = \tau|_{t_0}$.

We now sketch the proof that $\tau_\psi \in \lfloor P \rfloor_{\text{RB}}(s)$, appealing to the intuitive understanding of the programming language semantics. To this end, we show that $\tau'$ and then $\tau''$ belong to $\lfloor P \rfloor_{\text{RB}}(s)$. We start by analyzing how the trace $\tau|_t$ is truncated to $\tau_t$ for every thread $t \neq t_0$. The transformation from $\tau$ to $\tau'$ can be viewed as erasing some suffixes of threads. Furthermore, as we noted in Section 2.4, $\lfloor P \rfloor_{\text{RB}}(s)$ includes incomplete program computations. This allows us to conclude that $\tau' \in \lfloor P \rfloor_{\text{RB}}(s)$.

The only aborted transaction included into $H_1' \psi$, by a thread $t$, can only be the last one in $\tau'|_t$. We also established that $(H_1' \psi)|_{t_0}$ does not contain aborted transactions. Hence, transactions in $\tau'$ whose status is changed from aborted to committed when switching to $\tau''$ do not have any actions following them in $\tau'$. Furthermore, $\lfloor P \rfloor_{\text{RB}}(s)$ allows committing or aborting transactions arbitrarily. This allows us to conclude that $\tau'' \in \lfloor P \rfloor_{\text{RB}}(s)$. For the same reason, we get $\tau_\psi \in \lfloor P \rfloor_{\text{RB}}(s)$.

Finally, we show that $\text{history}(\tau_\psi)|_{\text{abortedtx}} \prec_{\text{abortedtx}} = H_1^c \psi$. It is sufficient to show that $\text{history}(\tau'')|_{\text{abortedtx}} = H_1'' \psi$; since $\tau_\psi = \tau'' H^{cc}$ and $H^{cc}$ contains only committed ac-
tions, this would imply

\[
\text{history}(\tau_\psi) \mid _{\neg \text{abortedtx}} = \text{history}(\tau'' H^cc) \mid _{\neg \text{abortedtx}} = \text{history}(\tau'') \mid _{\neg \text{abortedtx}} H^cc = H'_1 \psi H^cc = H^c_\psi.
\]

By the choice of \(\tau_t\) for \(t \neq t_0\), every transaction in \((H'_1 \psi)\mid_t\) is also in \(\tau_t\). Hence, \(H'_1 \psi\) is a subsequence of \(\text{history}(\tau'_t)\). By the definition of \(\tau''\) and \(H'_1\), \(H'_1 \psi\) is a subsequence of \(\text{history}(\tau'')\). Then since \(H'_1 \psi\) does not contain aborted transactions, \(H'_1 \psi\) is a subsequence of \(\text{history}(\tau'') \mid _{\neg \text{abortedtx}}\).

Thus, to prove \(\text{history}(\tau'') \mid _{\neg \text{abortedtx}} = H'_1 \psi\) it remains to show that every non-aborted transaction in \(\text{history}(\tau'')\) is in \(H'_1 \psi\). Since the construction of \(\tau''\) from \(\tau'_t\) changes the status of only those transactions that belong to \(H'_1 \psi\), it is sufficient to show that every non-aborted transaction in \(\text{history}(\tau'_t)\) is in \(H'_1 \psi\). We consider the following cases, depending on the thread \(t\) the transaction is by.

- \(t \neq t_0\) is such that \(\tau'_t \mid \tau_t = \tau_t \neq \varepsilon\). Let \(\psi_t\) be the last action in \(\tau_t\). Let \(T = \text{txof}(\psi_t, H_1 \psi)\). By the choice of \(\tau_t\) we have \(T \in H'_1 \psi\}; then by Definition 6.6(iii) we get (6.1). Since any transaction \(T'\) in \(\text{history}(\tau'_t)\) is either \(T\) or is such that \(T' \prec (H_1 \psi)\mid_t T\), this implies the required.

- \(t = t_0\). Let \(T = \text{txof}(\psi, H_1 \psi) \in H'_1 \psi\). Then by Definition 6.6(iii) we get (6.1).

Since any transaction \(T'\) in \(\text{history}(\tau'\mid_{t_0})\) is either \(T\) or is such that \(T' \prec (H_1 \psi)\mid_{t_0} T\), this implies the required.

This concludes the proof that \(\text{history}(\tau'') \mid _{\neg \text{abortedtx}} = H'_1 \psi\).

Definition 6.10 matches a history of \(T_C\) with one of \(T_A\) using the sequential consistency relation, possibly after transforming the former with \(cVWC_{\text{past}}\). The following lemma shows that the sequential consistency relation implies observational refinement with respect to thread-local trace projections.

**Lemma 6.6** For any well-formed histories \(H\) and \(S\):

\[
H \sqsubseteq_{\text{seq}} S \implies \forall \tau_H \in \text{WfTraces. history}(\tau_H) = H \implies \exists \tau_S \in \text{WfTraces. history}(\tau_S) = S \land \forall t. \tau_H\mid_t = \tau_S\mid_t.
\]

**Proof:** Consider \(H, S\) and \(\tau_H\) such that \(H \sqsubseteq_{\text{seq}} S\) and \(\text{history}(\tau_H) = H\). Note that \(|H| = |S|\). To obtain the desired trace \(\tau_S\), we inductively construct a sequence of traces \(\tau^i \in \text{WfTraces}, i = 0..|S|\) with histories \(H^i = \text{history}(\tau^i) \in \text{WfHistory}\) such that

\[
H^i\mid_i = S\mid_i; \quad H^i \sqsubseteq_{\text{seq}} S; \quad \forall t. \tau_H\mid_t = \tau^i\mid_t.
\] (6.2)

We then let \(\tau_S = \tau^{|S|}\), so that for every thread \(t\), \(\tau_H\mid_t = \tau^{|S|}\mid_t\) and \(\text{history}(\tau^{|S|}) = H^{|S|} = H^{|S|}\mid_{|S|} = S\mid_{|S|} = S\), as required. Note that the condition \(H^i \sqsubseteq_{\text{seq}} S\) in (6.2) is not used to establish the required properties of \(\tau_S\); we add it so that the induction goes through.
In the base case, \( \tau^0 = \tau_H \), so that \( H^0 = H \) and all the requirements in (6.2) hold trivially.

For the induction step, assume we have well-formed histories \( H^i \) and \( S \) and a well-formed trace \( \tau^i \) such that

\[
\text{history}(\tau^i) = H^i; \quad H^i|_i = S|_i; \quad H^i \sqsubseteq_{\text{seq}} S; \quad \forall t. \tau_H|_t = \tau^i|_t.
\]

We will show that there is a history \( H^{i+1} \in \text{WfHistory} \) and a trace \( \tau^{i+1} \in \text{WfTraces} \) such that

\[
\text{history}(\tau^{i+1}) = H^{i+1}; \quad H^{i+1}|_{i+1} = S|_{i+1}; \quad H^{i+1} \sqsubseteq_{\text{seq}} S; \quad \forall t. \tau_H'|_t = \tau^{i+1}|_t.
\]

Let \( S = S_1 \psi S_2 \), where \( |S_1| = i \). By assumption, \( \text{history}(\tau^i)|_i = H^i|_i = S|_i = S_1 \). Thus, for some traces \( \tau_1 \) and \( \tau_2 \), we have \( \tau^i = \tau_1 \tau_2 \), where \( \tau_1 \) is the minimal prefix of \( \tau^i \) such that \( \text{history}(\tau_1) = S_1 \). We also have \( H^i \sqsubseteq_{\text{seq}} S \), and, hence, \( S \) is a permutation of \( H^i \) preserving the order of actions within a thread (Definition 6.7).

Hence, for some traces \( \tau_3 \) and \( \tau_4 \), we have

\[
\text{history}(\tau) = \psi, \quad \tau_2 = \tau_3 \psi \tau_4, \quad \tau^i = \tau_1 \tau_2 = \tau_1 \tau_3 \psi \tau_4.
\]

Let \( \psi = (\_, t, \_) \). We note that, since \( \sqsubseteq_{\text{seq}} \) preserves the order of actions within a thread and \( \text{history}(\tau_1) = S_1 \), we have \( \text{history}(\tau_3|_t) = \epsilon \).

Let \( \tau^{i+1} = \tau_1 (\tau_3|_t) \psi (\tau_3|_{-t}) \tau_4 \) and \( H^{i+1} = \text{history}(\tau^{i+1}) \). Intuitively, \( \tau^{i+1} \) is obtained from \( \tau^i = \tau_1 \tau_3 \psi \tau_4 \) by moving all the actions in \( \tau_3 \) performed by thread \( t \), together with \( \psi \), to the position right after \( \tau_1 \).

Since \( \text{history}(\tau_3|_t) = \epsilon \), \( \text{history}(\tau_1) = S_1 \) and \( |S_1| = i \), we get:

\[
H^{i+1}|_{i+1} = (\text{history}(\tau_1 (\tau_3|_t) \psi (\tau_3|_{-t}) \tau_4))|_{i+1} = S_1 \psi = S|_{i+1},
\]

as required. We also have:

\[
\tau^{i+1}|_t = (\tau_1 (\tau_3|_t) \psi (\tau_3|_{-t}) \tau_4)|_t = (\tau_1|_t) (\tau_3|_t) \psi (\tau_4|_t) = (\tau_1 \tau_3 \psi \tau_4)|_t = \tau^i|_t;
\]

\[
\tau^{i+1}|_{-t} = (\tau_1 (\tau_3|_t) \psi (\tau_3|_{-t}) \tau_4)|_{-t} = (\tau_1|_{-t}) (\tau_3|_{-t}) (\tau_4|_{-t}) = (\tau_1 \tau_3 \psi \tau_4)|_{-t} = \tau^i|_{-t}.
\]

Hence, for any thread \( t' \), we have \( \tau^i|_{t'} = \tau^{i+1}|_{t'} \), and \( H^{i+1}|_{t'} = H^i|_{t'} = S|_{t'} \), giving \( H^{i+1} \sqsubseteq_{\text{seq}} S \).

The following lemma shows that the trace \( \tau' \) resulting from the transformation in Lemma 6.6 can be produced by a program \( P \) if so can the original trace \( \tau \).

**Lemma 6.7** Let \( \tau \) be a trace such that \( \tau \in [P]_{\text{RB}}(s) \) and let \( S \) be such that \( \text{history}(\tau) \sqsubseteq_{\text{seq}} S \). Then there exists \( \tau' \in [P]_{\text{RB}}(s) \) such that \( \text{history}(\tau') = S \) and \( \forall t. \tau'|_t = \tau|_t \).

75
**Proof:** Since history($\tau$) $\subseteq_{seq} S$ and $\tau \in [P]_{RB}(s)$, Lemma 6.6 implies that there exists a trace $\tau' \in WfTraces$ such that history($\tau'$) = $S$ and $\forall t. \tau|_t = \tau'|_t$.

Since $\tau \in [P]_{RB}(s)$, for some $\tau''$ we have $\tau\tau'' \in Tr'(P)$. This implies $(\tau\tau'')|_t \in Tr'_t(C_t)$ for any thread $t$. Since $\tau|_t = \tau'|_t$, we have $(\tau'\tau'')|_t \in Tr'_t(C_t)$. Then by the definition of $Tr(P)$ in Figure 2.1, we get $\tau' \in Tr(P)$.

Since $\tau \in [P]_{RB}(s)$, we also have $eval_{RB}(s, \tau) \neq \emptyset$. Now we show that $eval_{RB}(s, \tau') \neq \emptyset$, which along with the fact that $\tau' \in Tr(P)$ implies $\tau' \in [P]_{RB}(s)$, which completes the proof.

The proof goes by case analysis depending on whether $\tau$ contains a fault action or not.

**Case I.** Let us assume first that $\tau$, and hence $\tau'$, does not contain a fault action. We inductively construct a sequence of traces $\tau^i \in WfTraces$, $i = 0..|\tau'|$ such that

$$\tau^0|_i = \tau'|_i; \quad \forall t. \tau^i|_t = \tau'|_t; \quad eval_{RB}(s, \tau^i) = eval_{RB}(s, \tau) \neq \emptyset. \tag{6.3}$$

Then for $i = |\tau'|$ we get $\tau^i = \tau'$, which implies the lemma.

For the base case, let $\tau^0 = \tau$, and all the requirements in (6.3) hold trivially.

For the induction step, assume that a trace $\tau^i$ satisfying (6.3) has been constructed. Let $\tau' = \tau_1\varphi \tau_2$, where $|\tau_1| = i$, and $\varphi = (\_ , t , \_ )$. By assumption, $\tau^i|_i = \tau'|_i$ and, since $\tau^i|_t = \tau'|_t$, for some traces $\tau'_2$ and $\tau''_2$, we get $\tau^i = \tau_1\tau'_2\varphi\tau''_2$, where $\tau'_2$ does not contain any actions by thread $t$. Let $\tau^{i+1} = \tau_1\varphi\tau'_2\tau''_2$; then $\tau^{i+1}|_{i+1} = \tau'|_{i+1}$. We now show that $\forall t. \tau^{i+1}|_t = \tau^i|_t$ and $eval_{RB}(s, \tau^{i+1}) = eval_{RB}(s, \tau^i)$, which implies the lemma.

First note that $\tau^i|_t = \tau^{i+1}|_t$ for any thread $t'$ since $\tau'_2$ does not contain any actions by thread $t$. Next, consider the case when the action $\varphi$ in $\tau^i$ is non-transactional. Since $\tau^i|_t = \tau'|_t$, so is the corresponding action $\varphi$ in $\tau'$. Assume that some action $\varphi' = (\_ , t' , \_ )$ in the subtrace $\tau'_2$ of $\tau'$ is non-transactional as well, where $t' \neq t$. Let $\varphi'$ be the $j$-th action by thread $t'$ in $\tau'$. Since $\tau^i|_t = \tau^i|_t$, $\varphi'$ is also the $j$-th action by thread $t'$ in $\tau'$ and is non-transactional in this trace. But then $\varphi'$ has to be in the subtrace $\tau'_2$ of $\tau'$ and thus follow $\varphi$.

Given the properties of $\tau'_2$ established above, by applying Proposition 2.1 repeatedly we get that $eval_{RB}(s', \varphi \tau'_2) = eval_{RB}(s', \tau'_2 \varphi)$ for any state $s'$. Hence,

$$eval_{RB}(s, \tau^{i+1}) = eval_{RB}(s, \tau_1\varphi\tau'_2\tau''_2) = eval_{RB}(s, \tau_1\tau'_2\varphi\tau''_2) = eval_{RB}(s, \tau^i).$$

**Case II.** We now consider the case when $\tau = \tau''(\_ , t , fault)$, i.e., $\tau$ contains a fault action by some thread $t$, and it is the last action in $\tau$. Since $\tau|_t = \tau'|_t$, we get $\tau|_t = \tau''|_t(\_ , t , fault)$. By Definition 2.2(ii), $\tau' = \tau''(\_ , t , fault)$, i.e., $\tau'$ cannot have an action after $(\_ , t , fault)$, and therefore $eval_{RB}(s, \tau') = eval_{RB}(s, \tau''(\_ , t , fault)) = i' = eval_{RB}(s, \tau)$.

Finally, Definition 6.10 matches only histories of committed transactions, but the histories of the traces in Lemma 6.7 also contain aborted transactions. Fortunately, the
following lemma allows us to add empty aborted transactions into the abstract history while preserving the sequential consistency relation.

**Lemma 6.8** Let $H$ be a well-formed history where all aborted transactions abort immediately and $S$ be such that $H \not\sqsubseteq_{\text{seq}} S$. There exists a well-formed history $S' \in \text{addab}(S)$ such that $H \sqsubseteq_{\text{seq}} S'$.

**Proof:** Let $n$ be the number of aborted transactions in $H$. To construct the desired $S'$, we inductively construct a sequence of histories $S_i$, $i = 0..n$ such that

$$\lvert \text{aborted}(S_i) \rvert = i; \quad S_i \in \text{addab}(S); \quad \{ \psi \mid \psi \in S_i \} \subseteq \{ \psi \mid \psi \in H \}; \quad \forall t, \forall \psi_1, \psi_2 \in S_i[t], \psi_1 \prec_H \psi_2 \implies \psi_1 \prec_{S_i} \psi_2. \quad (6.4)$$

We then let $S' = S_n$, so that $H \sqsubseteq_{\text{seq}} S'$.

For $i = 0$, we take $S_0 = S$, and all the requirements in (6.4) hold vacuously. Assume a history $S_i$ satisfying (6.4) was constructed; we get $S_{i+1}$ from $S_i$ by the following construction. Let $H = H_1 \psi_b H_2 \psi_a H_3$, where $\psi_b = (\_, t, \text{txbegin}), \psi_a = (\_, t, \text{aborted})$, $H_2 \mid t = \varepsilon$ and

$$\neg \exists \psi'. \psi' = (\_, \_, \text{txbegin}) \in H_1 \land \text{txof}(\psi', H) \in \text{aborted}(H) \land \psi' \notin S_i.$$

That is, out of all aborted transactions in $H$ that are not in $S_i$, $\psi_b \psi_a$ is the one with the earliest $\text{txbegin}$.

If $H_1$ does not contain any action by thread $t$, then let $S_{i+1} = \psi_b \psi_a S_i$. We only need to show that for any $\psi' \in S_i[t]$ we have $\psi' \prec_H \psi_b \implies \psi' \prec_{S_{i+1}} \psi_b$ and $\psi_a \prec_H \psi' \implies \psi_a \prec_{S_{i+1}} \psi'$. The latter holds by the construction of $S_{i+1}$. To show the former, observe that, $H_1$ does not contain actions by thread $t$. Hence, we cannot have $\psi' \prec_H \psi_b$ for any $\psi'$.

The rest of the proof deals with the case that $H_1$ contains an action by thread $t$. Let $\psi$ be the last committed or aborted action by $t$ in $S_i$ that is also in $H_1$ and let $S_i = S' \psi S''$. We then let $S_{i+1} = S' \psi_b \psi_a S''$.

We again need to show that for any $\psi' \in S_i[t]$ we have $\psi' \prec_H \psi_b \implies \psi' \prec_{S_{i+1}} \psi_b$ and $\psi_a \prec_H \psi' \implies \psi_a \prec_{S_{i+1}} \psi'$.

Assume $\psi' \prec_H \psi_b$ for some $\psi' \in S_i[t]$; then $\psi' \in H_1$. By the choice of $\psi_b$ and $\psi_a$, all the actions in $H_1$ by $t$ are in $S_i$, and by the choice of $\psi$, all such actions are in $S' \psi$, and hence, $\psi' \prec_{S_{i+1}} \psi_b$.

Now assume $\psi_a \prec_H \psi'$ for some $\psi' \in S_i[t]$; then $\psi' \in H_3$. Since $\psi \prec_H \psi_a$, we have $\psi \prec_H \psi'$. Hence, $\psi \prec_{S_i} \psi'$, i.e., $\psi' \in S''$, which implies $\psi_a \prec_{S_{i+1}} \psi'$.

**Theorem 6.9** Let $\mathcal{T}_C$ and $\mathcal{T}_A$ be transactional memories. If $\mathcal{T}_A$ satisfies CLP1, then $\mathcal{T}_C \sqsubseteq_{\text{vwc}} \mathcal{T}_A \implies \mathcal{T}_C \sqsubseteq_{\text{vwc}} \mathcal{R}_B \mathcal{T}_A$.

**Proof:** Assume $\mathcal{T}_C \sqsubseteq_{\text{vwc}} \mathcal{T}_A$. Consider $\tau \in [P, \mathcal{T}_C]\mathcal{R}_B(s)$ and let $H = \text{history}(\tau)$.
Let $\tau = \tau_1 \psi_2 \chi$, where $\chi = (., t_0, \text{fault})$ is transactional and $\psi$ is the last TM interface action by thread $t_0$. Then $\tau_2|_{t_0}$ consists of transactional actions. Hence, $\tau = \tau_1 \psi_2|_{t_0} \phi \in [P, T_C]_{\text{RB}}(s)$. By our assumption, $T_C \subseteq \text{vwc} T_A$. Then there exists $H^c_\psi \in \text{cVWCpast}(\text{history}(\tau))$ and $S \in T_A$ such that $H^c_\psi \subseteq \text{seq} S$. By Lemma 6.5, for some trace $\tau_\psi$ we have $H_\psi \subseteq \text{RT} S$ and $\tau_\psi|_{t_0} = \tau|_{t_0}$. By Proposition 4.6, $\tau_\psi|_{\text{abortact}} \subseteq [P]_{\text{RB}}(s)$. Using Lemma 6.8, we get a history $S'$ such that $\text{history}(\tau_\psi|_{\text{abortact}}) \subseteq \text{seq} S'$ and $S' \in \text{addab}(S)$. Since $S \in T_A$ and $T_A$ is closed under immediate aborts, we get $S' \in T_A$. Hence, by Lemma 6.7, for some $\tau' \in [P, T_A]_{\text{RB}}(s)$ we have $\tau'|_{t_0} = \tau_\psi|_{t_0} = \tau|_{t_0} = \chi$, as required.

### 6.4 Transactional Memory Specification 2 (TMS2)

For completeness, the following definition formalizes the notion of TMS2 in our framework.

**Definition 6.12** A history $H$ is in the TMS2 relation with a history $S$, denoted $H \equiv_{\text{tms2}} S$, if $H \equiv_{\text{RT}} S$ and

$$
\forall \psi_1, \psi_2 \in H. \psi_1 = H(i) = S(i') = (., \cdot, \text{committed}) \land \psi_2 = H(j) = S(j') = (., \cdot, \text{txcommit}).
$$

$$
\text{txof}(\psi_2, H) \in \text{committed}(H) \land i < j \implies i' < j'.
$$

A TM $T_C$ is in the TMS2 relation with a TM $T_A$, denoted by $T_C \equiv_{\text{tms2}} T_A$, if $\forall H \in T_C. \exists S \in T_A. H \equiv_{\text{tms2}} S$. A TM $T_C$ is **TMS2** if $T_C \subseteq_{\text{tms2}} T_{\text{atomic}}$. A history $H \in T_C$ is **TMS2** if $T_C$ is TMS2.

Informally, a history $H$ is TMS2 if there is a non-interleaved, complete and legal history $S$ that is in real-time relation with $H$, and preserves the order of commit operations in $H$.

Figure 6.1(d) shows a history that is opaque, and hence STMS, but not TMS2. Since the history is complete, its suffix completion $H_c$ is the history itself. It is easy to see that the complete, non-interleaved history $S = T_2 T_1$ is legal and $H_c \equiv_{\text{RT}} S$, making the history opaque. However, since $T_1$ is a committed transaction and it is completed before the invocation of txcommit by the transaction $T_2$, TMS2 restricts the transaction $T_1$ to precede $T_2$ in $S$, which makes it not legal, and hence the history is not TMS2.
Chapter 7

STMS is a safety property

The set of histories specified by the consistency criterion constitutes a safety property, as defined by Owicki and Lamport [32], Alpern and Schneider [2] and refined by Lynch [29]: if it is non-empty, prefix-closed and limit-closed.

Definition 7.1 A property $P$ is prefix-closed if for every history $H \in P$, every prefix $H'$ of $H$ is also in $P$.

Definition 7.2 A property $P$ is limit-closed if for every infinite sequence of finite histories $H_0, H_1, \ldots$ such that for every $i$, $H_i \in P$ and $H_i$ is a prefix of $H_{i+1}$, the limit of the sequence is also in $P$.

In this chapter, we show that, under the assumption that every transaction eventually completes, STMS is a safety property, and to prove that a TM implementation is STMS, it suffices to prove that all its finite histories are STMS.

Notice that the set of histories in a TM $T$ is, by construction, prefix-closed. Therefore, every infinite history of $T$ is the limit of an infinite sequence of ever-extending finite histories of $T$. Thus, to prove that $T$ satisfies a safety property $P$, it is enough to show that all finite histories of $T$ are in $P$. Indeed, limit-closure of $P$ then implies that every infinite history of $T$ is also in $P$.

The following lemma shows that STMS is prefix-closed, if the abstract TM satisfies the following closure property.

A history $H'$ is a thread-prefix of a history $H$ if for every thread $t$, $H'|_t$ is a prefix of $H|_t$. We denote by $t\text{prefix}(H)$ the set of all thread-prefixes of $H$.

CLP5 A TM $T$ is closed under removing thread suffix if whenever $H \in T$, we also have $H' \in T$ for any history $H' \in t\text{prefix}(H)$.

Note that CLP5 is not satisfied by the TM specification $T_{\text{atomic}}$ defined in Section 2.3, as removing a suffix of a thread can make the history not legal.

Lemma 7.1 Let $H$ be a well-formed history such that $H \subseteq_{\text{stms}} T$, for an abstract TM $T$ that satisfies CLP5. Then, for every integer $i$, $H|_i \subseteq_{\text{stms}} T$. 

79
\textbf{Proof:} Let $H$ be a well-formed history such that $H \subseteq_{\text{stms}} T$, for a TM $T$. By Definition 3.8(i), there exists a history $H' \in \text{removecp}(H)$, a history $H^c \in \text{justify}(H'|_{\neg\text{live}})$ and a history $S \in T$ such that $H^c \subseteq_{\text{RT}} S$.

We construct a history $H^c_i$ and a history $S_i \in T$ such that $H^c_i \subseteq_{\text{RT}} S_i$. Let $H''$ be a subsequence of $H^c_i$ such that $H'' \in \text{removecp}(H_i)$ and for every transaction $T \in \text{copending}(H_i)$, $T \in \text{tx}(H'')$ only if $T(\_\_\_, \text{committed}) \in \text{committed}(H^c)$. Let $H^c_i \in \text{justify}(H''|_{\neg\text{live}})$ and let $S_i$ be the projection of $S$ on all actions in $H^c_i$.

Since $H^c \subseteq_{\text{RT}} S$, for every thread $t$, $H^c|_t = S|_t$, which further implies, by the construction of $S_i$ that for every thread $t$, $H^c_i|_t = S_i|_t$. Furthermore, the real-time order between actions in $H^c_i$ is preserved in $S_i$ and hence, the real-time order in $H^c_i$ is preserved in $S_i$. This is true since for any pair of actions $\psi$ and $\psi'$ in $H^c_i$ and $S_i$,

$$\psi \prec_{H^c_i} \psi' \implies \psi \prec_{H^c} \psi' \implies \psi \prec_S \psi' \implies \psi \prec_{S_i} \psi'.$$

Therefore, $H^c_i \subseteq_{\text{RT}} S_i$. Since $S_i$ is a projection of $S$, for every thread $t$, we can write $S|_t = (S_i)|_t S_i$, for some history $S_i$. Since $T$ satisfies CLP5 and $S \in T$, $S_i \in T$.

Let $\psi$ be a response action in $H$, that is not a committed or aborted action. By Definition 3.8(ii), there is a history $H^c_\psi \in \text{cSTMSpast}(H|_\psi)$ and a history $S_\psi \in T$ such that $H^c_\psi \subseteq_{\text{RT}} S_\psi$. Since $H|_i$ is a prefix of $H$, $(H|_i)|_\psi = H|_\psi$, and hence, $H^c_\psi \in \text{cSTMSpast}((H|_i)|_\psi)$, implying that $H^c_\psi \subseteq_{\text{RT}} S_\psi$ for some $S_\psi \in T$. This completes the proof. \hfill $\blacksquare$

The following lemma shows that if a history $H$ is in the real-time relation with a history $S$, then every prefix of $H$ is in real-time relation with a thread-prefix of $S$.

**Lemma 7.2** Let $H, S \in \text{WfHistory}$ such that $H \subseteq_{\text{RT}} S$. Then, for every integer $i$, there is a history $S_i$ such that $S_i$ is a subsequence of $S$ and $H|_i \subseteq_{\text{RT}} S_i$.

**Proof:** Let $S_i$ be the projection of $S$ on all actions in $H|_i$. Write $H = H|_i H'$, for some history $H'$. Since $H \subseteq_{\text{RT}} S$, for every thread $t$, we have $H|_t = S|_t = (H|_i)|_t H'|_t$. Since $S_i$ is the projection of $S$ on all actions in $H|_i$, $S_i$ is a subsequence of $S$.

Furthermore, the real-time order between actions in $H$ is preserved in $S$, and hence, the real-time order in $H|_i$ is preserved in $S_i$. This is true since for any pair of actions $\psi$ and $\psi'$ in $H|_i$ and $S_i$, if $\psi \prec_{H|_i} \psi'$ then $\text{txof}(\psi, H|_i)$ is complete, which implies

$$\psi \prec_{H|_i} \psi' \implies \psi \prec_H \psi' \implies \psi \prec_S \psi' \implies \psi \prec_{S_i} \psi'.$$

Therefore, $H|_i \subseteq_{\text{RT}} S_i$, completing the proof. \hfill $\blacksquare$

Proving that STMS is limit-closed relies on König’s Path Lemma [27], which is formulated as follows. Let $G$ be a rooted directed graph and let $v_0$ be the root of $G$. $G$ is \textbf{connected} if every vertex in $G$ is reachable from $v_0$. $G$ is \textbf{finitely branching} if every vertex in $G$ has a finite out-degree. $G$ is \textbf{infinite} if it has infinitely many vertices.
**Lemma 7.3 (König’s Path Lemma [27])** If $G$ is an infinite connected finitely branching rooted directed graph, then $G$ contains an infinite sequence of distinct vertices $v_0, v_1, \ldots$, such that $v_0$ is the root, and for every $i \geq 0$, there is an edge from $v_i$ to $v_{i+1}$.

**Theorem 7.4** Under the restriction that in any infinite history $H$, every transaction $T \in \text{tx}(H)$ is complete, opacity is a limit-closed property.

**Proof:** Let $H_1, \ldots, H_i, \ldots$ be well-formed histories such that for all integers $i$, $H_i$ is a prefix of $H_{i+1}$, and $H_i \sqsubseteq_{RT} S_i$, for some non-interleaved history $S_i$. Let $H$ be the infinite history that is the limit of the histories $H_i$. Our goal is to show that there is a non-interleaved history $S$ such that $H \sqsubseteq_{RT} S$.

Let $\text{projcom}(S_j, H_i)$, where $j \geq i$, denote the projection of $S_j$ on all actions in $H_i$ such that for every action $\psi$ in $H_i$, $\text{txof}(\psi, H_i) \in \text{completed}(H_i)$. Informally, $\text{projcom}(S_j, H_i)$ is the projection of $S_j$ on all complete transactions in $H_i$.

Let $H_0 = \varepsilon$. We construct a directed graph $G = (V, E)$ as follows. The vertices of $G$ are all tuples $(H_i, S_i)$, such that $H_i \sqsubseteq_{RT} S_i$, where $S_i$ is a non-interleaved history. There is a directed edge from $(H_i, S_i)$ to $(H_j, S_j)$ in $E$ if and only if $j = i + 1$ and $\text{projcom}(S_i, H_i) = \text{projcom}(S_{i+1}, H_i)$. Note that for every $H_i$ there is at least one vertex $(H_i, S_i)$, since for all integers $i$, $H_i \sqsubseteq_{RT} S_i$, for a history $S_i$.

We use König’s Path Lemma to show that the resulting graph $G$ contains an infinite path $(H_0, S_0), (H_1, S_1), \ldots$ and the limit of histories $S_i$ is a history $S$ such that $H \sqsubseteq_{RT} S$.

The graph $G$ has only one root $(H_0, S_0)$, where $H_0 = S_0 = \varepsilon$. Clearly, every vertex is reachable from the root.

There are only finitely many actions in $H_i$ and each $S_i$ is a permutation of these actions, so there can only be finitely many histories $S_i$ for each $H_i$. Thus, there are only finitely many vertices of the form $(H_i, S_i)$. Since all outgoing edges of a vertex $(H_i, S_i)$ are directed to vertices of the form $(H_{i+1}, S_{i+1})$, the outdegree of every such vertex is also finite.

By Lemma 7.2, since $H_i = (H_{i+1})_{|i}$, we get that $S_i$ is a subsequence of $S_{i+1}$, which along with the fact that $H_i \sqsubseteq_{RT} S_i$, gives $\text{projcom}(S_{i+1}, H_i) = \text{projcom}(S_i, H_i)$. Therefore, for every vertex $(H_{i+1}, S_{i+1})$, there is a vertex $(H_i, S_i)$ such that $\text{projcom}(S_{i+1}, H_i) = \text{projcom}(S_i, H_i)$. Thus, we can iteratively construct a path from $(H_0, S_0)$ to every vertex $(H_i, S_i)$ in $G$, implying that $G_H$ is connected.

By König’s Path Lemma, $G$ contains an infinite path starting from the root vertex $(H_0, S_0)$. Let $(H_0, S_0), (H_1, S'_1), \ldots, (H_i, S'_i), \ldots$ be the infinite path.

Finally, we construct an infinite history $S$ such that $H \sqsubseteq_{RT} S$. For every integer $k$, let $\psi^k_i$ be the $k$-th committed or aborted action in $S'_i$, and let $\text{idx}(k)$ be the minimum index such that for every $j \geq \text{idx}(k)$, $S'_j|_{\psi^k_i}$ is complete. Informally, $S'_{\text{idx}(k)}$ is the first $S'_i$ in the infinite path in which first $k$ transactions are complete. For every integer $k$, $\text{idx}(k)$ exists and is unique since we take the minimum of all such indexes. Let $T_k$ be the $k$-th transaction in $S'_{\text{idx}(k)}$, i.e., $T_k = \text{txof}(\psi^k_{\text{idx}(k)}, S'_{\text{idx}(k)})$. 

81
Let $S = T_1 T_2 T_3 \ldots$. We argue that $H \sqsubseteq_{RT} S$. Since all transactions in $H$ are complete, for every committed or aborted action $\psi$ in $H$, there is $H_i$ such that $\psi \in H_i$. Since $H_i \sqsubseteq_{RT} S'_i$, $\psi$ is in $S'_i$. Furthermore, since $S'_i$ is a subsequence of $S'_{i+1}$ (and any following $S'_j$), if $S'_i \downarrow \psi$ is complete then for every $j > i$, $S'_j \downarrow \psi$ is complete. Therefore, $S$ contains all the transactions in $H$.

Thus, $S$ contains all actions in $H$, and $S$ respects the real-time order of $H$; otherwise there would be a vertex $(H_i, S'_i)$ such that $S'_i$ is not equivalent to $H_i$ or violates the real-time order of $H_i$. Thus, $H \sqsubseteq_{RT} S$, which completes the proof.

From Lemma 6.2 and Theorem 7.4, we get the following result.

**Corollary 7.5** Under the restriction that in any infinite history $H$, every transaction $T \in \text{tx}(H)$ is complete, STMS is a limit-closed property.
Chapter 8

Related Work

8.1 Transactional Memory Basics

Herlihy and Moss [24] suggested to use transactional memory to implement lock free data structures. Shavit and Touitou [36] showed Transactional Memory can be implemented in software (STM). Herlihy et al. [23] proposed a scheme that provides a software API for programmers. There are many variants of it like obstruction-free STM (OFTM) [15], where each process that does not encounter contention from any other process is guaranteed progress, or like hybrid hardware-software TM [9] which uses both hardware and software mechanisms to improve performance.

Many STM implementations have been given so far. DSTM [23] is an object based STM that maintains two versions of each object, a current or working version and an old stable version. A transaction marks an object that it is writing. All read operations in a transaction are validated with every new read. LSA [34] is also an object based STM that has multiple versions for each object unlike DSTM that has only two. It is based on validity interval for snapshots. There is also no revalidation of previous reads. It uses a global counter for validation, which is incremented when an update transaction commits. TL2 [10] uses a global version clock. There is a lock for every memory location, augmented by version number. An update transaction acquire locks on location to be written, increment the global version clock and try to commit by validating read set. NOrec [8] has a global sequence clock for update transactions. Every transaction maintains a local write buffer for write operations and writes to the memory only after validating. It does not hold the lock while committing. Some STMs work better for write-dominated workloads, while others work better for read-dominated workload. ASTM [30] is an adaptive STM that switches between two such STMs based on the current workload.

Previous work on other consistency conditions for TM [4, 11, 14, 16, 26] was discussed in Chapter 6.
8.2 Database Consistency Conditions

The notion of transactions is well-known from the database area, with the associated notion of serializability [33]. Contrary to the popular belief that database consistency conditions only consider committed transactions, there is extensive literature that considers consistency of the aborted transactions as well, as long as they are running. Rigorousness [7] is the strongest database consistency condition. It restricts a transaction to access a data item only if all the conflicting transactions have completed. Strictness [5] relaxes this requirement by allowing a transaction to write a data item even if the conflicting transactions have not completed. Avoiding Cascading Aborts (ACA) [5] is a weaker condition than strictness, and restricts a transaction to read a data item written by completed transactions only. Recoverability [17] is the weakest database consistency condition, which restricts a transaction reading a data item to complete only if all the conflicting transactions have completed. Weikum and Vossen [38] proved the containment relationship between all these database conditions.

Opacity and VWC are strictly weaker than rigorousness, but incomparable with strictness and other database consistency conditions [3] (Figure 8.1). It was also shown [3] that all TM implementations that write to shared memory only after calling commit, ensure strictness but not all of them ensure rigorousness and opacity.

8.3 Programming Language Semantics for TM

Some prior work has investigated the semantics of different programming languages with atomic blocks and the feasibility of their efficient implementation. Abadi et al. [1] formalize transactional memory at the programming language level using the automatic mutual exclusion construct, similar to our atomic block. They have studied trade offs between semantic simplicity and feasibility of different implementation strategies for TM. Harris et al. [19] formalized the ideas of transactional memory in the setting of the functional language Haskell. Moore and Grossman [31] present semantics for a simple
λ-calculus with transactions, and study strong and weak atomicity as well as nesting of transactions.

The only prior work that relates the programming languages and TM implementation, in a formal manner, is by Harris et al. [20], which proved that a specific TM implementation, Bartok-STM, validates a particular semantics of atomic blocks in a programming language.

Our work is complementary to previous proofs that certain TM systems satisfy opacity [6], as it lifts such results to the language level. Our work is also more general than that of Harris et al. [20], since our results allow establishing the semantics for any TM implementation satisfying STMS. However, some of the above-mentioned papers, e.g., [1, 31], investigate advanced language interfaces that we do not consider, such as nested transactions and access to shared data both inside and outside transactions.

8.4 Observational Refinement

We employ a well-known technique from the theory of programming languages, called observational refinement [21, 22], to explore the most appropriate way to specify TM consistency.

Observational refinement has previously been used to characterize correctness criteria for libraries of concurrent data structures. Filipovic et al. [12] proved that sequential consistency [28] is necessary and sufficient for observational refinement, and so is linearizability [25] when client programs can interact via shared global variables. Gotsman and Yang [13] adjusted linearizability to account for infinite computations and showed its sufficiency for observational refinement in the case when the client can observe the validity of liveness properties.

Our work takes this approach from the simpler setting of concurrent libraries to the more elaborate setup of transactional memory. Since we allow the abstract transactional memory to have incomplete transactions, we hope that in the future we can generalize our consistency condition to specify liveness properties, along the lines of [13].
Chapter 9

Conclusions

This thesis establishes an approach for evaluating and comparing TM consistency conditions. We introduced STMS, a new consistency condition, and proved that STMS is necessary and sufficient for observational refinement for a programming language in which local variables modified by a transaction are not rolled back upon an abort. STMS is derived from TMS1 [11], which also relaxes the constraint of a single justifying abstract history. However, STMS and TMS1 differ in the way they treat aborted transactions. TMS1 does not allow including previous aborted transactions in the abstract history, while STMS requires the abstract history to include all the previous aborted transactions. We also proved that TMS1 is necessary and sufficient for observational refinement for a programming language in which local variables are rolled back upon an abort.

We proved that STMS is limit-closed, when every transaction completes in an infinite history. We also showed the relation between STMS and other consistency conditions, such as Opacity [16], DU-opacity [4], VWC [26] and TMS2 [11]. We proved that every complete history that is STMS, is also opaque, and a TM that is STMS is also opaque, if it contains a completion of every history in the TM.

We proved that VWC, a weaker consistency condition than STMS and TMS1, is sufficient for a weaker notion of observational refinement. It is likely that for other programming languages or notions of observations, other consistency criteria, such as DU-opacity and TMS2, will be necessary or sufficient for observational refinement, resulting in different trade-offs between the efficiency of TM implementations and the flexibility of their programming interfaces.

We hope that the link between TM consistency conditions and programming language abstractions we establish in this thesis will enable TM implementors and language designers to make informed decisions about such trade-offs. Our approach can also reduce the effort of proving that a TM implements its programming interface correctly, by only requiring its developer to show that it satisfies the corresponding consistency condition.
Bibliography


 vår främlinga utsikt

记忆力是多线程编写代码的保障。它可以简化平行编程，因为开发者可以指定某些代码块为原子块，并且可以确保它们在不被干扰的情况下执行。其中一个主要的挑战在于，设定一种内存的一致性条件，以便为存活的事务提供保证，即那些尚未完成（成功或失败）的事务。

尽管提出了几种内存一致性条件，但都没有成功地从形式上描述原子块接口的语义。内存一致性条件需要足够宽松，以便在实际实现中提供灵活性，但又足够强大，能够防止内存事务行为不当的不良行为，这可能会导致运行时错误。

这种最后一个特性可以被正式地通过观测精化（observational refinement）来描述。它定义了两个内存事务版本之间的关系，其中可以从具体版本推断出使用抽象版本的程序特性。因此，抽象版本可以用作具体版本的规范。

这个工作引入了一个新的内存一致性条件STMS（Strong Transactional Memory Specification），并且证明它是观测精化条件的必要和充分条件。

此结果表明，STMS是这些环境中内存一致性的最弱条件。

此外，此工作还证明，STMS是一个闭包性质，这意味着如果一个历史满足STMS条件，那么它的所有子历史也满足STMS条件。并且在所有最终完成的事务下，STMS是一个闭包性质，这意味着如果一个无限序列的子序列都满足STMS条件，那么整个序列也满足STMS条件。这表明STMS是一个安全性质。

最后，STMS是一个比透明度（opacity）弱的条件，透明度是内存一致性的一个已知和公认的条件。对于最后完成的事务，STMS和透明度是等价的。

以上结果提供了一种新的方法来评估和比较内存一致性的条件。

几项新发现的内存一致性条件有助于证明，它们可以通过证明STMS的一致性条件来实现，因为它们只需要证明内存一致性条件。

המחקר נעשה בהנחתה של פרופסור חגי עטייה, בפקולטה להנדסה המחשב, ודר”ר ענש רייצקי, באוניברסיטת תל-אביב.

אני מצהרו על חומת הכתיפה הגדולה בחשדותם.
A Methods for Verification of Program Codes — Theory, Applications and Implementation

Haim Gal

Chairman

Professor Yehuda Barash

Department of Computer Science

Sandeep Anees

Technion — Israel Institute of Technology

September 2015
גישה מבוססת-שפות חכונות לאלגוריתמים

סכיאות

סאנדימ האנס