TECHNIQUES FOR VERIFYING TRANSACTIONAL PROGRAMS AND LINEARIZABILITY

by

ÖMER SUBAŞİ

A Thesis Submitted to the
Graduate School of Engineering
in Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in

Computer Science and Engineering

Koç University

Mayıs, 2012

Koç University

Graduate School of Sciences and Engineering

This is to certify that I have examined this copy of a master's thesis by

ÖMER SUBAŞİ

and have found that it is complete and satisfactory in all respects,

and that any and all revisions required by the final

examining committee have been made.

Committee Members:

Assistant Professor Serdar Taşıran

Associate Professor Öznur Özkasap

Associate Professor Alper Demir

Date:

To my family

ABSTRACT

In this thesis, we consider two significant software verification problems regarding concurrent, multi-threaded programs. First, we consider the problem of proving the linearizability of a concurrent implementation. We suggest a sound method for verifying a concurrent data structure implementation is linearizable to its sequential specification based on the QED proof system. Our method is based on transforming the concurrent implementation into the specification aimed for the implementation. The transformation is governed by proof rules of the proof system. Each transformation step preserves certain behaviors of the program that are relevant to the specification of the program. At the limit, we obtain a program that is being the sequential specification considered for the correctness of the original concurrent program. In our approach, we provide the formalization of the linearizability notion, concurrent programs as well as the proof system and its rules. We then state our theoretical findings.

Second, we study the verification of transactional programs with programmer-defined conflict detection. While programmer-defined conflict detection is desirable in terms of performance issues of the transactional memory systems, such relaxed conflict detection complicates the verification of the programs that use these transactional systems. In particular, the ability to use sequential reasoning provided by conventional transactional memories is lost when the relaxed conflict detection is introduced. In our approach, we first model and formalize such transactional programs. Then, we provide a recipe for the verification process in which we regain the ability to use sequential reasoning. This recipe includes abstractions provided by the programmer on the original program. After the abstractions are introduced, the verification problem becomes the sequential verification problem automated by sequential verification tools such as VCC and HAVOC. Our soundness theorem guarantees that once the abstracted program is verified, so is the original transactional program.

ÖZETÇE

Bu tezde, çoklu-iş parçacıklı ve koşut-zamanlı programların doğrulanması ile ilgili iki önemli probleme değinmekteyiz. Önce koşut-zamanlı bir uygulamanın doğrusallaştırılabilirliğini ispatlamak problemini ele alıyoruz. Geçerli bir ispat sistemine dayanarak doğrusallaştırılabilirlik ispatlarının mümkün kılındığı bir metod sunuyoruz. Metodumuz koşut-zamanlı uygulamanın amaçlanan tanımlamalarına dönüştürülmesine dayanmaktadır. Bu dönüştürülmeler ispat sistemin kuralları tarafından yönetilmektedir. Her dönüşüm adımı programın doğruluk ile ilgili davranışlarını korumaktadır. Limitte programın doğruluğunu gösteren tanımına ulaşılır. Yaklaşımımızda, doğrusallaştırılabilirlik kavramını , koşut-zamanlı programları ve ispat sisteminin kurallarını tanımlamaktayız. Ardından teorik bulgularımıza yer vermekteyiz.

İkinci olarak, programcı tarafından tanımlanan çakışmaları bulan işlemsel programların doğrulanması problemini ele alıyoruz. Performans açısından bu tür sistemler istenilse de bu sistemler kendilerini kullanan işlemsel programların doğrulanmasını zorlaştırmaktadır. Özellikle de bu sistemler dizisel kanıtların yapılmasını engellemektedir. Yaklaşımımızda, önce bu tip programları modelliyoruz. Sonra, dizisel kanıtların yapılmasını sağlayan bir reçete sunuyoruz. Bu reçetede program üzerinde soyutlamalar yapılmaktadır. Soyutlamalar yapıldıktan sonra, doğrulama işi otomatik dizisel doğrulamaya dönüşmektedir. Bu da VCC ve HAVOC gibi dizisel doğrulama araçlarıyla yapılabilir. Ana teoremimiz ise, soyutlanmış program doğrulamasının ilk orjinal işlemsel programın doğrulaması anlamına geldiğini ispatlamaktadır.

ACKNOWLEDGMENTS

I would like to express my immense gratitude to my advisor Assist. Prof. Serdar Taşıran. His strong guidance, experience, understanding, patience and invaluable advices helped me to conduct my graduate study and do academic research. I would like to thank the other members of my thesis committee, Öznur Özkasap and Alper Demir for critical reading of this thesis and and for their valuable comments.

I am grateful to The Scientific and Technical Research Council of Turkey (TÜBİTAK) for providing financial support under scholarship programme for students studying for an MSc with a thesis requirement (2210). My research presented in this thesis was also supported by TÜBİTAK under project grant 111E135.

I am grateful to Tayfun Elmas and Ali Sezgin for their continuous support, strong contribution and guidance during my graduate study and research.

I am grateful to Serpil Ekşioğlu for her endless support during my eight years in Koç University.

I would like to thank all my friends from Koç University, especially the members of ENG-230, İsmail, Erdal, Hassan, Ayşe Nur, Özge, Can, and İdil.

I must give immense gratitude to my family.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

INTRODUCTION

There are new emerging multiprocessor technologies in computing. Due to these advances and performance requirements, concurrent software becomes mainstream in many computing systems such as databases, operating systems and web servers. To make use of these multiprocessor systems as well as match some performance criteria, concurrent software makes use of intricate synchronization techniques, fine-grain locking and complicated non-blocking operations. Thus, because of the complexity of the concurrent software, the verification of such software becomes extremely difficult to achieve.

This thesis is concerned with two different aspects of verification of concurrent software. First, we study the correctness of concurrent data structure implementations. In this first aspect, we study how to prove the linearizability of the concurrent implementations. This study is based on the former work of Elmas et. al [9]. The correctness condition linearizability [14] is prevalently used for concurrent data structure implementations. This condition constructs a relation between the implementation and the specification of the data structure. To prove the linearizability of a concurrent implementation, we use QED proof system [9] in which we make use of atomicity of the actions as a proof tool. Our focus in this thesis will be mostly on the theoretical part of our joint work [8]. We will study this aspect of verification of concurrent software and our theoretical findings in Chapter 2.

The second part of this thesis is concerned with the verification of transactional programs. These programs make use some transactional memory implementation in certain parts of them where they need the atomicity guarantee. In this thesis, we will study types of programs using transactional memory that has relaxed conlict detection mechanism as opposed to conventional transactional memory systems. Transactional memory [12] provides atomic code blocks to the programmmer where they need atomicity or mutual exclusion. These blocks are often composable and easy to reason about. Thus, the verification of such concurrent transactional programs becomes easy. However, standard transactional memory systems with such conventional atomic blocks suffer from performance. Due to this performance issues, some transactional memory systems offer different mechanisms that relaxes on the conflict detection in order to gain in terms of performance. Subsequently, such relaxed transactional programs become hard to formalize and verify. Our study in this thesis achieves to model, formalize and verify transactional programs that use relaxed conflict detection. We present our work in Chapter 3 in detail. This work is published in WoDET 2012 workshop [26].

Our general contributions in this thesis are:

- Theory of verifying linearizability using QED proof system
- Method for model, formalize and verify transactional programs with programmer-defined conflict detection
- Verified large benchmarks from STAMP suite [5]

We will state our specific contributions in Chapter 2 and Chapter 3.

The outline of the thesis is as follows: In the next section, we will provide related work on proving linearizability and transactional program verification. Then, Chapter 2 discusses our first main contribution regarding proving linearizability using the QED proof system with the focus being the theoretical part of our joint work [8]. Chapter 3 discusses our work on the verification of relaxed transaction programs which is second main part of our work. Chapter 4 concludes the thesis.

1.1 State of Art

1.1.1 Linearizability

Refinement between a concurrent program and its sequential specification is well-studied [1, 15, 17, 18]. Previous work showed that, under certain conditions, auxiliary variables enable construction of an abstraction map to prove refinement [1, 15]. However, in practice writing an abstraction map for programs with fine-grained concurrency remains a challenge since there are a large number of cases to consider. [23] used a complex abstraction map, called *aggregation function*, that completes the atomic transactions that are committed but not yet finished. The refinement proofs in [14, 13, 10, 6], despite being supported by automated proof checkers, all require manual guidance for the derivation of the proof, requiring the user to manage low-level logical reasoning. On the other hand, in our method the user guides the proof via code transformations at the programming language level. Recently, [28] provided a tool that automates the derivation of the proof using shape abstraction. To our knowledge, its automating ability is limited to linked-list based data structures and it still requires identification of the possible commit points.

Owicki-Gries [22, 14] and rely-guarantee [30] methods have been used in refinement proofs. However, in the case of fine-grained concurrency, deriving the proof obligations in both approaches requires expertise. The idea of local reasoning is exploited by separation logic [2] which is not particularly useful for shared objects with high level of interference. In these cases, we show that abstraction is an important tool to reduce the effects of interference.

Wang and Stoller [31] statically prove linearizability of the program using its sequentially executed version as the specification. Their notion of atomicity is defined over a fixed set of primitives, which is limited in the case of superficial conflicts. On the other hand, our notion of atomicity is more general and supported by abstraction to prove atomicity even under high level of interference. They provided hand-crafted proofs for several non-blocking algorithms, and our proofs are mechanically checked. In [11], Groves gives a hand-proof of the linearizability of the nonblocking queue, by reducing executions the fine-grained program to its sequential version. His use of reduction is non-incremental, and must consider the commutativity of each action by doing a global reasoning, while our reasoning is local.

1.1.2 Transactional Program Verification

Attiya et al. [3] provide a set of techniques for reducing concurrent programs to sequential programs for simplifying verification. They show that by only doing sequential reasoning that a program correctly implements two-phase, tree, and hand-over-hand locking protocols in the concurrent context. Relying on this result, one can soundly verify any safety property by only considering serial executions. This work considers concurrency control mechanisms that guarantee conflict-serializable executions, i.e, ones in which no concurrent conflicting accesses are allowed. A key distinguishing feature of our work is that we handle transaction executions that are not serializable. We explicitly consider and model conflicts and verify correctness in their presence. To do this, we reduce the original, unserializable program to a provably-serializable abstract program.

Our soundness theorem builds upon our previous work [9] on verifying concurrent programs by combining abstraction and reduction. One could directly use the proof system in [9] to verify our benchmarks, although this would require a low-level proof script for each program. In this work, we model programs using programmer-defined conflicts such that the proof can be performed in a streamlined manner, without devising a proof script or applying the low-level proof steps in [9]. The user only has to provide an abstraction invariant and sequentially verify some assertions and the intended properties of the program. The verification of assertions in our approach implies that there exists a proof of the same property in the style of [9], while our approach avoids pairwise commutativity checks between actions.

Our modeling of transactions from the programmer's point of view is based on Michael L. Scott's work on defining a sequential specification for TMs [24]. This view of TMs allowed us to reason about transactions at the source-code level and without having to consider the low-level (possibly out of order) execution of transactional accesses by the TM. Using his formalization, we are able to use the requirements satisfied by correct TM implementations implicitly in our modeling and verification.

Finally, correctness of transactions with relaxed conflict detection and snapshot consistency are related. One use of relaxed conflicts is to allow transactions to work locally on a (possibly stale) snapshot, as in the Labyrinth example. In general, however, our method verifies the correctness of transactions that succeed despite stale reads, and these stale reads do not have to constitute a snapshot.

Chapter 2

PROVING LINEARIZABILITY USING QED PROOF SYSTEM

2.1 Introduction

Linearizability is a well-known correctness criterion for concurrent data structure implementations [14]. Linearizability establishes a connection between the implementation of the data structure and the sequential specification of the data structure. It states that every concurrent operation of the implementation takes effect atomically between its invocation, or call, and return points. The correct effect of the operation is determined by the sequential specification of the data structure.

The typical way to prove that a data structure is linearizable is to construct an abstraction map from concurrent implementation to the sequential specification [1]. When constructing such an abstraction map, one needs to identify the commit points or actions of the implementation. This is because commit points are the points where the effect of the implementation becomes visible to other threads. In the abstraction map, commit point is mapped to a point in the sequential specification. The other operations in the implementation are mapped to identity transitions in the sequential specification. Identifying commit points becomes difficult when the concurrent implementation use fine-grained concurrency and intricate synchronization. This is because the commit action consists of smaller actions that make visible changes to other threads. The abstraction map needs to consider these changes while mapping a concurrent operation to a sequential specification operation. This causes the abstraction map construction to be difficult under fine-grained concurrency and intricate synchronization.

In this section, we propose a sound proof system which simplifies the linearizability proofs. The key idea is to rewrite the concurrent program with larger atomic blocks. Using atomicity, we provide an effective way to prove linearizability as follows: When proving a concurrent implementation is linearizable with respect to a sequential specification, we transform the implementation into the specification in a sequence of phase. In the first phase, we use abstraction and reduction to collect atomic actions into larger atomic actions. By doing this, we eliminate the effects of the thread interleavings. In the second phase, variable addition and hiding to make the implementation closer to the specification. In the end, we obtain a code that represents the specification. Our soundness theorem imply the linearizability of the implementation.

Our method has important advantages. The first one is that the abstraction map is constructed incrementally during the program transformation. That is, we avoid complex construction of the map all at once. Another advantage of our method is that we do not require the identification of commit points. This advantage is even more important when the commit point can only be determined at runtime depending of the thread schedule. Last, we prove the soundness of our method in which during program transfomation we preserve original program behaviors which are related to the linearizability of the program.

Our contributions are:

- Defining linearizability in QED proof system
- Proving that complementing other methods with our transformations is sound
- Proving that transforming implementation to specification guarantees linearizability of implementation

In section 2.2, we give formal syntax and semantics of the concurrent programs. Next, in section 2.3, we define our sound proof system with formal proof rules. In section 2.4 we formally define linearizability and related notions. In that section, we give full proofs about the proof system, in particular the proof of the main soundness theorem which is our core contribution.

2.2 Concurrent programs: syntax and semantics

In this section we formalize the syntax and semantics of our programs.

Program. A program *P* is a tuple $P = \langle Global_P, Proc_P \rangle$. $Global_P$ is the set of global variables which are uniquely named. $Proc_P$ is a set of procedures. A procedure is a tuple $\langle \rho, \mathit{local}_\rho, \mathit{body}_\rho \rangle$, where ρ is the name of the procedure, local_ρ is the set of *local variables*, and body_ρ is the *body* of the procedure.

We distinguish the input variables $\overrightarrow{in}_\rho \subseteq \mathit{local}_\rho$ and the output variables $\overrightarrow{out}_\rho \subseteq \mathit{local}_\rho$. The tuple $\langle \rho, \overrightarrow{in}_\rho, \overrightarrow{out}_\rho \rangle$ is called the *signature* of the procedure. The signatures of the procedures in $Proc$ form the signature of the program, denoted $\mathsf{Sig}(P)$. We assume the convention that the variables in $\overrightarrow{in}_\rho$

are read-only and $\overrightarrow{out}_\rho$ are write-only, and the rest of the variables in $local_\rho$ can be both read and updated.

We use Var_P to denote $Global_P \cup \bigcup_{\rho \in Proc} local_\rho$. We assume that each local variable is used in a unique procedure. Var'_P consisting of the primed version of each variable in Var_P. We omit the subscripts when the program and the procedure are clear from the context.

Execution model. Let Tid be the set of all thread identifiers. Without loss of generality, we assume that each thread calls one procedure ρ from $Proc$, and terminates when ρ returns. we refer to the current thread id through the special variable $tid \in Global$, whose domain is Tid.

Syntax. We assume that each atomic statement α, which we call an *atomic action*, is in the form: `assert` $a; p$. Let ρ be the procedure whose body contains α, and $V = Global \cup local_\rho$. The *assert predicate* a be over only unprimed variables from V. The *transition predicate* p is over both primed and unprimed variables in $V \cup V'$. For any action α, let ϕ_α and τ_α denote its assert and transition predicates. For instance, $\phi_\alpha = a$ and $\tau_\alpha = p$, for α given above.

We use sequential composition (;), choice ($\Box$) and loop ($^{\circlearrowleft}$) operators to form *compound statements*. We also define the *nullary action* stop, which appears only at runtime and intuitively marks the end of fully executing a statement. A *full action* is either the *nullary action* stop which intuitively marks the end of the code, or a compound action c sequentially composed with the nullary action, c; stop. Let $Atom$ and $Full$ denote the set of all atomic and full actions, respectively.

Program states. A program state s is a pair consisting of

- a *variable valuation* σ_s that maps a thread id and a variable to a value, such that $\sigma_s(t, g) = \sigma_s(u, g)$ for all states s and thread id's t, u, whenever g is a global variable.

- a *code map* ϵ_s that keeps track of a (compound) statement for each thread, such that $\epsilon_s(t) = c$ means that at program state s, the remaining part of the program to be executed by thread t is given by c.

A program state s is called *initial* if $\forall t \in Tid.\ \exists \rho \in Proc.\ \epsilon_s(t) = body_\rho$, i.e. every thread is about to call a procedure. State s is called *final* if $\epsilon_s(t) =$ stop, for all $t \in Tid$. We write $Initial(s)$ to denote that s is an initial state and write $Final(s)$ to denote that s is a final state.

Let $\sigma_s|_V$ denote the projection of valuation σ_s on $V \subseteq Var$. Define $s|_V$ to be the program state $(\sigma_s|_V, \epsilon_s)$.

$$\boxed{\hookrightarrow \subseteq \mathit{Full} \times \ (\mathit{Atom} \cup \{\lambda\}) \times \mathit{Full}}$$

$$\text{A-EVAL}\ \frac{\gamma \in \mathit{Atom}}{\gamma; c_1 \overset{\gamma}{\hookrightarrow} c_1} \qquad \text{C-LEFT}\ \frac{\gamma = \lambda}{c_1 \Box c_2 \overset{\gamma}{\hookrightarrow} c_1} \qquad \text{C-RIGHT}\ \frac{\gamma = \lambda}{c_1 \Box c_2 \overset{\gamma}{\hookrightarrow} c_2} \qquad \text{L-ITER}\ \frac{\gamma = \lambda}{c_1^{\circlearrowleft} \overset{\gamma}{\hookrightarrow} c_1; c_1^{\circlearrowleft}} \qquad \text{L-SKIP}\ \frac{\gamma = \lambda}{c_1^{\circlearrowleft}; c_2 \overset{\gamma}{\hookrightarrow} c_2}$$

$$\text{S-EVAL}\ \frac{c_1 \overset{\gamma}{\hookrightarrow} c_2}{c_1; c_3 \overset{\gamma}{\hookrightarrow} c_2; c_3}$$

Figure 2.1: Obtaining all possible subactions of a given full action via the silent transformation relation, $\hookrightarrow$.

Predicates over program variables. For an assert predicate x, let $x[t]$ denote the predicate in which all free occurrences of tid is replaced with t. We say that a program state s satisfies $x[t]$, denoted as $s \vDash x[t]$ or as $x[t](s)$, if $x[t]$ evaluates to true when all free occurrences of each unprimed variable v is replaced with $\sigma_s(t, v)$. An assert predicate is called a *state predicate* if it does not contain any free occurrence of tid.

Similarly, the pair of program states (s_1, s_2) satisfies a transition predicate $p[t]$, denoted as $(s_1, s_2) \vDash p[t]$ or as $p[t](s_1, s_2)$, if $p[t]$ evaluates to true when each unprimed variable v is replaced with $\sigma_{s_1}(t, v)$ and each unprimed variable v' is replaced with $\sigma_{s_2}(t, v)$.

Let $\mathsf{fv}(p)$ be the set of free variables in the (state or transition) predicate p.

2.2.1 Operational semantics

Configurations. The evaluation of a full statement is given in terms of the *silent transformation relation*, $\hookrightarrow$, whose definition is given in Fig. 2.1. Intuitively, if we imagine the execution of a full statement represented as a flowchart with an explicit control pointer denoting what to execute next, the silent transformation relation corresponds to advancing the control pointer over the flowchart not modifying any program variable's value. When this imaginary control pointer selects a branch, it is represented by the label λ which is called the *invisible transition*. Otherwise, the label is the content of the box over which the control pointer passes.

For full actions c and d, and a string $\overline{\gamma} = \gamma_1 \ldots \gamma_n$ over $\mathit{Atom} \cup \{\lambda\}$, we let $c \overset{\overline{\gamma}}{\hookrightarrow} d$ denote a sequence of silent transformations

$$c = c_0 \xhookrightarrow{\gamma_1} c_1 \ldots \xhookrightarrow{\gamma_n} c_n = d$$

A program state s' is in $\mathsf{conf}(s)$, the *configurations* reachable from program state s, if, for all t, there exists some string $\overline{\gamma}_t$ such that $\epsilon_s(t) \xhookrightarrow{\overline{\gamma}_t} \epsilon_{s'}(t)$. Intuitively, s' is a configuration of s if s' can be obtained by moving forward the control pointer of each thread's program an arbitrary number of, possibly 0, steps.

Let s and s' be program states, t be a thread id. Then, s' is called a (t, α)-*successor* of s, if the following conditions hold:

- $\epsilon_s(t) \xhookrightarrow{\lambda^k \alpha} \epsilon_{s'}(t)$, for some $k \geqslant 0$.
- for all $u \neq t$, $\epsilon_{s'}(u) = \epsilon_s(u)$,

Intuitively, s' is a (t, α)-successor of s if at s thread t has α as a possible next action and s' is the same as s except the control flow at t skips over α.

Execution semantics. We assume a sequentially-consistent memory model. For thread t and $\gamma \in Atom$, (t, γ) is called a *transition label*. Intuitively, $s \xrightarrow{(t,\alpha)} s'$ holds when t can execute α next (in which case s' is a (t, α) successor of s), all other threads do not update their control flow, all local variables of other threads remain the same, the global variables and local variables of t are updated so that the transition predicate of α is satisfied. Formally, $s \xrightarrow{(t,\alpha)} s'$ if $(s, s') \models \tau_\alpha[t]$ and for all $u \neq t$ and for any local variable x, $\sigma_s(u, x) = \sigma_{s'}(u, x)$.

Run. A run r of the program is a sequence of state transitions:

$$r = r_1 \xrightarrow{(t_1,\alpha_1)} r_2 \xrightarrow{(t_2,\alpha_2)} \cdots \xrightarrow{(t_{n-1},\alpha_{n-1})} r_n$$

Let $Tid(r)$ denote the set of threads occurring in r. Let r_i denote the i^{th} program state, and $r(i)$, the i^{th} transition label (t_i, α_i) in r. For a state predicate ϕ, we say that r is a run of *P* from ϕ if $Initial(r_1)$ and $r_1 \models \phi$.

The run is *maximal* if r_n cannot make any transition. We will only consider maximal runs of the programs.

Trace. A *trace* is a sequence of transition labels, $\mathbf{l} = l_1 \ldots l_k$. The trace moves a state s_1 to s_{k+1}, written $s_1 \xrightarrow{\mathbf{l}} s_{k+1}$, if there is a run r of P over $\mathbf{l}$, such that $r_j = s_j$, for all $1 \leqslant j \leqslant k + 1$ and $r_i \xrightarrow{l_i} r_{i+1}$.

Violation-freedom. A run r of P from ϕ is called a *violation* if $\neg\phi_\alpha[t](r_k)$ evaluates to true for some (t, α). Intuitively, a violation is a run of P that starts from an initial program state s_1 and reaches a program state s_k which violates the assert predicate, ϕ_α, of an action α which thread t can execute at state s_k. A run is said to be *successful* if it is not a violation. We indicate a successful run as $s_1 \xrightarrow{1} s_2$ and a violation as $s_1 \xrightarrow{1} \mathsf{error}$.

2.3 Program transformations

In this section, we formalize our notion of proof and introduce the rules for the proof calculus. A *proof state* is the pair $(P, \mathcal{I})$, where P is a program, and $\mathcal{I}$ is a state predicate, called the *inductive invariant* of the program. We require that for every proof state $(P, \mathcal{I})$, all the atomic actions of *P* preserve $\mathcal{I}$. An atomic action α preserves $\mathcal{I}$, written $\alpha \leftrightarrows \mathcal{I}$, if $s_1 \xrightarrow{(t,\alpha)} s_2$ and $s_1 \models \mathcal{I}$ imply $s_2 \models \mathcal{I}$.

A proof consists of rewriting the input program, denoted P_1, iteratively so that, in the limit, one arrives at a program, denoted P_n, that can be verified by sequential reasoning methods. Formally, the proof is expressed as $(P_1, \mathsf{true}) \dashrightarrow (P_2, \mathcal{I}_2) \dashrightarrow \cdots \dashrightarrow (P_n, \mathcal{I}_n)$ Each proof step is governed by a proof rule, which we present below.

The following proof rule states the general form of updating $\mathcal{I}$, replacing it with a stronger invariant.

Rule 1 (Invariant) *Replace invariant $\mathcal{I}_1$ with $\mathcal{I}_2$ if $\alpha \leftrightarrows \mathcal{I}_2$ for all the actions α in P, and $\mathcal{I}_2 \Rightarrow \mathcal{I}_1$.*

The basic idea in reduction and abstraction is to replace an action with another action that simulates the former.

Definition 1 (Simulation) *Let α, β be actions, t be an arbitrary thread id. We say β simulates α at proof state $(P, \mathcal{I})$, written $(P, \mathcal{I}) \vdash \alpha \preceq \beta$, if both of the following hold:*

$$\mathbf{S1}.\quad (\mathcal{I} \wedge \neg\phi_\alpha) \quad\Rightarrow\quad \neg\phi_\beta \qquad\qquad \mathbf{S2}.\quad (\mathcal{I} \wedge \tau_\alpha) \quad\Rightarrow\quad (\neg\phi_\beta \vee \tau_\beta)$$

Intuitively, **S1** states that if there is a violation with α, there has to be a violation with β substituted in place of α. **S2** states that for each violation-free run, replacing α with β results in either a violation, or a violation-free run with the same end state.

2.3.1 Reduction

Reduction, due to Lipton [20], creates coarse-grained atomic statements by combining fine-grained actions. An action α can be combined with another action if α is a certain type of mover. A mover is an action that can commute over actions of other threads in any run. We write $(P, \mathcal{I}) \vdash \alpha : m$ to indicate that α is $m-$mover in the proof state $(P, \mathcal{I})$, where $m \in \{\mathbb{L}, \mathbb{R}\}$.

We decide that an action α is a mover by statically checking a simulation relation, that states that commuting α with every β can lead to the same state or goes wrong. An assert predicate x is *p-stable*, if $\forall s, s'.x(s) \wedge p(s, s') \Rightarrow x(s')$.

Let $\mathsf{wp}(p, x)$, the *weakest (liberal) pre-condition* of predicate x for transition predicate p, stand for all states which cannot reach a state where x evaluates to false after executing p. Formally, $\mathsf{wp}(p, x) = \{s \mid \forall s'. p(s, s') \Rightarrow x(s')\}$. For two transition predicates p and q, define their composition $p \cdot q$, as the transition predicate $p \cdot q = \{(s_1, s_2) \mid \exists s_3. p(s_1, s_3) \wedge q(s_3, s_2)\}$. The operator $[\![\ ]\!]$ expresses the result of combining two actions to one atomic action.

$$[\![\alpha; \beta]\!] \quad = \quad \texttt{assert}\,(\phi_\alpha \wedge \mathsf{wp}(\tau_\alpha, \phi_\beta)); (\tau_\alpha \cdot \tau_\beta) \qquad [\![\alpha \Box \beta]\!] \quad = \quad \texttt{assert}\,(\phi_\alpha \wedge \phi_\beta); (\tau_\alpha \vee \tau_\beta)$$

Definition 2 (Left-mover) *Action α is a left-mover in proof state $(P, \mathcal{I})$, denoted $(P, \mathcal{I}) \vdash \alpha : \mathbb{L}$, if the following holds for every action β in P and every pair of distinct thread ids t and u: $(P, \mathcal{I}) \vdash [\![\beta[u]\ ;\ \alpha[t]]\!] \preceq [\![\alpha[t]\ ;\ \beta[u]]\!]$.*

Definition 3 (Right-mover) *Action α is a right-mover in proof state $(P, \mathcal{I})$, denoted $(P, \mathcal{I}) \vdash \alpha : \mathbb{R}$, if, for every action β in P, and every pair of distinct thread ids t and u: $(P, \mathcal{I}) \vdash [\![\alpha[t]\ ;\ \beta[u]]\!] \preceq [\![\beta[u]\ ;\ \alpha[t]]\!]$ and $\phi_\beta[u]$ is $\tau_\alpha[t]$-stable.*

The reduction rules below define the conditions under which non-atomic statements are transformed to atomic actions. We omit the rules about procedure calls and parallel composition which are similar to those of [9].

Rule 2 (Reduce-Sequential) *Replace occurrences of $\alpha\ ;\ \gamma$ with $[\![\alpha\ ;\ \gamma]\!]$ if either $(P, \mathcal{I}) \vdash \alpha : \mathbb{R}$ or $(P, \mathcal{I}) \vdash \gamma : \mathbb{L}$.*

Rule 3 (Reduce-Choice) *Replace occurrences of $\alpha \Box \gamma$ with $[\![\alpha \Box \gamma]\!]$.*

Rule 4 (Reduce-Loop) *Replace occurrences of $\alpha^{\circlearrowleft}$ with β if the following hold:*

L1. $(P, \mathcal{I}) \vdash \alpha : m$ *s.t.* $m \in \{\mathbb{R}, \mathbb{L}\}$ **L2**. $\beta \leftrightarrows \mathcal{I}$

L3. $\phi_\beta \Rightarrow \tau_\beta[Var/Var']$ **L4**. $(P, \mathcal{I}) \vdash [\![\beta\ ;\ \alpha]\!] \preceq \alpha$

2.3.2 Abstraction

An abstraction step consists of replacing an action α with another action β, which in principle leads to less interference with other actions.

Rule 5 (Abstraction) *Replace the action α with action β if $\beta \leftrightarrows \mathcal{I}$ and $(P, \mathcal{I}) \vdash \alpha \preceq \beta$.*

This rule is usually applied for an action `assert` $a; p$ by replacing it with 1) `assert` $b; p$ such that $b \Rightarrow a$ or 2) with `assert` $a; q$ such that $p \Rightarrow q$. While the former corresponds to adding extra assertions to the action, the latter adds more transitions.

2.3.3 Variable introduction and hiding

Intuitively, variable introduction rewrites some actions in the program so that these can refer to a new (*history*) variable. Variable hiding is the dual of variable introduction; each action is rewritten so that it does no longer refer to the hidden variable. Hiding a variable also requires quantifying out the variable in the invariant.

In order to ensure soundness, in both cases, we need a relation between actions over different sets of variables. For this, we extend our simulation relation ($\preceq$) for each rule. In addition, we require that the input and output variables of the procedures ($\overrightarrow{in}_\rho, \overrightarrow{out}_\rho$) are fixed during the proof; the rules below are not applicable to these variables.

Rule 6 (Add-Variable) *Add the new variable v to Var_P, and replace every action α with β whenever $(P, \mathcal{I}) \vdash \alpha \preceq_{+v} \beta$, which holds if the following are both valid:*

$$\mathbf{A1}.\quad (\mathcal{I} \wedge \neg\phi_\alpha) \;\Rightarrow\; (\forall v.\, \neg\phi_\beta) \qquad\qquad \mathbf{A2}.\quad (\mathcal{I} \wedge \tau_\alpha) \;\Rightarrow\; (\forall v.\, \neg\phi_\beta \vee (\exists v'.\, \tau_\beta))$$

Rule 7 (Hide-Variable) *Remove the existing variable v from the program, and replace the invariant $\mathcal{I}$ with $\exists v.\, \mathcal{I}$. Replace every action α with β whenever $(P, \mathcal{I}) \vdash \alpha \preceq_{-v} \beta$, which holds if the following are both valid:*

$$\mathbf{H1}.\quad (\exists v.\, \mathcal{I} \wedge \neg\phi_\alpha) \;\Rightarrow\; \neg\phi_\beta \qquad\qquad \mathbf{H2}.\quad (\exists v, v'.\, \mathcal{I} \wedge \tau_\alpha) \;\Rightarrow\; (\neg\phi_\beta \vee \tau_\beta)$$

In both of the rules, the first condition (**A1**, **H1**) states that violations are preserved. The second condition (**A2**, **H2**) states that transitions (over the common variables of α and β) are either preserved or additional violations are introduced.

2.4 Linearizability

In this section, we give the formal definitions and proofs for the linearizability.

Good and Bad. Below, we define $Good(P, \mathcal{I})$ as the set of pre- and post-state pairs associated with succeeding (maximal) runs of program *P* from states satisfying $\mathcal{I}$. $Bad(P, \mathcal{I})$ is the set of pre-states associated with violations. Formally,

$$\begin{aligned} Good(P, \mathcal{I}) &= \{(s_1, s_2) \mid Initial(s_1),\ s_1 \models \mathcal{I},\ \exists \mathbf{l}.\ s_1 \xrightarrow{\mathbf{l}} s_2,\ Final(s_2)\} \\ Bad(P, \mathcal{I}) &= \{s_1 \mid Initial(s_1),\ s_1 \models \mathcal{I},\ \exists \mathbf{l}.\ s_1 \xrightarrow{\mathbf{l}} \mathsf{error}\} \end{aligned}$$

P is said to be *good* from $\mathcal{I}$ if $Bad(P, \mathcal{I}) = \emptyset$; it is called *bad* from $\mathcal{I}$, otherwise.

Theorem 1 *Let* $(P_1, \mathcal{I}_1) \dashrightarrow \cdots \dashrightarrow (P_n, \mathcal{I}_n)$ *be a sequence of proof steps. Let* $V = Var_{P_1} \cap Var_{P_n}$ *and* $X = (Var_{P_1} \cup Var_{P_n}) \backslash \mathrm{V}$. *The following hold:*

$$\begin{aligned} &\mathbf{C1}.\quad Bad|_V(P_1, \exists X.\, \mathcal{I}_n) \subseteq Bad|_V(P_n, \exists X.\, \mathcal{I}_n) \\ &\mathbf{C2}.\quad \forall (s_1, s_n) \in Good|_V(P_1, \exists X.\, \mathcal{I}_n) : \\ &\qquad \mathbf{a}.\ s_1 \in Bad|_V(P_n, \exists X.\, \mathcal{I}_n) \quad or \quad \mathbf{b}.\ (s_1, s_n) \in Good|_V(P_n, \exists X.\, \mathcal{I}_n) \end{aligned}$$

Note that, since the input and output variables of procedures are fixed during the proof, so the set V above will always be nonempty. A corollary of the above theorem is that, if P_n is good from $\mathcal{I}_n$, then P_1 is good from $\mathcal{I}_n$. This means that, one can prove the assertions in P_1 by gradually obtaining programs with coarser-grained concurrency using our proof rules.

We prove below the relation between P_1 and P_n in order to prove linearizability. We first define *behavioral simulation*, a special type of simulation that relates two programs through their observable behaviors over procedure input and output values.

Behavioral simulation. Let $r = s_1 \xrightarrow{\mathbf{l}} s_n$ be a (maximal) run of the program. Let ρ be the procedure executed by t. We call the tuple $(t, \rho, \sigma_{s_1}(t, \overrightarrow{in}_\rho), \sigma_{s_n}(t, \overrightarrow{out}_\rho))$ the behavior of t in r and denote it by $\mathsf{beh}(r, t)$. The behavior includes the name of the procedure called by t, along with the values of the input and the output variables of the procedure. Notice that the first and the last states of the run provide us the values of $\overrightarrow{in}_\rho$ and $\overrightarrow{out}_\rho$, respectively. We write $\mathsf{Beh}(r)$ to denote $\{\mathsf{beh}(r, t) \mid t \in Tid(r)\}$.

We define $\mathsf{fst}(r, t)$ and $\mathsf{lst}(r, t)$ be the indices of first and the last actions of t in r. Formally, with $L = \{i \mid r(i) = (t, \alpha)\}$, $\mathsf{fst}(r, t) = \min(L)$ and $\mathsf{lst}(r, t) = \max(L)$. Let $\ll_r$ be a partial order over $Tid(r)$ ordering threads that do not execute concurrently: $t \ll_r u$ if $\mathsf{lst}(r, t) < \mathsf{fst}(r, u)$.

Definition 4 *Let P and P' be two programs with* $\mathsf{Sig}(P) = \mathsf{Sig}(P')$*, and let $\mathcal{I}$ be a state predicate. Let $X_1 = \mathsf{fv}(\mathcal{I}) \backslash Var_P$ and $X_2 = \mathsf{fv}(\mathcal{I}) \backslash Var_{P'}$. P' behaviorally-simulates P from $\mathcal{I}$, denoted $P \lhd_{\mathcal{I}} P'$ if for each maximal run r of program P from $\exists X_1.\mathcal{I}$, there exists a maximal run r' of P' from $\exists X_2.\mathcal{I}$ such that 1)* $\mathsf{Beh}(r) = \mathsf{Beh}(r')$ *and 2)* $\ll_r \subseteq \ll_{r'}$

Linearizability. We now define the notion of linearizability in [14]. In this notion a run of the program is described by a sequence of meta-actions invoke and response. The sequence of these meta-actions is called the history of the run. The history of the run r is denoted $\mathbf{H}_r$, and $\mathbf{H}(i)$ denotes the i^{th} meta-action in $\mathbf{H}$.

$Tid(\mathbf{H})$ denotes the set of thread identifiers in $\mathbf{H}$. Let $\mathbf{H}|t$ be the subhistory of $\mathbf{H}$ projected on the thread t.

An invoke action $\mathsf{inv}\langle\rho, xs, t\rangle$ describes a call to a procedure ρ with arguments xs by thread t. A response action $\mathsf{res}\langle\rho, ys, t\rangle$ describes a return from the procedure ρ with return values ys in thread t. We omit the elements of the meta-actions and refer to them using the dotted notation (e.g., $\mathsf{inv}.\rho$).

We say that $(\mathsf{inv}, \mathsf{res})$ is a *pair of matching invoke and response* in $\mathbf{H}$, if $\mathbf{H}|t(i) = \mathsf{inv}$ and $\mathbf{H}|t(i+1) = \mathsf{res}$ for some thread t and index i. We say that a history $\mathbf{H}$ is *sequential*, if for each matching pair $(\mathsf{inv}, \mathsf{res})$ in $\mathbf{H}$, $\mathbf{H}(i) = \mathsf{inv}$ and $\mathbf{H}(i+1) = \mathsf{res}$ for some index i.

We say $\mathbf{H}$ and $\mathbf{H}'$ are *equivalent*, denoted by $\mathbf{H} \cong \mathbf{H}'$, if $Tid(\mathbf{H}) = Tid(\mathbf{H}')$ and $\forall t \in Tid(\mathbf{H}).\ \mathbf{H}|t = \mathbf{H}'|t$.

Given a history $\mathbf{H}$, we define *response-invoke preservation relation* $\preceq_{\mathbf{H}}$ as follows: Two matching invoke-response pairs $(\mathsf{inv}, \mathsf{res})$ and $(\mathsf{inv}', \mathsf{res}')$ in $\mathbf{H}$ are ordered, denoted, $(\mathsf{inv}, \mathsf{res}) \preceq_{\mathbf{H}} (\mathsf{inv}', \mathsf{res}')$, if $\mathbf{H}(i) = \mathsf{res}$ and $\mathbf{H}(j) = \mathsf{inv}'$ for some $i < j$.

A history $\mathbf{H}$ is linearizable to a sequential history $\mathbf{H}'$ if $\mathbf{H} \cong \mathbf{H}'$ and $\preceq_{\mathbf{H}} \subseteq \preceq_{\mathbf{H}'}$.

Definition 5 (Linearizability) *Let P' be an atomic program and $\mathcal{I}$ be an assert predicate. A program P is* linearizable to P' from $\mathcal{I}$ *if every history of P from $\mathcal{I}$ is linearizable to some sequential history of P' from $\mathcal{I}$.*

The following theorem connects behavioral simulation to the notion of linearizability. We say *P is linearizable to P' from $\mathcal{I}$* to restrict the definition of linearizability to runs of P and P' from $\mathcal{I}$. A program P is called an *atomic program* if for every $\rho \in Proc_P$, body_ρ is an atomic action.

Theorem 2 *Let P' be an atomic program that is good from $\mathcal{I}$. A program P is* linearizable to P' *from $\mathcal{I}$ iff $P \lhd_{\mathcal{I}} P'$.*

PROOF.

Assume P is *linearizable to* P' from $\mathcal{I}$. Let r be a run of P from $\mathcal{I}$. Consider its history $\mathbf{H}_r$. By assumption, there exists a sequential history $\mathbf{H}'$ of P' from $\mathcal{I}$ such that $\mathbf{H}$ is linearizable to $\mathbf{H}'$. Then there exists a run r' of P'. We obtain $\mathbf{H}'$ from r' such that $Tid(\mathbf{H}) = Tid(\mathbf{H}') = Tid(r) = Tid(r')$. In addition, $\mathsf{Beh}(r') = \mathsf{Beh}(r)$ since the input and output values from $\mathbf{H}$ and $\mathbf{H}'$ are equivalent. In addition, $\preceq_{\mathbf{H}} \subseteq \preceq_{\mathbf{H}'}$ implies $\ll_r \subseteq \ll_{r'}$. Thus for every r, we can find a run r' such that $P \lhd_{\mathcal{I}} P'$.
Assume $P \lhd_{\mathcal{I}} P'$. Let $\mathbf{H}$ be history of program P. We have a run r of P from which we obtain $\mathbf{H}$. Since $P \lhd_{\mathcal{I}} P'$, there exits a run r' of P' such that $\mathsf{Beh}(r) = \mathsf{Beh}(r')$ and $\ll_r \subseteq \ll_{r'}$. We first construct sequential history $\mathbf{H}'$ of r' of atomic program P' as described in the definition of a history by means of invocation and return of procedures in r'. Since $\mathsf{Beh}(r) = \mathsf{Beh}(r')$ we know that $Tid(r) = Tid(r')$ and hence $Tid(\mathbf{H}) = Tid(\mathbf{H}')$ and $\forall t \in Tid(r)$, $\mathsf{beh}(r,t) = \mathsf{beh}(r',t)$. Thus, $\forall t \in Tid(\mathbf{H})$ $\mathbf{H}|t = \mathbf{H}'|t$. That is, $\mathbf{H} \cong \mathbf{H}'$. We also have $\ll_r \subseteq \ll_{r'}$ which implies $\preceq_{\mathbf{H}}$ $\subseteq \preceq_{\mathbf{H}'}$ since we do not change order of actions when deriving the history of a run. Hence, $\mathbf{H}$ is linearizable to $\mathbf{H}'$, and P is linearizable to P'.

To prove the soundness theorem, first we need to prove some lemmas.

Below, we extend the definition of state transitions as follows:

- $\sigma_1 \xrightarrow{l} \sigma_2$ holds if there exists ϵ_1, ϵ_2 such that $(\sigma_1, \epsilon_1) \xrightarrow{l} (\sigma_2, \epsilon_2)$
- $\epsilon_1 \xrightarrow{l} \epsilon_2$ holds if there exists σ_1, σ_2 such that $(\sigma_1, \epsilon_1) \xrightarrow{l} (\sigma_2, \epsilon_2)$

Note that, $\sigma_1 \xrightarrow{l} \sigma_2$ and $\epsilon_1 \xrightarrow{l} \epsilon_2$ imply $(\sigma_1, \epsilon_1) \xrightarrow{l} (\sigma_2, \epsilon_2)$.

Definition 6 (Behaviorally-simulating runs) *Let P and P′ be two programs with* $\mathsf{Sig}(P) = \mathsf{Sig}(P')$*, and let* $\mathcal{I}$ *be a state predicate. Let r and r′ be maximal runs of P and P′, respectively. The run r′ is said to* behaviorally-simulate r*, denoted* $r \blacktriangleleft r'$*, if 1)* $\mathsf{Beh}(r) = \mathsf{Beh}(r')$*, and 2)* $\ll_r \subseteq \ll_{r'}$.

For statement c, we define $c[\beta/\alpha]$ be the statement in which every occurrence of α is replaced by β. In addition, $\epsilon[\beta/\alpha]$ and $P[\beta/\alpha]$ are defined to be ϵ and P in which the same replacement is applied to every $\epsilon(t)$ and *body*(ρ), respectively.

The following lemma states that after replacing α with β in the program, the sequences of ϵ-transitions remain the same.

Lemma 1 *Let P be a program. The following holds.*

- *For $t \neq u$, if $\epsilon_1 \xrightarrow{(t,\alpha)(u,\beta)} \epsilon_2$ be a run of P, then $\epsilon_1 \xrightarrow{(u,\beta)(t,\alpha)} \epsilon_2$ is a run of P.*
- *If $\epsilon_1 \xrightarrow{(t,\alpha)} \epsilon_2$ be a run of P, then $\epsilon_1[\beta/\alpha] \xrightarrow{(t,\alpha)} \epsilon_2[\beta/\alpha]$ is a run of $P[\beta/\alpha]$.*
- *If $\epsilon_1 \xrightarrow{(t,\alpha)(t,\beta)} \epsilon_2$ be a run of P, then $\epsilon_1[\gamma/(\alpha;\beta)] \xrightarrow{(t,\gamma)} \epsilon_2[\gamma/(\alpha;\beta)]$ is a run of $P[\gamma/(\alpha;\beta)]$.*
- *If $\epsilon_1 \xrightarrow{(t,\alpha)} \epsilon_2$ be a run of P, then $\epsilon_1[\gamma/(\alpha\Box\beta)] \xrightarrow{(t,\gamma)} \epsilon_2[\gamma/(\alpha\Box\beta)]$ is a run of $P[\gamma/(\alpha;\beta)]$.*
- *If $\epsilon_1 \xrightarrow{(t,\beta)} \epsilon_2$ be a run of P, then $\epsilon_1[\gamma/(\alpha\Box\beta)] \xrightarrow{(t,\gamma)} \epsilon_2[\gamma/(\alpha\Box\beta)]$ is a run of $P[\gamma/(\alpha;\beta)]$.*
- *If $\epsilon_1 \xrightarrow{(t,\alpha_1)...(t,\alpha_n)} \epsilon_2$ be a run of P, then $\epsilon_1[\beta/\alpha^{\circlearrowleft}] \xrightarrow{(t,\beta)} \epsilon_2[\beta/\alpha^{\circlearrowleft}]$ is a run of $P[\beta/\alpha^{\circlearrowleft}]$.*

PROOF.

This lemma holds from the operational semantics of the language given in Section 2.2.1.

Lemma 2 $\blacktriangleleft$ *is reflexive and transitive.*

PROOF.

We know that equivalence on Beh and $\subseteq$ relation on $\ll$ are both reflexive and transitive. Thus, $\blacktriangleleft$ is trivially reflexive and transitive.

The following lemma states that final state of a run determines the behavior of that run.

Lemma 3 (Final state) *Let $(P, \mathcal{I})$ be a proof state such that P is good from $\mathcal{I}$. Let $r = r_1 \xrightarrow{\mathbf{l}} r_n$ and $r' = r'_1 \xrightarrow{\mathbf{l}'} r'_m$ are two maximal runs of P from $\mathcal{I}$ such that $Tid(r) = Tid(r')$ and every thread calls the same procedure in r and r'. If $\sigma_{r_n} = \sigma_{r'_m}$, then $\mathsf{Beh}(r) = \mathsf{Beh}(r')$.*

PROOF.

The behaviors are encoded by input and output variables of the threads, which are all local. For each thread $t \in Tid(r)$ let ρ be the procedure called by t. The input variables $\overrightarrow{in}_\rho$ are read-only, and output variables $\overrightarrow{out}_\rho$ are only written by t. Thus, the behavior $\mathsf{beh}(r, t)$ can be determined by only looking at the values $\overrightarrow{in}_\rho$ and $\overrightarrow{out}_\rho$ at r_n and r_m. Thus, the equivalence of the final states implies the equivalence of the behaviors.

Lemma 4 (Right-mover) *Let $(P,\mathcal{I})$ be a proof state such that P is good from $\mathcal{I}$, and α be action that is right-mover. Let $r = r_1 \xrightarrow{\mathbf{l_1}} r_i \xrightarrow{(t,\alpha)\mathbf{l}} r_j \xrightarrow{(t,\gamma)\mathbf{l_2}} r_n$ be a maximal run of P from $\mathcal{I}$., such that $\mathbf{l} = (u_1,\beta_1)(u_2,\beta_2)\cdots(u_m,\beta_m)$ and $t \notin \{u_1,\ldots,u_m\}$. There exists a maximal run $r' = r_1 \xrightarrow{\mathbf{l_1}} r_i \xrightarrow{\mathbf{l}(t,\alpha)} r_j \xrightarrow{(t,\gamma)\mathbf{l_2}} r_n$ of P from $\mathcal{I}$, such that $r \blacktriangleleft r'$.*

PROOF.

Assume that $(P,\mathcal{I}) \vdash \alpha : \mathbb{R}$. We will do induction on the length m of the sequence $\mathbf{l} = (u_1,\beta_1)(u_2,\beta_2)\cdots(u_m,\beta_m)$. In the base case, that is when $m = 0$, the claim trivially holds since we take $r = r'$. In the inductive case, we consider the run $r = r_1 \xrightarrow{\mathbf{l_1}} r_i \xrightarrow{(t,\alpha)\mathbf{l}} r_k \xrightarrow{(u_{m+1},\beta_{m+1})} r_j \xrightarrow{(t,\gamma)\mathbf{l_2}} r_n$ such that $t \notin \{u_1,\ldots,u_m,u_{m+1}\}$. By the inductive hypothesis, there exists a run $r'' = r_1 \xrightarrow{\mathbf{l_1}} r_i \xrightarrow{\mathbf{l}(t,\alpha)} r_k \xrightarrow{(u_{m+1},\beta_{m+1})} r_j \xrightarrow{(t,\gamma)\mathbf{l_2}} r_n$ such that $r'' \blacktriangleleft r$. We want to show that there exists $r' = r_1 \xrightarrow{\mathbf{l_1}} r_i \xrightarrow{\mathbf{l}(u_{m+1},\beta_{m+1})(t,\alpha)} r_j \xrightarrow{(t,\gamma)\mathbf{l_2}} r_n$. Let r'' be as follows: $r'' = r_1 \xrightarrow{\mathbf{l_1}} r_i \xrightarrow{\mathbf{l}} r_z \xrightarrow{(t,\alpha)} r_k \xrightarrow{(u_{m+1},\beta_{m+1})} r_j \xrightarrow{(t,\gamma)\mathbf{l_2}} r_n$. We know that $\sigma_{r_z} \models \phi_{\beta_{m+1}} \wedge \mathsf{wp}(\tau_{\beta_{m+1}}, \phi_\alpha)$, since P is good from $\mathcal{I}$. In addition, $\sigma_{r_z} \xrightarrow{(u_{m+1},\beta_{m+1})(t,\alpha)} \sigma_{r_j}$ holds, since $(P,\mathcal{I}) \vdash \alpha : \mathbb{R}$, and $\tau_{\beta_{m+1}} \cdot \tau_\alpha \Rightarrow \tau_\alpha \cdot \tau_{\beta_{m+1}}$. Then, $r_z \xrightarrow{(u_{m+1},\beta_{m+1})} r_s \xrightarrow{(t,\alpha)} r_j$ for some r_s holds.
Next, we want to show that $r \blacktriangleleft r'$. $r \blacktriangleleft r''$ holds by the inductive hypothesis. Since the final states of r'' and r' are equivalent, $\mathsf{Beh}(r'') = \mathsf{Beh}(r')$ by Lemma 3. The only way to break a relation in $\ll_{r''}$, by commuting (t,α) with (u_{m+1},β_{m+1}) is when α is the last action of t and β_{m+1} is the first action of u_{m+1}. However, this contradicts with the existence of γ. Thus $\ll_{r''} \subseteq \ll_{r'}$. Then, $r \blacktriangleleft r'$ since $r'' \blacktriangleleft r'$, $r'' \blacktriangleleft r'$, and $\blacktriangleleft$ is transitive.

Lemma 5 (Left-mover) *Let $(P,\mathcal{I})$ be a proof state such that P is good from $\mathcal{I}$, and γ be action that is left-mover. Let $r = r_1 \xrightarrow{\mathbf{l_1}(t,\alpha)} r_i \xrightarrow{\mathbf{l}(t,\gamma)} r_j \xrightarrow{\mathbf{l_2}} r_n$ be a successful, maximal run of P from $\mathcal{I}$., such that $\mathbf{l} = (u_1,\beta_1)(u_2,\beta_2)\cdots(u_m,\beta_m)$ and $t \notin \{u_1,\ldots,u_n,u_{n+1}\}$. There exists a maximal run $r' = r_1 \xrightarrow{\mathbf{l_1}(t,\alpha)} r_i \xrightarrow{(t,\gamma)\mathbf{l}} r_j \xrightarrow{\mathbf{l_2}} r_n$ of P from $\mathcal{I}$, such that $r \blacktriangleleft r'$.*

PROOF.

The proof is similiar to that of Lemma 4

The following main soundness theorem states that each good program reached during the proof behaviorally simulates the initial program.

Theorem 3 (Soundness) *Let $(P_1, \mathcal{I}_1) \dashrightarrow \cdots \dashrightarrow (P_n, \mathcal{I}_n)$ be a sequence of proof steps such that P_n is good from $\mathcal{I}_n$. Then forall $1 \leqslant i \leqslant n$, $P_1 \lhd_{\mathcal{I}_n} P_i$ holds.*

PROOF.

Since P_n is good from $\mathcal{I}_n$, for all $1 \leqslant i \leqslant n$, P_i is good from $\mathcal{I}_n$, by Theorem 1. We will prove the theorem by induction on the length of the proof, i.e. the number of proof states reached during the proof. In the base case, there is only one proof state $(P_1, \mathcal{I}_1)$. The claim holds trivially for $(P_1, \mathcal{I}_1)$, since $P_1 \lhd_{\mathcal{I}_1} P_1$, by definition of $\lhd$ and Lemma 2.

In the inductive case, we proceed as follows: Consider part of proof: $(P_1, \mathcal{I}_1) \dashrightarrow \cdots \dashrightarrow (P_i, \mathcal{I}_i) \dashrightarrow (P_{i+1}, \mathcal{I}_{i+1}) \dashrightarrow \cdots \dashrightarrow (P_n, \mathcal{I}_n)$. We know that for all $1 \leqslant j \leqslant i$, $P_1 \lhd_{\mathcal{I}_n} P_j$ holds, by inductive hypothesis. We want to show $P_1 \lhd_{\mathcal{I}_n} P_{i+1}$. For any $1 \leqslant k \leqslant n$, let $X_k = Var_{P_k} \backslash \mathsf{fv}(\mathcal{I}_n)$. Let run $r = r_1 \xrightarrow{1} r_N$ of P_1 from $\exists X_1 . \mathcal{I}_n$. We claim that there exists a run r' of P_{n+1} from $\exists X_{i+1} . \mathcal{I}_n$ such that $r \blacktriangleleft r'$. We will prove by case split on the proof rule.

CASE: Rule INVARIANT: Replace invariant $\mathcal{I}_1$ with $\mathcal{I}_2$ if $\alpha \leftrightarrows \mathcal{I}_2$ for all the actions α in ***P***, and $\mathcal{I}_2 \Rightarrow \mathcal{I}_1$.

Since $P_i = P_{i+1}$, so taking $r' = r$ makes the claim hold, since $r \blacktriangleleft r$ by Lemma 2.

CASE: Rule ABSTRACT: Replace the action α with action β if $\beta \leftrightarrows \mathcal{I}$ and $(P, \mathcal{I}) \vdash \alpha \preceq \beta$.

We know $(P_i, \mathcal{I}_i) \vdash \alpha \preceq \beta$. By proof rule, $\mathcal{I}_{n+1} = \mathcal{I}_n$ and $X_i = X_{i+1}$, since $Var_{P_i} = Var_{P_{i+1}}$. We construct r' from r as follows: Take $r'_1 = (\sigma_{r_1}, \epsilon_{r_1}[\beta/\alpha])$. Assume that $r'_1 \xrightarrow{r'}_j$ so that $\forall 1 \leqslant k \leqslant j.\ r'_j = (\sigma_{r_j}, \epsilon_{r_j}[\beta/\alpha])$ and transition $r_j \xrightarrow{(t,\gamma)} r_{j+1}$. We claim $(\sigma_{r_j}, \epsilon_{r_j}[\beta/\alpha]) \xrightarrow{(t,\gamma)} (\sigma_{r_{j+1}}, \epsilon_{r_{j+1}}[\beta/\alpha])$. If $\gamma \neq \alpha$, the claim trivially holds by operational semantics of atomic actions, and Lemma 1. Then, let $\gamma = \alpha$. Since $(P, \mathcal{I}) \vdash \alpha \preceq \beta$, and P is good from $\mathcal{I}$. $\sigma_{r_j} \xrightarrow{(t,\beta)} \sigma_{r_{j+1}}$. We also know that $\epsilon_{r_j}[\beta/\alpha] \xrightarrow{(t,\beta)} \epsilon_{r_{j+1}}[\beta/\alpha]$, by Lemma 1. Therefore,$(\sigma_{r_j}, \epsilon_{r_j}[\beta/\alpha]) \xrightarrow{(t,\alpha)} (\sigma_{r_{j+1}}, \epsilon_{r_{j+1}}[\beta/\alpha])$. Next, for this case we want to show $r \blacktriangleleft r'$. By the construction above, $Tid(r) = Tid(r')$, and every procedure executes the same procedure in both r and r'. Also, $\forall 1 \leqslant j \leqslant N.\ r_j = r'_j$. Thus, $\mathsf{Beh}(r) = \mathsf{Beh}(r')$ by Lemma 3. For every thread $t \in Tid(r)$, $\mathsf{fst}(r,t) = \mathsf{fst}(r',t)$ and $\mathsf{lst}(r,t) = \mathsf{lst}(r',t)$. Thus $\ll_r = \ll_{r'}$. Since $X_i = X_{i+1}$, $r_1 \models \exists X_n . \mathcal{I}_n$ implies $r'_1 \models \exists X_{i+1} . \mathcal{I}_n$. Thus r' is a run of P_{i+1} from $\exists X_{i+1} . \mathcal{I}_n$ such that $r \blacktriangleleft r'$.

CASE: Rule REDUCE-SEQUENTIAL: Replace occurrences of $\alpha\ ;\ \gamma$ with $[\![\alpha\ ;\ \gamma]\!]$ if either $(P, \mathcal{I}) \vdash \alpha : \mathbb{R}$ or $(P, \mathcal{I}) \vdash \gamma : \mathbb{L}$.

Because of the proof rule and $Var_{P_i} = Var_{P_{i+1}}$, we know that $\mathcal{I}_{n+1} = \mathcal{I}_n$ and $X_i = X_{i+1}$. We will proceed the proof for this case considering the cases where α is right-mover and γ is left-mover.

CASE: $(P_i, \mathcal{I}_i) \vdash \alpha : \mathbb{R}$

We will construct r'', a run of P_i from r, such that $r \blacktriangleleft r''$ by induction on the number of (α, γ) pairs not appearing adjacent in r. In the base we can simply have $r'' = r$, so $r \blacktriangleleft r''$. In the inductive case, we assume $r'' = r_1 \xrightarrow{\mathbf{l_1}} r_i \xrightarrow{(t,\alpha)\mathbf{l}(t,\gamma)} r_j \xrightarrow{\mathbf{l_2}} r_N$, such that $\mathbf{l} = (u_1, \beta_1)(u_2, \beta_2) \cdots (u_m, \beta_m)$, $t \notin \{u_1, \ldots, u_m, u_{m+1}\}$, and $r \blacktriangleleft r''$. By Lemma 4, $r''' = r_1 \xrightarrow{\mathbf{l_1}} r_i \xrightarrow{\mathbf{l}(t,\alpha)(t,\gamma)} r_j \xrightarrow{\mathbf{l_1}} r_n$ exists such that $r \blacktriangleleft r''$. We contruct the run r'' such that for every $r''(k) = (t, \alpha)$ for some thread t, $r''(k+1) = (t, \gamma)$, i.e., all (α, γ) pairs appear adjacent to each other. As a result of the induction, $r \blacktriangleleft r''$. Then, we construct r', a run of P_{i+1} from r'' by induction on the length of the run. For the base case, we simple take $r'_1 = r''_1$. For the inductive case, we have $r' = r''_1 \xrightarrow{\mathbf{l}} r''_k$ for some $\mathbf{l}$ and the transition $r''_k \xrightarrow{(t,\alpha)} r''_{k+1} \xrightarrow{(t,\gamma)} r''_{k+2}$. We claim that $(\sigma_{r''_k}, \epsilon_{r''_k}[[\![\alpha;\gamma]\!]/\alpha;\beta]) \xrightarrow{(t,[\![\alpha;\gamma]\!])} (\sigma_{r''_{k+2}}, \epsilon_{r''_{k+2}}[[\![\alpha;\gamma]\!]/\alpha;\gamma])$. By definition of $[\![\ ]\!]$, $[\![\alpha;\gamma]\!] = \texttt{assert}\,(\phi_\alpha \wedge \mathsf{wp}(\tau_\alpha, \phi_\gamma)); (\tau_\alpha \cdot \tau_\gamma)$. Since the run r is successful, $r''_k \models \phi_\alpha$, and in addition $r''_{k+1} \models \phi_\gamma$. We also know $r''_k \models \mathsf{wp}(\tau_\alpha, \phi_\gamma)$. By definition of $\cdot$, $(\tau_\alpha(r_k, r''_{k+1}) \wedge \tau_\gamma(r''_{k+1}, r''_{k+2})) \Rightarrow (\tau_\alpha \cdot \tau_\gamma)(r''_k, r''_{k+2})$. We also have $\sigma_{r''_k} \xrightarrow{(t,[\![\alpha;\gamma]\!])} \sigma_{r''_{k+2}}$ and $\epsilon_{r''_k}[[\![\alpha;\gamma]\!]/\alpha;\gamma] \xrightarrow{(t,[\![\alpha;\gamma]\!])} \epsilon_{r''_{k+2}}[[\![\alpha;\gamma]\!]/\alpha;\gamma]$, by Lemma 1. Next we need to show $r'' \blacktriangleleft r'$. By the construction we did, $Tid(r'') = Tid(r')$, and every thread executes the same procedure in both r'' and r'. Also, by construction, $r''_N = r'_N$. Then, together with Lemma 3, $\mathsf{Beh}(r'') = \mathsf{Beh}(r')$. $\ll_{r''} = \ll_{r'}$, since the construction we did does not introduce in r' any more interleavings than that of r'' between two threads. Now, $r'_1 \models \exists X_{i+1}.\mathcal{I}_n$, by $\sigma_{r_1} = \sigma_{r''_1} = \sigma_{r'_1}$, $r_1 \models \exists X_i.\mathcal{I}_n$. Thus r' is a run of P_{i+1} from $\exists X_{i+1}.\mathcal{I}_n$. $r \blacktriangleleft r'$, by $r \blacktriangleleft r''$, $r'' \blacktriangleleft r'$, and Lemma 2.

CASE: $(P_i, \mathcal{I}_i) \vdash \gamma : \mathbb{L}$

This case is similar to the right mover case, but using Lemma 5.

CASE: Rule REDUCE-CHOICE: Replace occurrences of $\alpha \,\square\, \gamma$ with $[\![\alpha \,\square\, \gamma]\!]$.

Because of the proof rule and $Var_{P_i} = Var_{P_{i+1}}$, we know that $\mathcal{I}_{n+1} = \mathcal{I}_n$ and $X_i = X_{i+1}$. We construct r', a run of P_{i+1} from r by induction on the length of the run. For the base case, we simple take $r'_1 = r_1$. For the inductive case, we have $r' = r_1 \xrightarrow{\mathbf{l}} r_k$ for some $\mathbf{l}$ and $r_k \xrightarrow{(t,\beta)} r_{k+1}$. We want to show that $(\sigma_{r_k}, \epsilon_{r_k}[[\![\alpha\square\gamma]\!]/\alpha;\gamma]) \xrightarrow{(t,\beta)} (\sigma_{r_{k+1}}, \epsilon_{r_{k+1}}[[\![\alpha\square\gamma]\!]/\alpha\square\gamma])$. If $\beta \neq \alpha$ and $\beta \neq \gamma$, then the claim trivially holds by operational semantics of atomic actions, and Lemma 1. Let $\beta = \alpha$ where the transition is part of $\alpha\square\gamma$. $[\![\alpha\square\gamma]\!] = \texttt{assert}\,(\phi_\alpha \wedge \phi_\gamma); (\tau_\alpha \vee \tau_\gamma)$, by definition of $[\![\ ]\!]$. Since P_i is good from $\exists X_i.\mathcal{I}$, $r_k \models \phi_\gamma$, and in addition to $r_k \models \phi_\alpha$. We also know that $\tau_\alpha(r_k, r_{k+1}) \Rightarrow (\tau_\alpha \vee \tau_\gamma)(r_k, r_{k+1})$ and $\sigma_{r_k} \xrightarrow{(t,[\![\alpha\square\gamma]\!])} \sigma_{r_{k+1}}$. By Lemma 1 $\epsilon_{r_k}[[\![\alpha\square\gamma]\!]/\alpha\square\gamma] \xrightarrow{(t,[\![\alpha\square\gamma]\!])} \epsilon_{r_{k+1}}[[\![\alpha\square\gamma]\!]/\alpha\square\gamma]$. A similar proof can be done when $\beta = \gamma$

where the transition is part of $\alpha\Box\gamma$. Next, we need to show $r \blacktriangleleft r'$. By the construction we did, $\mathit{Tid}(r) = \mathit{Tid}(r')$, and every thread executes the same procedure in both r and r'. Also, by construction, $r_N = r'_N$. Then, together with Lemma 3, $\mathsf{Beh}(r) = \mathsf{Beh}(r')$. $\ll_r = \ll_{r'}$, since the construction we did does not introduce in r' any more interleavings than that of r between two threads. Now, $r'_1 \models \exists X_{i+1}.\mathcal{I}_n$, by $\sigma_{r_1} = \sigma_{r'_1}$, $r_1 \models \exists X_i.\mathcal{I}_n$. Thus r' is a run of P_{i+1} from $\exists X_{i+1}.\mathcal{I}_n$.

CASE: Rule REDUCE-LOOP

We know that $\mathcal{I}_{n+1} = \mathcal{I}_n$ and $X_i = X_{i+1}$, since $\mathit{Var}_{P_i} = \mathit{Var}_{P_{i+1}}$. There are two cases where the body α is right or left mover.

CASE: $(P_i, \mathcal{I}_i) \vdash \alpha : \mathbb{R}$

We will construct r'', a run of P_i from r in the following way: For each occurrence of the pair α, γ, contruct a new run from the current one, by moving every α (due to $\alpha^{\circlearrowleft}$) to the right, until it becomes adjacent to the last occurrence of α. We claim $r \blacktriangleleft r''$. We will do induction on the number of pairs α^1, α^2, where α^2 is last iteration of the loop, that do not appear adjacent in r. In the base case, we can take $r'' = r$, then $r \blacktriangleleft r''$. In the inductive case, we take the run $r = r_1 \xrightarrow{\mathbf{l_1}} r_i \xrightarrow{(t,\alpha^1)\mathbf{l}(t,\alpha^2)} r_j \xrightarrow{\mathbf{l_2}} r_n$ such that $\mathbf{l} = (u_1, \beta_1)(u_2, \beta_2)\cdots(u_m, \beta_m)$ and $t \notin \{u_1, \ldots, u_n, u_{n+1}\}$, and construct the run $r'' = r_1 \xrightarrow{\mathbf{l_1}} r_i \xrightarrow{\mathbf{l}(t,\alpha^1)(t,\alpha^2)} r_j \xrightarrow{\mathbf{l_1}} r_n$. By Lemma 4, $r \blacktriangleleft r''$ holds. In addition, the number of pairs α, γ in r'' is less than r. We contruct r'' such that for every $r''(k) = (t, \alpha)$ for some thread t, either $r''(k)$ is the last iteration, or $r''(k+1) = (t, \alpha)$. Then, $r \blacktriangleleft r''$. Next, we construct r', a run of P_{i+1} from r'' in the following way: We will replace every occurrence of $r''_k \xrightarrow{(t,\alpha)^*} r''_{k+m}$ with $(\sigma_{r''_k}, \epsilon_{r''_k}[\alpha^{\circlearrowleft}/\beta]) \xrightarrow{(t,\beta)} (\sigma_{r''_{k+m}}, \epsilon_{r''_{k+m}}[\alpha^{\circlearrowleft}/\beta])$. If $m = 0$, since τ_β is reflexive, we have the transition $(\sigma_{r''_k}, \epsilon_{r''_k}[\alpha^{\circlearrowleft}/\beta]) \xrightarrow{(t,\beta)} (\sigma_{r''_k}, \epsilon_{r''_k}[\alpha^{\circlearrowleft}/\beta])$. If $m > 0$, by $(P_i, \mathcal{I}_i) \vdash [\![\beta; \alpha]\!] \preceq \beta$, and since P_i is good from $\exists X_i.\mathcal{I}_n$,$\sigma_{r''_k} \xrightarrow{(t,[\![\alpha;\gamma]\!])} \sigma_{r''_{k+m}}$ holds. Also by operational semantics, $\epsilon_{r''_k}[\alpha^{\circlearrowleft}/\beta] \xrightarrow{(t,\beta)} \epsilon_{r''_{k+m}}[\alpha^{\circlearrowleft}/\beta]$. Next, we want to show $\mathit{Tid}(r'') = \mathit{Tid}(r')$. Since the final states of r'' and r' are equivalent, $\mathsf{Beh}(r'') = \mathsf{Beh}(r')$ by Lemma 3. For every thread $t \in \mathit{Tid}(r'')$, replacing consecutive transitions of t with again transitions of t does not effect relative ordering of t with other threads. Thus $\ll_r = \ll_{r'}$. We also have $r_1 = r''_1$, by construction and Lemma 4. $r'_1 \models \exists X_{i+1}.\mathcal{I}_n$, by $\sigma_{r''_1} = \sigma_{r'_1}$, $r_1 \models \exists X_i.\mathcal{I}_n$. Thus r' is a run of P_{i+1} from $\exists X_{i+1}.\mathcal{I}_n$. $r \blacktriangleleft r'$, by $r \blacktriangleleft r''$ and $r'' \blacktriangleleft r'$.

CASE: $(P_i, \mathcal{I}_i) \vdash \alpha : \mathbb{L}$

This case is similar to the right mover case, but using Lemma 5.

CASE: Rule ADD-VARIABLE: Add the new variable v to Var_P, and replace every action α with β

whenever $(P, \mathcal{I}) \vdash \alpha \preceq_{+v} \beta$, which holds if the following are both valid:

$$\textbf{A1}.\quad (\mathcal{I} \wedge \neg\phi_\alpha) \;\Rightarrow\; (\forall v.\, \neg\phi_\beta) \qquad \textbf{A2}.\quad (\mathcal{I} \wedge \tau_\alpha) \;\Rightarrow\; (\forall v.\, \neg\phi_\beta \vee (\exists v'.\, \tau_\beta))$$

We know by proof rule $\mathcal{I}_{n+1} = \mathcal{I}_n$. If v is a local variable, then $X_i = X_{i+1}$; otherwise $X_i = X_{i+1} \cup \{v\}$, since $Var_{P_{i+1}} = Var_{P_i} \cup \{v\}$. We will construct r' from r by induction on the number of states in r. In the base case, If $v \in \mathit{local}_\rho$ for some procedure ρ, then take $r'_1 = (\overline{\sigma}_{r_1}, \epsilon_{r_1}[\beta/\alpha])$, where $\overline{\sigma}_{r_1} = \sigma_{r_1}[t, v \mapsto x]$ and x is some value, for every thread t calling ρ. If v is global, then take $r'_1 = (\overline{\sigma}_{r_1}, \epsilon_{r_1}[\beta/\alpha])$, where $\overline{\sigma}_{r_1} = \sigma_{r_1}[v \mapsto x]$ such that $\overline{\sigma}_{r_1} \models \exists X_{i+1}.\mathcal{I}_n$. There exists x since $\sigma_{r_1} \models X_i.\mathcal{I}_n$ and $X_i = X_{i+1} \cup \{v\}$. In the inductive case, we assume $r' = r'_1 \xrightarrow{1} r'_k$ so that forall $1 \leqslant j \leqslant k$, $r'_j = (\overline{\sigma}_{r_j}, \epsilon_{r_j}[\beta/\alpha])$, where $\overline{\sigma}_{r_j} = \sigma_{r_j}[v \mapsto x]$, with x chosen during the construction properly. Now assume $r_k \xrightarrow{(t,\alpha)} r_{k+1}$ exists. We want to show $r'_k \xrightarrow{(t,\alpha)} (\sigma_{r_{k+1}}, \epsilon_{r_{k+1}}[\beta/\alpha])$. $\overline{\sigma}_{r_k} \xrightarrow{(t,\beta)} \overline{\sigma}_{r_{k+1}}$, where $\overline{\sigma}_{r_{k+1}} = \sigma_{r_{k+1}}[v \mapsto x]$ such that $\tau_\beta(\overline{\sigma}_{r_k}, \overline{\sigma}_{r_{k+1}})$ holds. There exists x since $\tau_\alpha(\sigma_{r_k}, \sigma_{r_{k+1}})$ and by condition **A2** of $\preceq_{+v}$. By Lemma 1, $\epsilon_{r_k}[\beta/\alpha] \xrightarrow{(t,\beta)} \epsilon_{r_{k+1}}[\beta/\alpha]$. Next, we want to show $r \blacktriangleleft r'$. We know $Tid(r) = Tid(r')$. Let $V = Var_{P_i} \cap Var_{P_{i+1}}$. Our construction ensures that $\sigma_{r_N}|_V = \sigma_{r'_N}|_V$, and $\forall \rho.\ v \notin \mathit{local}_\rho$, thus $\mathsf{Beh}(r) = \mathsf{Beh}(r')$ by Lemma 3. We also know that for every thread $t \in Tid(r)$, $\mathsf{fst}(r,t) = \mathsf{fst}(r',t)$ and $\mathsf{lst}(r,t) = \mathsf{lst}(r',t)$. Thus $\ll_r = \ll_{r'}$. Now, if v is local, then $r'_1 \models \exists X_{i+1}.\mathcal{I}_n$ since $r_1 \models \exists X_i.\mathcal{I}_n$ and $X_i = X_{i+1}$. If v is global, $r'_1 \models \exists X_{i+1}.\mathcal{I}_n$ is ensured by our construction, choosing $\sigma_{r'_1}(v)$ properly to satisfy $\exists X_i.\mathcal{I}_n$. Thus r' is a run of P_{i+1} from $\exists X_{i+1}.\mathcal{I}_n$.

CASE: Rule HIDE-VARIABLE: Remove the existing variable v from the program, and replace the invariant $\mathcal{I}$ with $\exists v.\, \mathcal{I}$. Replace every action α with β whenever $(P, \mathcal{I}) \vdash \alpha \preceq_{-v} \beta$, which holds if the following are both valid:

$$\textbf{H1}.\quad (\exists v.\, \mathcal{I} \wedge \neg\phi_\alpha) \;\Rightarrow\; \neg\phi_\beta \qquad \textbf{H2}.\quad (\exists v, v'.\, \mathcal{I} \wedge \tau_\alpha) \;\Rightarrow\; (\neg\phi_\beta \vee \tau_\beta)$$

We know by the proof rule that $\mathcal{I}_{n+1} = \exists v.\mathcal{I}_n$ and $X_i = X_{i+1}$. We will construct r' from r by induction on the number of states in r. In the base case, we can take $r'_1 = (\overline{\sigma}_{r_1}, \epsilon_{r_1}[\beta/\alpha])$, where $\overline{\sigma}_{r_1} = \sigma_{r_1}|_{Var_{P_i}}$. $r_1 \models X_i.\mathcal{I}_n$, $\mathcal{I}_{n+1} = \exists v.\mathcal{I}_n$ and $X_i = X_{i+1}$ imply that $r'_1 \models X_{i+1}.\mathcal{I}_n$. In the inductive case, assume that $r' = r'_1 \xrightarrow{1} r'_k$ so that forall $1 \leqslant j \leqslant k$, $r'_j = (\overline{\sigma}_{r_j}, \epsilon_{r_j}[\beta/\alpha])$, where $\overline{\sigma}_{r_j} = \sigma_{r_j}|_{Var_{P_i}}$. Also assume that transition $r_k \xrightarrow{(t,\alpha)} r_{k+1}$ exists. We claim that $r'_k \xrightarrow{(t,\alpha)} (\sigma_{r_{k+1}}, \epsilon_{r_{k+1}}[\beta/\alpha])$. Then we have $\overline{\sigma}_{r_k} \xrightarrow{(t,\beta)} \overline{\sigma}_{r_{k+1}}$, where $\overline{\sigma}_{r_{k+1}} = \sigma_{r_{k+1}}|_{Var_{P_i}}$, since condition **H2** of $\preceq_{-v}$ implies that if $\sigma_{r_k} \xrightarrow{(t,\alpha)} \sigma_{r_{k+1}}$ holds then $\sigma_{r_k}|_{Var_{P_i}} \xrightarrow{(t,\beta)} \sigma_{r_{k+1}}|_{Var_{P_i}}$ holds. By Lemma 1, $\epsilon_{r_k}[\beta/\alpha] \xrightarrow{(t,\beta)} \epsilon_{r_{k+1}}[\beta/\alpha]$. Next, we want to show $r \blacktriangleleft r'$. We have $Tid(r) = Tid(r')$. Let $V = Var_{P_i} \cap Var_{P_{i+1}}$. Our construction ensures that $\sigma_{r_N}|_V = \sigma_{r'_N}|_V$, and $\forall \rho.\ v \notin \mathit{local}_\rho$, thus $\mathsf{Beh}(r) = \mathsf{Beh}(r')$ by Lemma 3. For every thread $t \in \mathit{Tid}(r)$, $\mathsf{fst}(r, t) = \mathsf{fst}(r', t)$ and $\mathsf{lst}(r, t) = \mathsf{lst}(r', t)$. Thus $\ll_r = \ll_{r'}$. Since $r_1 \models \exists X_i.\mathcal{I}_n$ and $X_i = X_{i+1}$, our construction ensures that $r'_1 \models \exists X_{i+1}.\mathcal{I}_n$ holds. Thus r' is a run of P_{i+1} from $\exists X_{i+1}.\mathcal{I}_n$.

Theorems 2 and 3 provide two options for proving linearizability of P_1 to the intended specification from $\mathcal{I}$, represented by an atomic program P_n. First, one can complement another proof method with ours, by first performing the proof $(P_1, \mathsf{true}) \dashrightarrow \cdots \dashrightarrow (P_k, \mathcal{I})$, and then applying her method to prove that P_k is linearizable to P_n. Once the proof passes, this implies that P_1 is also linearizable to P_n, since our transformations preserve all the behaviors of the program relevant to linearizability. Alternatively, one can keep transforming $(P_k, \mathcal{I})$ up to $(P_n, \mathcal{I})$, and complete the full proof of linearizability in our system. Note that, for the theorems to ensure soundness in these cases, one must also prove that P_k (resp. P_n) is good from $\mathcal{I}$. The latter is formalized by the following.

Corollary 4 *Let $(P_1, \mathsf{true}) \dashrightarrow \cdots \dashrightarrow (P_n, \mathcal{I})$ be a sequence of proof steps, such that P_n is an atomic program that is good from $\mathcal{I}$. Then, P_1 is linearizable to P_n from $\mathcal{I}$.*

In this chapter, we presented our work regarding proving linearizability of the concurrent data structures. We provided our formal framework for concurrent programs. We defined our formal proof system to prove linearizability of the concurrent program. In the end we provided our proof of the main soundness theorem.

Chapter 3

VERIFIYING TRANSACTIONAL PROGRAMS WITH PROGRAMMER-DEFINED CONFLICT DETECTION

3.1 Introduction

Transactional memory [12] provides atomic blocks by which programmers are able to use sequential reasoning about the correctness of the concurrent implementations. By these atomic blocks, concurrent programming and its verification become easy. This is because to verify the concurrent program consisting of these atomic blocks, we can safely ignore the interleaved executions of such atomic code blocks due to the atomicity guarantee provided by transactional memmory. In addition, atomic blocks enable programmers to use sequential verification tools.

In reality, while transactional memory implementations provide atomic blocks, these blocks are executed highly concurrent to achieve high performance. Two concurrent transactions are said to conflict if they access the same memory location and at least one of them modifies it. When blocks are being executed if a conflict occurs one of the blocks is aborted and re-executed in order to provide atomicity.

Code below is for a sorted and singly-linked list:

```
list_insert (list_t* listPtr, node_t* node):
  atomic {  // or, atomic (!WAR) {
      prev = &(listPtr->head);
      curr = prev->next;

      while (curr != NULL && curr->key < node->key){
          prev = curr;
          curr = curr->next;
      }

     node->next = curr;
     prev->next = node;
  }
```

Now assume that we have two transactions, **Tx1** and **Tx2**, trying to insert nodes concurrently into a linked list. Assume we have the following interleaving of the transactions where time flows from left to right:

```
    Tx1                         Tx2                                  Tx1
------------      ----------------------------       -------------------------------
⟨read n_left⟩ ... ⟨read n_left⟩ ⟨write to n_left⟩ ... ⟨read n_right⟩ ⟨write to n_right⟩
```

Tx1 inserts a node after node n_right which is closer to the tail of the list and **Tx2** inserts a node after node n_left which is closer to the head of the list. First, **Tx1** reads node pointers and passes node n_left. Then, **Tx2** reads node pointers and inserts a node after the node n_left. After **Tx2** finishes, **Tx1** inserts a node after n_right. This situation is called write-after-read (WAR) conflict. In such cases, many conflicts can occur and thus transactions can abort, if many different transactions concurrently try to insert nodes into the same linked list.

To gain in terms of performance, transactional memory implementations provide different ways to detect conflicts in which they ignore conflicts that do not hurt correctness of the program. For example, some implementations offer early discard mechanisms where the implementations discard certain locations from the read or write sets of the transaction in order to discard some conflicts that do not hurt correctness [25]. Some researchers suggest programmer-defined conflict detection [27]. The programmer annotates a block with '!WAR' and the hardware will execute that block by ignoring the write-after-read conflicts. This way performance gain is achieved.

When we apply the method of [27], the atomicity guarantee of the transaction memory implementation and the ability to use sequential reasoning are lost. If we use '!WAR' annotation technique of [27], **Tx1** will succesfully commit. But in this case, since the read of n_left of **Tx1** is old and is interleaved with transaction **Tx2**, we cannot say **Tx1** is executed as atomically. Thus, to prove the correctness of the program under this new programmer-defined conflict mechanism, one needs to think about concurrency and concurrent executions of the linked list implementation.

If we analyse the nature of the write-after-read conflicts, we can see that this type of conflicts does not hurt correctness. To see why, even if **Tx1** inserts a node after the insert of **Tx2**, the nodes are inserted in the correct places and the linked list is sorted and connected after the insertions. In additon, both inserted nodes are reachable from head.

In this work, we propose a verification method and formal model to prove correctness of programs under programmer-defined conflict mechanisms. Our method regains the ability to use sequential reasoning. We perform the correctness proof of the programs in two main steps. First, we perform a contract-based sequential proof on the transactional program. The specifications of the programs, which define what the correctness of the program means, consist of pre-conditons, post-conditions, loop invariants and invariants. For instance for the linked list example, the list being sorted and the node being in the list after insertion are the correctness conditons of the linked list implementation. Second, we perform what is called 'read abstraction'. We replace reads of variables on which we ignore write-after-read conflicts with reads that allow more behaviors and that are non-deterministic. For instance for the linked list example above, we replace the read of curr->next in curr = curr->next with a non deterministic read that returns a node reachable from head. Then, our main soundness theorem, which builds on Scott's fundamental theorem for TM [24], and on [9], states that this modified and abstract version of the program can be treated as atomic. After applying read abstraction, we perform a sequential verification on the abstract program which include the specifications of the original program determining the correctness of the original program. Once we proved these specifications, our main soundness theorem states that these specifications hold also for the original program.

To evaluate our method, we verified pre-conditons and post-conditions of transactions in the Genome and Labyrinth benchmarks from the STAMP [5] benchmark suite that use TM with relaxed detection of conflicts as stated in Titos et al [27] by sequential verification tools.

Our contributions are:

- We provide a formal model for programs using TM and programmer-controlled conflict detection. This formalization allows static, modular proof techniques to be used on programs using TM.

- We provide a sound rely-guarantee approach that makes it possible to use only sequential verification tools.

- Our soundness argument builds on a result by Scott and the soundness of the QED proof system for concurrent programs. Unlike QED, our approach requires *no pairwise mover checks* or iterative proof steps. Transactions become atomic by construction. This allows our proof method to use *sequential verification tools only.*

- We have demonstrated our approach on two large benchmarks from STAMP.

In the next motivation section, we will explain our approach with a running example. In section 3.3, we will formalize transactional programs and their exacutions. In section 3.4, we will define different semantics for transactional programs. In section 3.5, we will show how we formally perform abstraction on the programs. In section 3.6, we will explain the experiments we carried out. Finally, in section 3.7, we formally state and prove our main soundness theorem.

3.2 Motivation

In this section, we will explain our approach with a StringBuffer pool implementation.

3.2.1 Motivating example: StringBuffer pool

Figure 3.1 shows a program that implements a data structure for using a pool of StringBuffer objects by concurrent threads. The pool is implemented by an array of 1000 StringBuffer object pointers. The global(shared) accesses are read from the pool array at line 4 and write to pool array at line 7. A cell in the pool array is called *full* if that cell contains a non-null pointer. A cell is called *empty* if it contains a null pointer.

Allocate method returns a pointer to a StringBuffer object that is stored in a full cell of the pool or returns a newly created object if there is no full cell in the pool. The allocate method traverses the pool array. If it finds a full cell, it makes it empty and returns the value stored in the cell. Otherwise, it returns a newly created pointer. The free method is the dual of of the allocate method.

We will use the following invariant to express the programmer's correctness condition:
SBPoolInvariant: At any time, every StringBuffer object must either be in the pool, or it must have been returned by a call to allocate method and is being used by the program. In addition, a call to free method must either insert the given pointer to the pool or deallocate the corresponding object.

3.2.2 Strict detection of conflicts for the StringBuffer pool

Now assume that the allocate and free methods are treated as transactions. We can consider the execution of allocate happens in two phases. In the read phase, transaction traverses the array and finds cells being empty. In the commit phase, the transaction reads a full cell and makes it NULL and returns the value stored in that cell. Assume we have transaction **Tx1** that runs allocate method. Assume in the read phase **Tx1** reads cells 1 to 100 and writes to 101^{st} cell. When **Tx1** in its read phase, assume another transaction **Tx2** running allocate method that makes a cell empty

```
StringBuffer* pool[] = new StringBuffer[1000];

StringBuffer* Allocate() {
  StringBuffer* ptr;
  for (int i = 0; i < 1000; ++i) {
    ptr = pool[i];  // havoc(ptr);
    if (ptr != NULL) { // check if full
       // assume(ptr == pool[i]);
      pool[i] = NULL; // empty cell
      return ptr;
     }
  }
  // default operation
  return new StringBuffer();
}

void Free(StringBuffer* buff) {
  StringBuffer* ptr;
  for (int i = 0; i < 1000; ++i) {
    ptr = pool[i];  // havoc(ptr);
    if (ptr == NULL) { // check if empty
       // assume(ptr == pool[i]);
      pool[i] = buff; // fill cell
      return;
     }
  }
  // default operation
  delete buff;
}
```

Figure 3.1: StringBuffer pool example. The abstract program for sequential verification is obtained by replacing the shaded and commented-out code with the line it appears.

before **Tx1** reads from it or assume **Tx2** running free method that fills a cell after Tx1 reads from it. In these cases, conventional transactional memory implementations with strict conflict detection mechanisms will detect a conflict and abort at least one of the two transactions. However, we see these write-after-read (WAR) conflicts do not cause the violation of SBPoolInvariant.

3.2.3 Relaxed detection of conflicts for the StringBuffer pool

When we use the method of [27] to increase transactional memory performance, we annotate a transaction with '!WAR' to tell the transactional memory implementation to ignore write-after-read conflicts during the program executions. Here, we note that we do not ignore conflicts where the conflicting read is followed by a write by the same transaction to the same variable. [27] reports that the relaxed detection of conflicts provided by compiler and hardware support decreases the number of aborted transactions by 50-90$\%$.

Transactional memory implementations with relaxed detection of conflicts does not guarantee atomicity for allocate and free methods as conventional transactional memory implementations would. Conflicting reads and writes are now allowed to be interleaved and transactions still commit successfully. Thus, we need to prove the correctness of the programs under the relaxed conflict detection without assuming atomic executions of the programs.

3.2.4 Transforming the program for sequential verification

We propose a two-step recipe for our running StringBuffer pool example:

Step 1 Transform the original program P to a new program P_{Abs}, by replacing standard reads from global variables with nondeterministic read operations. In particular, replace every assignment $l = x$, where x is a global and l is a local variable, with an assignment that nondeterministically chooses a value of proper type and assigns it to l. In other examples, we will need a more limited nondeterminism. We will see examples of limited nondeterministic reads in experiments section.

Step 2 Verify "sequentially" the intended correctness condition in P_{Abs}.

In Step 1, we obtain the program P_{Abs} from Figure 3.1 by (i) replacing line 4 with the code that is shaded and commented out at the same line, and (ii) uncommenting the shaded code at line 6. The new program P_{Abs} is a sound, sequential and abstracted version of P. Our soundness theorem states that the result of the verification in Step 2 can be safely carried to P. In section 3.7, we will prove soundness theorem.

The key idea of transforming P to P_{Abs} is to incorporate the effects of the conflicting writes that are ignored by the transactional memory implementation on the global reads of a transaction that are subject to conflicts. We incorporate these effects into the code statically by rewriting these reads to nondeterministic reads. For our example, the cells of the pool array are the global (shared) variables, on which conflicts can happen. Our transformation to P_{Abs}, which is given in shaded comments, replaces assignments from pool[i] to local variable ptr at line 4 of Figure 3.1 with a completely nondeterministic assignment represented by the *havoc* statement which is a special command in the Boogie language [4]. The new nondeterministic assignment is a sound abstraction of the original deterministic assignment, because the new assignment can still assign ptr variable all the values that can be assigned by the original program. Our soundness theorem states that if SBPoolInvariant is verified on the abstract program P_{Abs}, the original program also satisfies SBPoolInvariant even when transactions are allowed to succeed despite WAR conflicts on pool[i].

When performing read abstractions on the original program, one needs to do read abstractions in a careful manner. In some cases, one may need to apply limited abstractions in the sense that there may be some condition on the nondeterministic reads, whose examples are in the experiments section. The first thing to be careful about is that we should not abstract reads from a location which the transaction later writes to. The transactional memory implementations using relaxed conflict detection does not ignore WAR conflicts in these cases. Such abstractions will not allow us to verify the correctness of the transactional programs. The assume statement at line 6 of the abstract program inserted to handle this situation. With this assumption we implement our intention that the last read from pool[i] is not abstracted. Assume statements are special statements that tell the verifier to ignore the program executions that violate them. These executions are not considered as a part of program behavior. Thus, during the verification process, executions that violate these assumptions are ignored.

The second thing to be careful about is that we found that abstracting reads to fully nondeterministic reads may be too coarse to prove the correctness of the programs as for STAMP benchmarks on which we carried out the experiments. For the benchmarks we worked on, we used an abstraction invariant which we define in Section 3.5 that expresses the effects on a read access from conflicting writes.

3.3 Transactional Programs and Executions

A program P consists of a set of transactions. Each transaction is a sequence of atomic statements. An arbitrary number of threads execute the transactions. For simplicity of exposition, each thread executes a single transaction. The variable t ranges over thread identifiers. As each thread is running a single transaction, we use t also to refer to the transaction instance being run by this thread.

Every thread executes in program order the statements in the transaction it is running. A particular execution of an atomic statement by a thread is called an *action* and results in an atomic state transition of the program. An execution of program P is obtained by a sequentially-consistent interleaving of actions by each thread in P. We write $\langle \mathbf{S} \rangle$ to express an action where $\mathbf{S}$ is a program statement being executed indivisibly. We also use α to denote an action, if its exact operation is not of interest.

We build on Scott's formalization of transactional programs [24]. We are particularly interested in reads from and writes to global variables. As is conventional when modeling TM, we assume that an action may read or write at most one global variable in a simple assignment from/to a local variable. Let x and l range over global and local variables, respectively. We denote by $\langle l := x \rangle$ a read from global variable x to local variable l, and similarly, by $\langle x := l \rangle$ a write to global x from local l. Global variable reads and writes, as well as transaction commit actions (described below) are *global actions*. Other kinds of statements, including local computations, procedure calls, branches, refer only to local variables.

Global access action $\langle \mathbf{S} \rangle$ can be annotated with *assumptions* and/or *assertions* as follows:

$$\langle \texttt{assert } \phi;\ \widetilde{\mathbf{S}};\ \texttt{assume } \psi \rangle$$

Normally $\widetilde{\mathbf{S}} = \mathbf{S}$, but we can also replace $\mathbf{S} = l := x$ with $\widetilde{\mathbf{S}} = \texttt{havoc}\ l$, which represents the assignment to local variable l of a nondeterministically chosen value from its domain. In the latter case, we use the statement `havoc` l to abstract (overapproximate) the read from global variable x as discussed later. The semantics of the assertion is that the execution goes to a designated failure state if ϕ is violated. Assumptions are used to limit nondeterminism (especially when `havoc` is used): execution can only continue if ψ is satisfied.

We express each execution of the program as a linear sequence of actions separated by program states: $E = s_0 \xrightarrow{\alpha_1}_{t_1} s_1 \xrightarrow{\alpha_2}_{t_2} s_2 \cdots \xrightarrow{\alpha_n}_{t_n} s_n$. Let $Act_E = \{\alpha_1, ..., \alpha_n\}$ be the set of actions in E. We write $Idx_E(\alpha_i)$ to refer to the index i of action α_i in the execution, and $Tr_E(\alpha_i)$ to refer to the transaction performing α_i. For integers i and j such that $i \leqslant j$, we write $[i..j]$ to refer to the set of integers $\{i, i+1, ..., j-1, j\}$. When talking about ordering between actions and states of the same execution, we overload the comparison operators over integers to actions and states, i.e., for

$\bullet \in \{<, >, \leqslant, \geqslant\}$, $\alpha_i \bullet_E \alpha_j$ (resp. $s_i \bullet_E s_j$) if $i \bullet j$. We use the standard definition of program state: a valuation from (global and local) variables to values of proper type.

A successful execution of a transaction t in E starts with the local action $\mathit{start}_E(t)$ and ends with the global (commit) action $\mathit{cmt}_E(t)$. Thus, for each action α_i of t, $\mathit{start}_E(t) \leqslant_E \alpha_i \leqslant_E \mathit{cmt}_E(t)$ holds. We write $s_0 \xrightarrow{E} s_n$ to denote that the execution leads s_0 to s_n.

Two executions E and E' are equivalent, written $E \equiv E'$, iff they satisfy the following conditions: (i) E and E' have the same set of threads running the same transactions, in particular, $\mathit{Act}_E = \mathit{Act}_{E'}$, (ii) the program order within each transaction is the same in E and E', and (iii) E and E' both lead from the same initial state to the same final state $s_0 \xrightarrow{E} s'$ and $s_0 \xrightarrow{E'} s'$.

Following Scott [24], a read action $\langle l := x \rangle$ by thread t is said to be *consistent* at a state s of the execution if its value returns the value written by either (i) the last committed write in the execution to x before the read action, or (ii) the last write to x by thread t, whichever comes later in the execution. The standard TM semantics requires consistency of a read action by thread t at two points in the execution: (i) the state reached after the read is performed, and (ii) right before $\mathit{cmt}(t)$ is performed. Programmer-defined conflict detection will relax the second consistency requirement in certain cases.

An execution is *complete* if the program runs without violating any assertions to completion all of its threads. Given an execution E, the *successful projection* of execution *successful*(E) is obtained by removing from E all actions and state transitions for aborted, failed, or uncompleted transactions. Correct TM operation requires that *successful*(E) is a consistent execution of P. Let $\mathbf{Execs}(P)$ contain all *complete and successful* executions of the program generated by the interleaving semantics. It is the largest set of executions of the program can generate with no constraints on concurrent transactions and conflicts between them. In the next section, we will explore different constraints on concurrent transactions.

3.4 Semantics under TM Concurrency Control

In this section, we define different sets of (successful) executions of a program allowed by TMs handling conflicts in various ways. We define three sets of semantics for a TM program P based on the set of executions they each allow: $\mathbf{Ser}(P) \subseteq \mathbf{Srbl}(P) \subseteq \mathbf{Rlx}(P) \subseteq \mathbf{Execs}(P)$.

Definition 7 (Serial semantics) $\mathbf{Ser}(P)$ *is the set of successful executions of P such that $E \in \mathbf{Ser}(P)$ iff every transaction t runs in E serially, i.e., all actions of t are contiguous, and not* *interleaved by actions from other transactions. Formally, $E \in \mathbf{Ser}(P)$ iff for each transaction t in E and action $start_E(t) \leqslant_E \alpha_i \leqslant_E cmt_E(t)$, $Tr(\alpha_i) = t$.*

Our definition of sequential semantics follows the usual single-global-lock semantics.

3.4.1 Semantics allowing interleavings of transactions

We say that two actions α_i and α_j from the same execution are *conflicting* with each other, if (i) $Tr(\alpha_i) \neq Tr(\alpha_i)$, (ii) α_i and α_j access the same global variable, and (iii) at least one of α_i and α_j is a write. We define each semantics by specifying where in the execution two conflicting actions are allowed to run without causing their transactions to fail. For this, we define the concept of a validity span. Given an execution E and an action α_i accessing a global variable in E, the *validity span* of α_i, denoted $ValSpan_E(\alpha_i)$, is the set of indices at which a *write* action conflicting with α_i is not allowed to take place. We will specify the set of executions a semantics allows by specifying $ValSpan_E(\alpha_i)$ for that semantics.

In the following, we define a serializable semantics.

Definition 8 (Serializable semantics) $\mathbf{Srbl}(P)$ *is the set of successful executions of P such that $E \in \mathbf{Srbl}(P)$ iff for every global action α_i of transaction t, $ValSpan_E(\alpha_i) = [i..Idx(cmt(t))]$.*

In words, once a transaction t accesses a global variable x, serializable semantics does not allow any other transaction to perform a conflicting write access to x between t's access and t's commit point. We note that, this is not the weakest possible serializable semantics but weak enough to present our technique for relaxed-conflict semantics.

In most TM implementations [12] the serializability condition above is ensured by maintaining the sets of addresses accessed (the read and write sets) by each transaction and aborting and rolling back transactions that experience disallowed conflicts.

As its name suggests, $\mathbf{Srbl}(P)$ contains executions where runs of transactions can be serialized around the commit action. The following theorem states that $\mathbf{Ser}(P)$ and $\mathbf{Srbl}(P)$ contains equivalent executions (with respect to $\equiv$).

Theorem 5 *For each $E \in \mathbf{Srbl}(P)$, there exists $E' \in \mathbf{Ser}(P)$ such that $E \equiv E'$.*

An important consequence of Theorem 5 is that, since every interleaved, serializable execution E is equivalent to a serial execution E', one can reason about the correctness of program P assuming that every transaction t runs sequentially and without considering interleavings of t with other transactions.

However, when we relax the restrictions on conflicting actions, we lose this advantage of reducing the reasoning to sequential verification. We next define another concurrency control semantics, $\mathbf{Rlx}(P)$, by changing the definition of span to allow conflicting actions to appear more frequently. This semantics specifies the executions provided by the TM with relaxed detection of conflicts using the `!WAR` annotation in [27].

Definition 9 (Relaxed-conflict semantics (!WAR)) $\mathbf{Rlx}(P)$ *is the set of executions such that* $E \in \mathbf{Rlx}(P)$ *iff for every global action* α_i *of transaction t, ValSpan*$_E(\alpha_i)$ *is given as follows:*

$$\mathit{ValSpan}_E(\alpha_i) = \begin{cases} [i..\mathit{Idx}(\mathit{cmt}(t))] & \textit{if} \quad \alpha_i \textit{ is a write action } \langle x := l \rangle \\ [i..\mathit{Idx}(\mathit{cmt}(t))] & \textit{if} \quad \alpha_i \textit{ is a read action } \langle l := x \rangle \textit{, and there is a write to } x \\ & \qquad \alpha_j \textit{, such that } \mathit{Tr}(\alpha_i) = \mathit{Tr}(\alpha_j) \textit{ and } \alpha_i <_E \alpha_j \\ \varnothing & \qquad \textit{otherwise} \end{cases}$$

In words, after transaction t reads from x, other transactions may write to x arbitrarily many times as long as t itself does not access x again. However, conflicting writes are never allowed between a write access and the corresponding commit action.

We note that more relaxed detection of conflicts than $\mathbf{Rlx}(P)$ are also possible. For example, we could unconditionally allow conflicting writes between a read access and the corresponding commit action. However, we found such semantics too permissive and thus unrealistic for the programs we worked on.

When we relax detection of conflicts as in $\mathbf{Rlx}(P)$ as given above, the same relationship between $\mathbf{Srbl}(P)$ and $\mathbf{Ser}(P)$ indicated in Theorem 5 does not hold any more between $\mathbf{Rlx}(P)$ and $\mathbf{Ser}(P)$. Thus, under the relaxed conflicts, we can no longer reason about the program sequentially. To see why, consider the following execution in (3.1) allowed by $\mathbf{Rlx}(P)$:

$$E = \quad \cdots \quad s_{i-1} \xrightarrow{\alpha_i = \langle l := x \rangle}_t s_i \quad \xrightarrow{\alpha_{i+1} = \langle x := l' \rangle}_u s_{i+1} \quad \cdots \quad \xrightarrow{cmt(t)}_t \quad \cdots \tag{3.1}$$

Assume that there is no access to x by t between α_{i+1} and $\mathit{cmt}(t)$. In an equivalent, serialized execution, we need to be able to execute α_i of t after α_{i+1} of u. Let the following be such an execution:

$$E' = \quad \cdots \quad s'_{i-1} \xrightarrow{\alpha'_i\langle x:=l'\rangle}_u s'_i \xrightarrow{\alpha'_{i+1}\langle l:=x\rangle}_t s'_{i+1} \quad \cdots \quad \xrightarrow{cmt'(t)}_t \quad \cdots \tag{3.2}$$

Let v be the value of x at state s_{i-1}. Clearly, the values of l in s_{i+1} and s'_{i+1} may not match, and we can not deduce that transaction t will behave in the same way in both executions; consequently it is not possible to prove that the final states of E and E' match. We are not able to claim that a serialized execution E' of P such that $E \equiv E'$ exists. This is why sequential reasoning about transactions becomes unsound. In the rest of this section, we will present a technique that will allow us to gain back the ability to soundly perform sequential reasoning.

3.5 Program Abstraction for Sequential Verification

Consider the execution fragment given in (3.1). We previously argued that we may not be able to obtain an equivalent execution of P by commuting $\langle l := x\rangle$ to the right of $\langle x := l'\rangle$. That is, swapping the execution order of $\langle l := x\rangle$ and $\langle x := l'\rangle$. To restore the ability to reason sequentially, we abstract, or in other words overapproximate, global read accesses of the form $\langle l := x\rangle$. This abstraction allows read actions to assign to the local variable l not only the current value of a global variable, but also other values that could be written to x by conflicting writes in a concurrent execution. When this is done to all read accesses on which conflicts are ignored, we obtain a new and more nondeterministic program P_{Abs} on which we show that one can use sequential reasoning and verify properties that carry over to the concurrent executions of the original program P. Below, we make this approach more precise.

We propose a two-step recipe:

Step 1 Transform program P to P_{Abs}, by (i) replacing reads from globals with nondeterministic read operations and (ii) annotating writes to globals with assertions.

Step 2 Verify "sequentially" the correctness of P_{Abs}, i.e., prove (i) all the assertions in P_{Abs} added in Step 1 and (ii) the desired correctness property.

We prove by Theorem 6 below that once all the assertions in P_{Abs} added in Step 1 are verified in Step 2, then a safety property can be verified sequentially in P_{Abs}. Moreover, if a safety property is verified in P_{Abs}, then it is sound to carry the result of this verification to P. We accomplish this by proving that for every interleaved execution of P there exists an equivalent serializable execution of P_{Abs}.

Abstraction invariant. The most permissive or nondeterministic abstraction of a global read statement $\langle l := x \rangle$ is replacing it with the statement $\langle \texttt{havoc}\ l \rangle$. This was the abstraction used in the StringBuffer pool example in Section 3.2.1. However, for many other programs, this abstraction is too loose or coarse and it is not possible to prove interesting properties of P_{Abs}. In order to provide finer control over how a read statement is abstracted, we allow the user to provide, for each global variable x, a predicate $Abs_x(v)$, we call the *abstraction invariant*. The argument of the predicate is the value of the variable x.We note that the abstraction invariant Abs_x may defined by referring to valuations of other variables in the current state. Section 3.6 gives examples to such invariants. Let $[\![Abs_x]\!]$ represent the set of values v such that $Abs_x(v)$ holds. Intuitively, the set $[\![Abs_x]\!]$ represents the set of values that may be written to x by other, conflicting transactions. Abstracting $\langle l := x \rangle$ to $\langle \texttt{havoc}\ l \rangle$ corresponds to an abstraction invariant of *true* and $[\![Abs_x]\!]$ being the domain of x. For instance, in the linked-list example, instead of an abstraction invariant of *true*, or a read resulting in a completely nondeterministic value, using the abstraction invariant, we constrain the result of the read to nodes that are reachable from the head of the linked list. In this way, the abstraction invariant constrains the values that a read statement $\langle l := x \rangle$ can write to l. It is important to note that the programmer only needs to provide a guess for the abstraction invariant. Our approach verifies in Step 2 the correctness of the invariant along with the desired correctness properties of the original program.

In Step 1 above, we transform program P to P_{Abs} performing the following changes for every global variable x:

1. Replace every action $\langle l := x \rangle$ with $\langle \texttt{assert}(Abs_x(x)); \texttt{havoc}\ l; assume(Abs_x(l)) \rangle$
2. Replace every action $\langle x := l \rangle$ with $\langle \texttt{assert}(Abs_x(l)); x := l \rangle$

Observe that, given a guess for the abstraction invariant, the two steps above are easily automated. The first modification replaces reads from x with a more abstract version of the action, which, instead of reading the current value of x, is now free to read any value from $[\![Abs_x]\!]$. Note that, the *assume* statement constrains the nondeterminism provided by the *havoc* statement. In order to ensure that this is a sound abstraction i.e., $[\![Abs_x]\!]$ contains the values the original read returns, we also insert an assertion before the read. In the style of rely-guarantee reasoning in [9], simulating state transitions with assertion violations is a sound abstraction of an action. If this assertion is discharged, then we are ensured that all values of x encountered by the original read statement fall inside the set $[\![Abs_x]\!]$, i.e., the new read statement has not removed any behaviors of the original program. We also annotate every write with an assertion. These assertions become proof obligations for the later sequential proof. When discharged, they ensure that the programmer's guess, $[\![Abs_x]\!]$, does indeed properly model all writes to x by conflicting write actions.

We note that the aim of this abstraction of reads is not to obtain an executable program but a program that can ve verified statically and be used to validate the original and unmodified program. The introduction of Abs_x and abstracting global reads and annotating global writes with assertions is a form of *rely-guarantee reasoning* [16]. Rely-guarantee reasoning for concurrent programs using abstractions in conjunction with commutativity and mover arguments, as made more precise below, is sound as was shown in work [9] on which our soundness theorem builds upon.

Theorem 6 (Soundness) *If no execution in* $\mathbf{Ser}(P_{Abs})$ *violates an assertion (i.e.,* P_{Abs} *is "sequentially" correct), then for each* $E \in \mathbf{Rlx}(P_{Abs})$*, there exists an execution* $E' \in \mathbf{Ser}(P_{Abs})$ *such that* $E \equiv E'$*.*

We provide the proof in section 3.7.

The theorem states that once we sequentially prove all the assertions we inserted in P_{Abs} at step Step 1, then we can conclude that all executions in $\mathbf{Rlx}(P_{Abs})$ have equivalent counterparts in $\mathbf{Ser}(P_{Abs})$. Thus, it is sufficient to prove a safety property such as invariants and pre- and post-conditions sequentially to reason about the correctness of the property in concurrent executions of P_{Abs}. Next, we apply this result to P.

Corollary 7 *If no execution in* $\mathbf{Ser}(P_{Abs})$ *violates an assertion (i.e.,* P_{Abs} *is "sequentially" correct), then no execution in* $\mathbf{Rlx}(P)$ *violates an assertion.*

We provide the proof in section 3.7.

The theorem states that in order prove a safety property for all concurrent executions of program P under transactional memory implementations with relaxed detection of conflicts with !WAR annotation, the user can carry out this verification over and the abstract program considering only serialized executions of P_{Abs}.

3.6 Experiments

We now demonstrate our verification technique on two benchmarks from Stanford Transactional Applications for Multi-Processing (STAMP) [5], a widely-used collection of concurrent benchmark programs containing pre-annotated transactional code blocks. Using the relaxed conflict detection !WAR annotations reported in the work of Titos et al. [27], for the Genome and Labyrinth benchmarks, we statically verified correctness of transactions that make use of programmer-defined conflict detection.

We carried out the correctness proofs of the transactions in two steps. First, we did a standard, contract-based sequential proof on the transactions in our benchmarks. For this step, we used HAVOC [19] for the linked list implementation in Genome and VCC [7] for Labyrinth, which are state-of-the-art modular verification tools for C programs. We note that we opted for using a different tool for each benchmark, since the features and logical theories required for the benchmarks are not all supported by the same tool. The contracts we had to write manually include invariants, pre- and post-conditions and loop invariants. In the second step, we applied the abstraction technique described in Section 3.5 on the benchmarks and repeated on the abstract program the sequential verification of contracts and specifications that constituted the desired correctness specification for the original program. By proving the abstract program sequentially and relying on our soundness theorem, we have showed that the original program satisfies the contracts not only in the sequential context but also in the concurrent context when running under a TM ignoring WAR conflicts. We found that the sequential verification of these benchmarks after the abstraction did not require any extra or stronger contracts compared to the original sequential proof. This indicates that the arguments about why the transactions work under relaxed detection conflicts originates from the arguments about why these transactions work sequentially. Our work validates the claims in [27] about the correctness of the transactions in the benchmarks and provides evidence that the intuitive reasoning about why programs can function correctly under TM relaxations can be expressed and verified systematically and sequentially with moderate effort.

We will be using the following shorthands in the proofs of the two benchmarks:

```
CheckNDAssign (T* ptr, T value) = assert(Abs(value)); havoc(ptr); assume(Abs(*ptr));
CheckAssign (T* ptr, T value)   = assert(Abs(value)); *ptr = value;
```

3.6.1 Linked list in Genome

```
   struct node_t { int key; node_t* next; }
   struct list_t { node_t* head; }

1  bool list_insert(list_t *listPtr, node_t *node) {
2    atomic (!WAR) {
3      node_t *prev, *curr = listPtr->head; int key = node->key;
4
5      do {
6        prev = curr;
7        curr = curr->next; // CheckNDAssign (&curr, curr->next, listPtr, key);
8      } while (curr != NULL && curr->key < key);
9
10     if (curr != NULL && curr->key == key)
11       return false; // key was present
12
13       // assume(prev->next == curr);
14     node->next = curr; // CheckAssign (&node->next, curr, listPtr, key);
15     prev->next = node; // CheckAssign (&prev->next, node, listPtr, key);
16     return true; // key was not present
17   }
18 }
```

Figure 3.2: The insertion operation of a sorted linked list. The abstract program for sequential verification is obtained by replacing the shaded and commented-out code with the line it appears.

Figure 3.2 shows the pseudocode for a linked list-based set implementation, which is used in the Genome bioinformatics and DNA gene sequencing benchmark of STAMP. We have restructured the code for simplicity of presentation. The figure shows the list_insert operation; while the remove operation is not used in Genome, the same reasoning for list_insert would be valid for removals from the list. In the code, the only shared and mutable variables are the next fields of list nodes; the key and head fields are shared but immutable.

The linked list stores a set of keys in sorted ascending order, and the list_insert operation adds a new node to the list preserving this ordering. Given a node to be inserted, list_insert follows the two-phase transaction pattern we discussed before: In the read phase (lines 5-8), it traverses the list by following the next pointers of the nodes until it finds a node curr whose key is $\geqslant$ node->key, or it reaches NULL at the end of the list. In the commit phase, list_insert either returns false (lines 10-11) indicating that node->key already exists in the list, or it inserts node to the list.

In the Genome benchmark, the body of list_insert is marked as a TM transaction annotated with

!WAR to ignore the write-after-read conflicts. Titos et al. [27] claimed that list_insert in this case operates as intended. Here, the programmer's intention is expressed as a post-condition for list_insert , which dictates that the code preserves the existing structure (e.g., sortedness) of the linked list and inserts the given node properly i.e., it returns true iff there was no node with the same key in the list and node was inserted in the list. We note that proving this post-condition in the concurrent context (due to the relaxation WAR of conflicts) corresponds to proving the linearizability of the list_insert operation, where the post-condition is a stronger version of the sequential specification for a set of keys. While proving linearizability for concurrent objects has been well-studied [14, 21, 8, 29], techniques for such proofs require complicated and error-prone reasoning about certain types of noninterference. In this paper, we aim to simplify this reasoning about noninterference using abstraction and prove the post-condition sequentially.

We will first explain the sequential proof of the linked list implementation which is done by one of our co-authors of CAV 2012 submission. Then we will continue with program abstraction phase.

The sequential proof of the linked list insert implementation. Code below constitutes the implementation, specification and verification of the insert method. Lines 5 to 43 we define the required structures and auxiliary fields, sets and addresses for the verification process. In particular, most importantly, we define set of reachable nodes in line 36 and 38 by using HAVOC environment. In lines 49 to 61, we define the invariant for the list data struture in steps(parts). Each step is described by comments in the code. From line 65 to 71, we take the conjunction of the parts of the invariant to constitute the final version of the invariant to be used in pre-conditions, post-conditions and loop invariants. Lines 74-77 define auxiliary containment functions to be used in verification process. After defining two shorthands inline 81 and 82, we finally specify the insert function in lines 85-89. The specification states that insert method expects a list that satisfies the list invariant as a pre-condition. As the post-condition, lines 86 and 87 state that after the insert method list invariant is guaranteed and the new list contains the data. Lines 88 and 89 state the frame conditions of the insert method in which one expect to see what part of the memory the method may modify. Thus, it is expected that after the insertion the size of the list is incremented and the dataset of the list now expands to contain the data being inserted. The verification of the insert method is based on the proving loop invariants to the loop in lines 108-115. The loop invariant provided is the following: Line 102 states that list invariant is maintained throughout the loop. Line 103 states that prev pointer is neither NULL nor not allocated. Line 104 states that prev pointer is reachable from the head of the list during the loop. Line 105 states that next pointer of prev pointer is curr pointer throughout the loop. Finally line 106 states that prev pointer's data is less than the data that is being inserted during the loop. After providing these loop invariants, the verifier is able to prove the post-condition we explained above. Thereby, we finished the sequential proof of the insert method.

```
1  #include <windows.h>
2  #include <havoc.h>
3  #include <malloc.h>
4
5  #define bool int
6  #if !(TRUE)
7  #error "TRUE is not 1"
8  #endif
9  #if (FALSE)
10 #error "FALSE is not 0"
11 #endif
12
13 #define data_t long
14
15 typedef struct list_node {
16   data_t data;
17   struct list_node* next;
18 } list_node_t, *plist_node_t;
19
20 typedef struct list {
21   list_node_t head;
22   size_t size;
23 } list_t, *plist_t;
24
25
26 // declare bound variables to use in __forall
27 __declare_havoc_bvar_type(_H_x, plist_node_t);
28 __declare_havoc_bvar_type(_H_y, plist_node_t);
29
30 // offset of next field
31 #define __offnext __field(plist_node_t,next)
32 // offset of data field
33 #define __offdata __field(plist_node_t,data)
34
35 // list of nodes accessible from h (inclusive) following next field until NULL
36 #define __listnext(h)  __btwn(__offnext, h, NULL)
37 // h1 can reach h2 by following next field
38 #define __reach(h1,h2) __setin(h2, __listnext(h1))
39
```

```
// set of data field addresses of nodes in S
#define __dataSet(S) __set_incr(S, __offdata)
// set of next field addresses of nodes in S
#define __nextSet(S) __set_incr(S, __offnext)

//////////////////////////////////////////////////
// list invariant (parameterized by head h)
//////////////////////////////////////////////////
// head is not null and allocated
#define list_inv1(h) (h != NULL && __allocated(h))
// head is in the list
#define list_inv2(h) (__setin(h, __listnext(h)))
// null is in the list
#define list_inv3(h) (__setin(NULL, __listnext(h)))
// all elements (except null) are allocated
#define list_inv4(h) (__forall(_H_x, __listnext(h),  _H_x != NULL ==> __allocated(_H_x)))
// all elements (except null) are pointer to list_node_t
#define list_inv5(h) (__forall(_H_x, __listnext(h),  _H_x != NULL ==> __type(_H_x, plist_node_t)))
// list is sorted
#define list_inv6(h) (__forall(__qvars(_H_x,_H_y), __listnext(h),
                               (_H_x != NULL && _H_y != NULL && _H_x != _H_y && __reach(_H_x, _H_y))
                               ==> (_H_x->data < _H_y->data)))

// s is used to to be able to use loop_inv in different contexts
// such as __ensures, __requires, __loop_assert
#define list_inv(s,h)
  s(list_inv1(h))
  s(list_inv2(h))
  s(list_inv3(h))
  s(list_inv4(h))
  s(list_inv5(h))
  s(list_inv6(h))


#define __list_not_contains(h, d)
  __forall(_H_x, __listnext(h->next), _H_x != NULL ==> _H_x->data != d)
#define __list_contains(h, d)
  (!__list_not_contains(h,d))

// shorthands for expressing list head and size
// (make sure that name of param to list_insert is listPtr)
#define PHEAD (&(listPtr->head))
#define PSIZE (&(listPtr->size))

```

```
//////////////////////////////////////////
list_inv(__requires, PHEAD)
list_inv(__ensures, PHEAD)
__ensures(__list_contains(PHEAD, data))
__modifies(PSIZE)
__modifies(__dataSet(__listnext(PHEAD)))
//////////////////////////////////////////
bool list_insert (plist_t listPtr, long data)
{
  plist_node_t prev;
  plist_node_t node;
  plist_node_t curr;

  // head is dummy and its data is less than any other given data
  __hv_assume(PHEAD->data < data);


  __loop_invariant (
                      list_inv(__loop_assert, PHEAD)
                      __loop_assert(prev != NULL && __allocated(prev))
                      __loop_assert(__reach(PHEAD, prev))
                      __loop_assert(prev->next == curr)
                      __loop_assert(prev != PHEAD ==> prev->data < data)
                      )
    for (prev = PHEAD, curr = prev->next; curr != NULL;)
      {
                if (data <= curr->data) {
                        break;
        }
                prev = curr;
                curr = curr->next;
      }

  if ((curr != NULL) &&
      (curr->data == data)) {
    return FALSE;
  }

  node = (plist_node_t) malloc(sizeof(list_node_t));
  if (node == NULL) {
    return FALSE;
  }

  node->data = data;
  node->next = curr;

  prev->next = node;

  *PSIZE += 1;

  return TRUE;
}
```

Program abstraction on the linked list insert implementation. We note that we refer to the code in figure 3.2 in the following. We obtain the abstract version of the code from Figure 3.2 by (i) replacing the deterministic read from curr->next at line 7 and the assignments at line 14-15 with the shaded and commented out code at the same line, and (ii) uncommenting the code at line 13. Note that, both CheckNDAssign and CheckAssign are defined above using the benchmark-specific abstraction invariant. In contrast with our StringBuffer example, replacing the assignment curr = curr->next at line 7 with a fully nondeterministic assignment (havoc curr) would clearly violate the post-condition, because in this case curr after the abstract assignment would point to any node even in a separate list. But, the programmer ensures that even under WAR conflicts, the curr pointer always points to some node in the given list or NULL. To alleviate this problem, we use the following abstraction invariant and restrict the nondeterminism in the assignments.

```
Abs(node_t *node, list_t *listPtr, int key) ≡ Reachable(listPtr->head, node)
           && (forall n ∈ ReachableSet(listPtr->head, node)\{node}: n->key < key)
```

Argument node refers to the nondeterministically chosen value by the abstracted read. This abstraction invariant expresses that although curr->next may be overwritten by other transactions, those writes cannot write a pointer to a node outside the list pointed by listPtr, and the key of nodes in the list before nondeterministically chosen node cannot exceed the given key. Here, we are able to precisely encode being inside the list (i.e., reachable from the head via next pointers) using the Reachable and ReachableSet functions available in the annotation language of HAVOC [19].

As explained in Section 3.5, we use two kinds of assertions in order to ensure that the read abstraction at lines 7 is sound. First, CheckNDAssign uses an assertion to check that using the current value of curr->next (as the original, deterministic read at line 7 would use) satisfies the abstraction invariant. Second, CheckAssign at line 14-15 uses assertions to check that the values written to node->next and prev->next satisfy the abstraction invariant. Although the abstraction

```
1  atomic (!WAR){
2    // copy global grid to local grid
3    grid_copy(localGridPtr, globalGridPtr);
4    localPaths = ..... // compute a set of shortest-path candidates using local grid
5    foreach (pointVectorPtr in localPaths) {
6      // insert collected paths to global grid
7      grid_addPath(globalGridPtr, pointVectorPtr);
8      // check if global grid was added a valid path
9      assert(isValidPath(globalGridPtr, pointVectorPtr));
10   }
11 }
12 grid_copy(localGridPtr, globalGridPtr):
13 for (int i = ....)
14    localGridPtr[i] = globalGridPtr[i]; // CheckNDAssign (&localGridPtr[i], globalGridPtr[i]);
15
16 grid_addPath(globalGridPtr, pointVectorPtr):
17 for (int i = ....) {
18    int j = pointVectorPtr[i];
19    globalGridPtr[j] = GRID_POINT_FULL ; // CheckAssign (&globalGridPtr[j], GRID_POINT_FULL );
20 }
```

Figure 3.3: Transaction computing shortest paths in a grid. The abstract program for sequential verification is obtained by replacing the shaded and commented-out code with the line it appears.

invariant seems complicated, it actually shares the same intuition with the loop invariant used in the standard sequential verification. Thus, these assertions are proved without needing extra/stronger contracts.

Finally, we used the assumption at line 13 to express that the nondeterministic assignment at line 7 chooses the deterministic value (i.e., prev->next) that would be used by the assignment. This ensures that the new node is inserted at the right place in the list. Note that, if another thread writes to prev->next after the last execution of line 7, this will trigger a write-after-write conflict and will abort one of the conflicting transactions, preserving the atomicity of list_insert . This guarantee implies an essential fact about the correctness, i.e., that prev->next points to curr before and during the insertion.

3.6.2 Labyrinth

Figure 3.3 (lines 1-11) shows the pseudo-code for the core transaction in the Labyrinth benchmark, which includes 2.3K lines of C code. The transaction is executed by multiple threads to find shortest paths connecting a given set of source-destination points in a 3D maze. The 3D maze is represented by a one-dimensional array, called a grid. Each cell in the grid stores an integer indicating whether that entry is available for a path to pass through (represented by constant GRID_POINT_-EMPTY = -1), or is not available to pass through (represented by constant GRID_POINT_EMPTY = -2), or is already marked as a potential point for a path (represented by a distance measure $\geqslant 0$). The goal is to connect a set of source-destination pairs with *valid* paths. A valid path should consist of contiguous cells, not intersect with another path, and not pass through an initially-blocked cell (storing GRID_POINT_FULL). The validity of paths is checked using an assertion after they are computed and added to the global grid (line 9). We first proved this assertion in VCC [7] sequentially, in addition to 60 contracts (pre- and post-conditions, program and loop invariants), encoding several complicated C language features including pointer and modular arithmetic, required for the modular verification.

Each transaction performs its operation in three steps: First, it makes a local copy of the global grid using the grid_copy operation (read phase), then it computes, over the local grid, a set of candidate shortest paths between the given source and destination cells, and finally, it commits its paths to the global grid using the grid_addPath operation (commit phase). Each path is added to the global grid by marking the cells occupied by the path with the special value GRID_POINT_FULL . Figure 3.3 also shows simplified pseudo-code for grid_copy(lines 12-14) and grid_addPath(lines 16-20), which contain the only accesses the shared, global grid.

Next we will explain the abstraction we done on the Labyrinth. After that we will explain the sequential proof of the Labyrinth.

Program abstraction on the Labyrinth. Consider the following scenario: After a transaction Tx1 takes a snapshot of the global grid and sees a cell C in the grid as empty (at line 3), Tx2 updates the global grid by writing GRID_POINT_FULL to C (at line 7). After this, if Tx1 computes another path passing through C and attempts to add that path to the global grid. This creates a write-after-write conflict and aborts one of the transactions. Thus, the transactional memory ensures that grid_addPath does not commit to overlapping paths to the global grid. If Tx1 does not write to C later, a standard transactional memory would still trigger a write-after-read conflict and abort by transactional memory implementation, although such an execution does not lead to invalid paths and should be allowed. The only drawback of such stale reads is redundant computation for paths in aborted transactions. Therefore, the programmer can safely annotate the transaction with !WAR, which instructs the transactional memory implementation to ignore the conflicts between these accesses.

In order to verify the transaction under the relaxation of !WAR conflicts, we use the following abstraction invariant.

```
Abs(int C2, int C1) ≡ (C1 = GRID_POINT_FULL  ⇒  C2 = GRID_POINT_FULL )
                 && (C1 = GRID_POINT_EMPTY   ⇒  C2 ∈ {GRID_POINT_EMPTY ,GRID_POINT_FULL })
```

Argument C2 refers to the nondeterministically chosen value for a cell and C1 refers to the current value of that cell. The invariant states that once a path is committed to the global grid, the cells on this path are not overwritten by another transaction (first conjunct), thus a transaction can assume that full cells will remain as is. Note that this condition restricts the nondeterminism so that an initially-blocked cell cannot be read as unblocked and used in a path. However, an empty cell can be overwritten with the value GRID_POINT_FULL , thus a transaction should anticipate such writes (second conjunct).

In the abstraction of the transaction in Figure 3.3 lines 14 and 19 are replaced by the commented CheckNDAssign and CheckAssign operations, respectively. For this example, the assertion in CheckNDAssign about the original assignment passes because the cells in the global grid can only contain GRID_POINT_EMPTY or GRID_POINT_FULL and the abstraction invariant is satisfied when C1=C2, i.e., when the abstract read choses to return the current value of the regarding cell. The assertion in CheckAssign about the writes to the global grid passes, as well, since the only value written is GRID_POINT_FULL and the abstraction invariant is always satisfied when C1=GRID_POINT_FULL . We were able to prove these assertion using the same contracts from the original sequential proof.

The sequential proof of the Labyrinth. The router_solve code below is the method where the write-after-read conflicts are ignored by the transactional memory implementation indicated by CONFLICT_UNSET_WAR signature (line 46). Write-after-read conflicts on PdoTraceback method are ignored. Our correctness condition is asserted by line 55 in the code below. It states that a path vector generated by PdoTraceback should be valid. We implemented the validity of a path by isValidPath logic formula in VCC (lines 1-20 below router_solve code). The validity condition states that successor points in the path vector should be adjacent and none of the points should be empty. Thus, we need to show that the post-condition of the PdoTraceback is that the return value is a valid path if it is not null. We do this by providing loop invariants to main while loop in the PdoTraceback method. Before going into the details of the loop invariants, the method traceToNeighbor tries to move in a given direction(for instance in the positive x direction) by one unit and store the new point in the *next* pointer. The verification of the traceToNeighbor is quite straightforward so we do not include it here. If the *curr* and *next* pointers are equal, this means we could not move in any direction so we stop (line 148 in PdoTraceback code). If the value of the next is zero this means that we have reached the destination (line 80 in PdoTraceback code). In this case we return a non-null path vector. Otherwise, we return null (line 156 in PdoTraceback code). The ghost variable xCoords, yCoords and zCoords (line 22-25 in PdoTraceback code) are defined and used for path validity checking.

In the following we refer to PdoTraceback code. Lines 32-60 show the invariants we provided for PdoTraceback method to verify. First four invariants state the thread-locality of the relevant pointers and the non-nullity of the constructed path vector. The invariants at line 37 and 38 state that the *curr* and *next* pointers' values are equal if we could not move to the next pointer by traceToNeighbor. Lines 40-44 state that ghost variables indeed hold the correct values. Lines 46-48 state that in the case of not reaching the destination, the ghost variables hold current values. Lines 49-52 state that the predecessor ghost variables hold current values in the case of non-failure movement by traceToNeighbor method. The lines 54-55 are about values of some constants. Line 57 states that we reach the destination if and only if the stopping variable is 2. Line 58 and 59 states in the case of non-failure movement by traceToNeighbor method the *curr* and *next* pointers are adjacent. Line 60 states that if the stopping variable is 2, then the path is valid. This loop invariant indeed lets the verifier to verify the post-condition of the PdoTraceback.

```
void
router_solve (void* argPtr)
_(requires thread_local((router_solve_arg_t*)argPtr))
_(requires thread_local(((router_solve_arg_t*)argPtr)->mazePtr))
_(requires thread_local(((router_solve_arg_t*)argPtr)->mazePtr->gridPtr))
_(requires thread_local(((router_solve_arg_t*)argPtr)->routerPtr))

{
    router_solve_arg_t* routerArgPtr = (router_solve_arg_t*)argPtr;
    router_t* routerPtr = routerArgPtr->routerPtr;
    maze_t* mazePtr = routerArgPtr->mazePtr;
    vector_t* myPathVectorPtr = PVECTOR_ALLOC(1);

    queue_t* workQueuePtr = mazePtr->workQueuePtr;
    grid_t* gridPtr = mazePtr->gridPtr;
    grid_t* myGridPtr =
    grid_alloc(gridPtr->width, gridPtr->height, gridPtr->depth); // Call regular malloc

    long bendCost = routerPtr->bendCost;
    queue_t* myExpansionQueuePtr = PQUEUE_ALLOC(-1);

    while (1)
   {
        pair_t* coordinatePairPtr;

        if (TMQUEUE_ISEMPTY(workQueuePtr)) {
            coordinatePairPtr = NULL;
        } else {
            coordinatePairPtr = (pair_t*)TMQUEUE_POP(workQueuePtr);
        }

        if (coordinatePairPtr == NULL) {
            break;
        }
        _(assume thread_local (coordinatePairPtr))
        coordinate_t* srcPtr = (coordinate_t*)coordinatePairPtr->firstPtr;
        coordinate_t* dstPtr = (coordinate_t*)coordinatePairPtr->secondPtr;

        bool_t success = FALSE;
        vector_t* pointVectorPtr = NULL;

       // TM_BEGIN(1);

        grid_copy(myGridPtr, gridPtr);

        //CONFLICT_UNSET_WAR;
        //CONFLICT_DISABLE_CONFLICT_SIGNATURE;

        if (PdoExpansion(routerPtr, myGridPtr, myExpansionQueuePtr,
                         srcPtr, dstPtr)) {

            pointVectorPtr = PdoTraceback(gridPtr, myGridPtr, dstPtr, bendCost);

            if (pointVectorPtr) {
                _(assert isValidPath(pointVectorPtr))
                //CONFLICT_SET_WAR; // Avoid remote writes while path is being validated & added
                TMGRID_ADDPATH(gridPtr, pointVectorPtr);
                //TM_LOCAL_WRITE(success, TRUE);
            }
        }
        //TM_END(1);

        if (success) {
            bool_t status = PVECTOR_PUSHBACK(myPathVectorPtr,
                                             (void*)pointVectorPtr);
            _(assert status)
        }
    }

    list_t* pathVectorListPtr = routerArgPtr->pathVectorListPtr;
    //TM_BEGIN(2);
    TMLIST_INSERT(pathVectorListPtr, (void*)myPathVectorPtr);
    //TM_END(2);

    PGRID_FREE(myGridPtr);
    PQUEUE_FREE(myExpansionQueuePtr);
}
```

```
_(logic bool isValidPath(vector_t* myPathVectorPtr)
 =(forall int i; 0<=i && i<(myPathVectorPtr->size-1) ==>
 isAdjacent(
   myPathVectorPtr->xCoords[i],
   myPathVectorPtr->xCoords[i+1],
   myPathVectorPtr->yCoords[i],
   myPathVectorPtr->yCoords[i+1],
   myPathVectorPtr->zCoords[i],
   myPathVectorPtr->zCoords[i+1]
  )

 &&
  myPathVectorPtr->points[i]  != GRID_POINT_EMPTY
 )
 &&
  myPathVectorPtr->points[0] != 0
 &&
  myPathVectorPtr->points[myPathVectorPtr->size-1] != 0

)
_(logic bool isAdjacent(long x1, long x2, long y1, long y2, long z1, long z2)=
  ( ( (_(unchecked)(x1-x2)==1) && (y1==y2)    && (z1==z2))    ||
 ( (_(unchecked)(x2-x1)==1) && (y1==y2)    && (z1==z2))    ||
 ( (x1==x2)    && (_(unchecked)(y1-y2)==1) && (z1==z2))    ||
 ( (x1==x2)     && (_(unchecked)(y2-y1)==1) && (z1==z2))     ||
 ( (x1==x2)     && (y1==y2)     && (_(unchecked)(z1-z2)==1)) ||
 ( (x1==x2)     && (y1==y2)     && (_(unchecked)(z2-z1)==1))  )
)
```

```
static vector_t*
PdoTraceback (grid_t* gridPtr, grid_t* myGridPtr,
              coordinate_t* dstPtr, long bendCost)
_(ensures result != NULL ==> isValidPath(result))
{

   vector_t* pointVectorPtr = vector_alloc(1);

   _(assume thread_local(pointVectorPtr))
   _(assume pointVectorPtr->size==0)
   _(assume  (pointVectorPtr!=NULL))
   point_t next;
   next.x = dstPtr->x;
   next.y = dstPtr->y;
   next.z = dstPtr->z;
   next.value = grid_getPoint(myGridPtr, next.x, next.y, next.z);
   next.momentum = MOMENTUM_ZERO;
   point_t curr;
   _(assume !((curr.x == next.x) &&
            (curr.y == next.y) &&
            (curr.z == next.z)))
   _(ghost pointVectorPtr->xCoords[(pointVectorPtr->size)] = next.x;)
   _(ghost pointVectorPtr->yCoords[(pointVectorPtr->size)] = next.y;)
   _(ghost pointVectorPtr->zCoords[(pointVectorPtr->size)] = next.z;)
   _(ghost pointVectorPtr->points[pointVectorPtr->size] = GRID_POINT_FULL;)

   _(assume thread_local(pointVectorPtr))

   int stopIterating = -1;
   _(assume next.value == 0 <==> ( stopIterating == 2))
    while (stopIterating <= 1)
        _(invariant thread_local(pointVectorPtr))
        _(invariant pointVectorPtr!=NULL)
        _(invariant thread_local(&next))
        _(invariant thread_local(&curr))
        _(invariant
    ((stopIterating ==-1) ==> isEqual(dstPtr->x, next.x, dstPtr->y, next.y, dstPtr->z, next.z))
            && ((stopIterating == 1) ==> isEqual(curr.x, next.x, curr.y, next.y, curr.z, next.z))
                   && ((stopIterating ==0 || stopIterating ==2) ==>
     ( (pointVectorPtr->xCoords[(pointVectorPtr->size)] == next.x)
                && (pointVectorPtr->yCoords[(pointVectorPtr->size)] == next.y)
               && (pointVectorPtr->zCoords[(pointVectorPtr->size)] == next.z)
       )
     )

          && ((stopIterating == 1) ==> ( (pointVectorPtr->xCoords[(pointVectorPtr->size)] == curr.x)
          && (pointVectorPtr->yCoords[(pointVectorPtr->size)] == curr.y)
          && (pointVectorPtr->zCoords[(pointVectorPtr->size)] == curr.z)))
          && ((stopIterating == 0 || stopIterating ==2) ==>
          ( (pointVectorPtr->xCoords[(pointVectorPtr->size)-1] == curr.x)
               && (pointVectorPtr->yCoords[(pointVectorPtr->size)-1] == curr.y)
               && (pointVectorPtr->zCoords[(pointVectorPtr->size)-1] == curr.z)))

          && (GRID_POINT_FULL  != 0 && GRID_POINT_EMPTY != 0)
          && (pointVectorPtr->points[pointVectorPtr->size] == GRID_POINT_FULL)

          && ((next.value == 0) <==> ( stopIterating == 2))
          && ((stopIterating ==2) ==> (isAdjacent(curr.x, next.x, curr.y, next.y, curr.z, next.z)))
          &&((stopIterating ==0) ==> (isAdjacent(curr.x, next.x, curr.y, next.y, curr.z, next.z)))
    && (( stopIterating == 2) ==> isValidPath(pointVectorPtr))

    )
  {
   stopIterating = 0;

   _(assume thread_local(&next))
   _(assume thread_local(&curr))
   _(ghost pointVectorPtr->xCoords[(pointVectorPtr->size)] = curr.x;)
   _(ghost pointVectorPtr->yCoords[(pointVectorPtr->size)] = curr.y;)
   _(ghost pointVectorPtr->zCoords[(pointVectorPtr->size)] = curr.z;)
   long* gridPointPtr = grid_getPointRef(gridPtr, next.x, next.y, next.z);
   vector_pushBack(pointVectorPtr, &next);

   ...
   /* Check if we are done */
   if (next.value == 0) {
                 _(ghost pointVectorPtr->xCoords[(pointVectorPtr->size)] = next.x;)
           _(ghost pointVectorPtr->yCoords[(pointVectorPtr->size)] = next.y;)
           _(ghost pointVectorPtr->zCoords[(pointVectorPtr->size)] = next.z;)
          stopIterating = 2; // Done with success
   }
   else {
         curr = next;
                 traceToNeighbor(myGridPtr, &curr, MOVE_POSX, TRUE, bendCost, &next);
         _(ghost pointVectorPtr->xCoords[(pointVectorPtr->size)] = next.x;)
         _(ghost pointVectorPtr->yCoords[(pointVectorPtr->size)] = next.y;)
   _(ghost pointVectorPtr->zCoords[(pointVectorPtr->size)] = next.z;)
        traceToNeighbor(myGridPtr, &curr, MOVE_POSY, TRUE, bendCost, &next);
         _(ghost pointVectorPtr->xCoords[(pointVectorPtr->size)] = next.x;)
         _(ghost pointVectorPtr->yCoords[(pointVectorPtr->size)] = next.y;)
         _(ghost pointVectorPtr->zCoords[(pointVectorPtr->size)] = next.z;)
        traceToNeighbor(myGridPtr, &curr, MOVE_POSZ, TRUE, bendCost, &next);
         _(ghost pointVectorPtr->xCoords[(pointVectorPtr->size)] = next.x;)
         _(ghost pointVectorPtr->yCoords[(pointVectorPtr->size)] = next.y;)
         _(ghost pointVectorPtr->zCoords[(pointVectorPtr->size)] = next.z;)
        traceToNeighbor(myGridPtr, &curr, MOVE_NEGX, TRUE, bendCost, &next);
         _(ghost pointVectorPtr->xCoords[(pointVectorPtr->size)] = next.x;)
         _(ghost pointVectorPtr->yCoords[(pointVectorPtr->size)] = next.y;)
         _(ghost pointVectorPtr->zCoords[(pointVectorPtr->size)] = next.z;)
        traceToNeighbor(myGridPtr, &curr, MOVE_NEGY, TRUE, bendCost, &next);
         _(ghost pointVectorPtr->xCoords[(pointVectorPtr->size)] = next.x;)
         _(ghost pointVectorPtr->yCoords[(pointVectorPtr->size)] = next.y;)
         _(ghost pointVectorPtr->zCoords[(pointVectorPtr->size)] = next.z;)
        traceToNeighbor(myGridPtr, &curr, MOVE_NEGZ, TRUE, bendCost, &next);
         _(ghost pointVectorPtr->xCoords[(pointVectorPtr->size)] = next.x;)
         _(ghost pointVectorPtr->yCoords[(pointVectorPtr->size)] = next.y;)
         _(ghost pointVectorPtr->zCoords[(pointVectorPtr->size)] = next.z;)

        /*
         * Because of bend costs, none of the neighbors may appear to be closer.
         * In this case, pick a neighbor while ignoring momentum.
         */
        if ((curr.x == next.x) &&
            (curr.y == next.y) &&
            (curr.z == next.z))
        {
            next.value = curr.value;

            traceToNeighbor(myGridPtr, &curr, MOVE_POSX, FALSE, bendCost, &next);
                 _(ghost pointVectorPtr->xCoords[(pointVectorPtr->size)] = next.x;)
                 _(ghost pointVectorPtr->yCoords[(pointVectorPtr->size)] = next.y;)
                 _(ghost pointVectorPtr->zCoords[(pointVectorPtr->size)] = next.z;)
            traceToNeighbor(myGridPtr, &curr, MOVE_POSY, FALSE, bendCost, &next);
                 _(ghost pointVectorPtr->xCoords[(pointVectorPtr->size)] = next.x;)
                 _(ghost pointVectorPtr->yCoords[(pointVectorPtr->size)] = next.y;)
                 _(ghost pointVectorPtr->zCoords[(pointVectorPtr->size)] = next.z;)
            traceToNeighbor(myGridPtr, &curr, MOVE_POSZ, FALSE, bendCost, &next);
                 _(ghost pointVectorPtr->xCoords[(pointVectorPtr->size)] = next.x;)
```

```
                _(ghost pointVectorPtr->yCoords[(pointVectorPtr->size)] = next.y;)
                _(ghost pointVectorPtr->zCoords[(pointVectorPtr->size)] = next.z;)
            traceToNeighbor(myGridPtr, &curr, MOVE_NEGX, FALSE, bendCost, &next);
                _(ghost pointVectorPtr->xCoords[(pointVectorPtr->size)] = next.x;)
                _(ghost pointVectorPtr->yCoords[(pointVectorPtr->size)] = next.y;)
                _(ghost pointVectorPtr->zCoords[(pointVectorPtr->size)] = next.z;)
            traceToNeighbor(myGridPtr, &curr, MOVE_NEGY, FALSE, bendCost, &next);
                _(ghost pointVectorPtr->xCoords[(pointVectorPtr->size)] = next.x;)
                _(ghost pointVectorPtr->yCoords[(pointVectorPtr->size)] = next.y;)
                _(ghost pointVectorPtr->zCoords[(pointVectorPtr->size)] = next.z;)
            traceToNeighbor(myGridPtr, &curr, MOVE_NEGZ, FALSE, bendCost, &next);
                _(ghost pointVectorPtr->xCoords[(pointVectorPtr->size)] = next.x;)
                _(ghost pointVectorPtr->yCoords[(pointVectorPtr->size)] = next.y;)
                _(ghost pointVectorPtr->zCoords[(pointVectorPtr->size)] = next.z;)

            if ((curr.x == next.x) &&
                (curr.y == next.y) &&
                (curr.z == next.z))
            {
               stopIterating = 1; // Done with failure
            }

        }
 }

  _(ghost pointVectorPtr->points[pointVectorPtr->size] = GRID_POINT_FULL;)
    }
    return (stopIterating == 2) ? pointVectorPtr: NULL;
}
```

3.7 Soundness Theorem

Theorem 8 (Soundness) *If no execution in $\mathbf{Ser}(P_{Abs})$ violates an assertion (i.e., P_{Abs} is "sequentially" correct), then for each $E \in \mathbf{Rlx}(P_{Abs})$, there exists an execution $E' \in \mathbf{Ser}(P_{Abs})$ such that $E \equiv E'$.*

Proof: We show that in P_{Abs} every transaction is composed of right-mover actions (as defined in [9]) in the sense that, for each execution $E \in \mathbf{Rlx}(P_{Abs})$ with a transaction t, we can obtain an execution $E'' \in \mathbf{Rlx}(P_{Abs})$ in which t is serialized (for each action $start_E(t) \leqslant_E \alpha_i \leqslant_E cmt_E(t)$, $Tr(\alpha_i) = t$) . Given that, we have proved in [9] that each transaction t is atomic, and consequently, every execution $E'' \in \mathbf{Rlx}(P_{Abs})$ is equivalent to some execution $E' \in \mathbf{Ser}(P_{Abs})$. To prove the right-moverness of the actions, we use (i) the guarantees from TM about the absence of write-after-write conflicts, and (ii) the validity of the assertions in P_{Abs} that were added during the abstraction step S1. For each action $\alpha_i = \langle \mathbf{S} \rangle$ of transaction t, we do a case split on the type of the statement $\mathbf{S}$:

- $\mathbf{S}$ does not access any global variables: In this case, α_i is trivially a right-mover.

- $\mathbf{S}$ writes to a global variable x: The only problematic case is when there is a write to x by another transaction between α_i and $cmt(t)$. However, by definition $\mathit{ValSpan}_E$ for $\mathbf{Rlx}(P)$, this scenario cannot happen.

- The subtle case is when $\mathbf{S}$ is an abstracted read. In this case, one can show that the right-moverness check for α_i as explained in [9] would always succeed.

Corollary 9 *If no execution in $\mathbf{Ser}(P_{Abs})$ violates an assertion (i.e., P_{Abs} is "sequentially" correct), then no execution in $\mathbf{Rlx}(P)$ violates an assertion.*

Proof: Recall that, P_{Abs} is a sound abstraction of P (as we make some reads return more values and add extra assertions, which are all validated). Therefore, for each $E \in \mathbf{Rlx}(P)$, there exists an execution $E' \in \mathbf{Rlx}(P_{Abs})$ such that $E \equiv E'$. By Theorem 6, there also exist an execution $E'' \in \mathbf{Ser}(P_{Abs})$ such that $E' \equiv E''$. Thus $E \equiv E''$. Consequently, if E'' of P_{Abs} does not violate an assertion, then E of P does not violate an assertion, either.

Chapter 4

CONCLUSIONS

In this thesis, we study two different verification problems in the context of concurrent, multi-threaded software. First, we study how to prove the correctness of concurrent data structure implementation when the correctness condition is linearizability. Our theory and way of verification are based on the sound proof system of QED. We first adapted generic definition of linearizability to the proof system. Then, we showed that each proof step in QED preserves a certain type of simulation relation. We then showed the critical connection between the simulation relation and the linearizability. Thereby, we concluded that the concurrent implementation is linearizable to the specification we got at the end of the QED proof. To reach the specification we aim, we use two main steps of the QED proofs, namely abstraction and reduction. While abstraction in general replaces an action with one that allows more behaviors, reduction composes two atomic actions into a single atomic action given that one of them has a proper mover type. With the addition of two new rules, variable introduction and hiding, we make implementation closer to the specification we aim in each proof step. Thus, when we reach the specification at the end of the proof, our soundness theorem guarantees that the initial concurrent implementation is linearizable to the sequential specification reached.

The second part of the thesis considers the problem of verifying transactional programs that uses some transactional memory implementation which has relaxed or programmer-defined conflict detection. While introducing and supporting relaxed conflict detection are beneficial in terms of performance, the verification becomes hard to do since the ability of sequential reasoning is lost. We provided a way to model and verify such transactional programs which regains the ability to use sequential reasoning to do the verification job. Our method was based on providing the abstractions on global variable reads and assertions to global variable writes needed for the soundness. The rest of the method is automated. After providing abstractions, one can use a sequential verification tool to verify the program. If the program verifies, then we conclude that the specifications of the program, given in terms of pre-condtions and post-conditions, assertions, loop invariants, holds under the relaxed conflict detection. If not, the abstractions most probably are not correct and the programmer tries to provide other abstractions that correctly describes and takes into account the conflicting actions in the concurrent executions of the program. We note that our approach reduces the verification of the transactional program to the sequential verification.

4.1 Future Work

4.1.1 Future work on verification of transactional programs with relaxed conflict detection

One of the first improvements for our method to verify transactional programs with programmer-defined conflict detection is to automate the deduction of the abstractions provided by the programmer. The conflicting actions in the programmer can be analyzed such that the correct abstraction can be found automatically by the verifier. In addition, the assertions needed for the global writes can be added to the program in an automated manner.

Other than the automated deduction and additions of the abstractions and assertions, different types of relaxed conflict detection mechanisms can be studied and formalized, then verified. We specifically consider ignoring write-after-read conflicts in the concurrent executions of the program. However, many different types of conflicts or conflict detection approaches can be studied. Thereby, we will obtain a much more general verification approach and solution for transactional programs.

BIBLIOGRAPHY

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.

[2] D. Amit, N. Rinetzky, T. W. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In *CAV*, pages 477–490, 2007.

[3] H. Attiya, G. Ramalingam, and N. Rinetzky. Sequential verification of serializability. *SIGPLAN Not.*, 45:31–42, January 2010.

[4] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. *FMCO*, 2005.

[5] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proc. of The IEEE International Symposium on Workload Characterization*, September 2008.

[6] R. Colvin, L. Groves, V. Luchangco, and M. Moir. Formal verification of a lazy concurrent list-based set algorithm. In *CAV*, pages 475–488, 2006.

[7] M. Dahlweid, M. Moskal, T. Santen, S. Tobies, and W. Schulte. Vcc: Contract-based modular verification of concurrent c. In *ICSE-Companion 2009.*, pages 429 –430, may 2009.

[8] T. Elmas, S. Qadeer, A. Sezgin, O. Subasi, and S. Tasiran. Simplifying the proof of linearizability with reduction and abstraction. In *TACAS 2010: Proc. of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2010.

[9] T. Elmas, S. Qadeer, and S. Tasiran. A calculus of atomic actions. In *POPL '09: ACM Symposium on Principles of Programming Languages*, New York, NY, USA, 2009. ACM.

[10] H. Gao, J. F. Groote, and W. H. Hesselink. Lock-free dynamic hash tables with open addressing. *Distrib. Comput.*, 18(1):21–42, 2005.

[11] L. Groves. Verifying michael and scott's lock-free queue algorithm using trace reduction. In *CATS '08: Symposium on Computing: the Australasian theory*, pages 133–142, Darlinghurst, Australia, 2008. Australian Computer Society, Inc.

[12] T. Harris, J. R. Larus, and R. Rajwar. *Transactional Memory, 2nd edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010.

[13] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *SPAA '04: ACM symposium on Parallelism in algorithms and architectures*, pages 206–215, New York, NY, USA, 2004. ACM.

[14] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[15] W. H. Hesselink. Eternity variables to prove simulation of specifications. *ACM Trans. Comput. Logic*, 6(1):175–201, 2005.

[16] C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, June 1981.

[17] B. Jonsson, A. Pnueli, and C. Rump. Proving refinement using transduction. *Distrib. Comput.*, 12(2-3):129–149, 1999.

[18] Y. Kesten, A. Pnueli, E. Shahar, and L. D. Zuck. Network invariants in action. In *CONCUR '02: The 13th International Conference on Concurrency Theory*, pages 101–115, London, UK, 2002. Springer-Verlag.

[19] S. Lahiri and S. Qadeer. Back to the future: revisiting precise program verification using smt solvers. *SIGPLAN Not.*, 43:171–182, January 2008.

[20] R. J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[21] P. W. O'Hearn, N. Rinetzky, M. T. Vechev, E. Yahav, and G. Yorsh. Verifying linearizability with hindsight. In *Proc. of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '10, pages 85–94, New York, NY, USA, 2010. ACM.

[22] S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.

[23] S. Park and D. L. Dill. Protocol verification by aggregation of distributed transactions. In *CAV '96: The International Conference on Computer Aided Verification*, pages 300–310, London, UK, 1996. Springer-Verlag.

[24] M. L. Scott. Sequential specification of transactional memory semantics. In *ACM SIGPLAN Workshop on Transactional Computing*. Jun 2006.

[25] T. Skare and C. Kozyrakis. Early release: Friend or foe? In *Workshop on Transactional Memory Workloads*. Jun 2006.

[26] O. Subasi, T. Elmas, A. Cristal, T. Harris, S. Tasiran, R. Titos-Gil, and O. Unsal. On justifying and verifying relaxed detection of conflicts in concurrent programs. *Workshop on Determinism and Correctness in Parallel Programming (WoDet)*, March 2012.

[27] R. Titos, M. E. Acacio, J. M. Garca, T. Harris, A. Cristal, O. Unsal, and M. Valero. Hardware transactional memory with software-defined conflicts. In *(To appear) High-Performance and Embedded Architectures and Compilation (HiPEAC'2012)*, January 2012.

[28] V. Vafeiadis. Shape-value abstraction for verifying linearizability. In *VMCAI '09: The International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 335–348, Heidelberg, 2009. Springer-Verlag.

[29] V. Vafeiadis. Automatically proving linearizability. In *CAV*, pages 450–464, 2010.

[30] V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro. Proving correctness of highly-concurrent linearisable objects. In *PPoPP '06 ACM Symposium on Principles and practice of parallel programming*, pages 129–136, New York, NY, USA, 2006. ACM.

[31] L. Wang and S. D. Stoller. Static analysis for programs with non-blocking synchronization. In *ACM SIGPLAN 2005 Symposium on Principles and Practice of Parallel Programming (PPoPP)*. ACM Press, June 2005.

VITA

Ömer Subaşi was born in Kars, in 1987. He received his B.Sc. degree in Computer Engineering with double major in Mathematics from Koç University in 2009. From September 2009 to May 2012, he worked as a teaching and research assistant at Research Center for Multi-core Software Engineering at Koç University, Turkey. During Spring 2010, he left his graduate study due to health reasons. His research was supported by TÜBİTAK. He has published papers about statically verifying concurrent software for TACAS'10 conference and workshops WoDET'12 and CSW'10. He plans to pursue a PhD study at Barcelona Super Computing Center having an Intel PhD scholarship.