Modular Verification of Concurrent Programs via Sequential Model Checking

Dan Rasin
Modular Verification of Concurrent Programs via Sequential Model Checking

Research Thesis

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Dan Rasin

Submitted to the Senate of the Technion — Israel Institute of Technology
Kislev 5778  Haifa  November 2017
This research was carried out under the supervision of Prof. Orna Grumberg and Dr. Sharon Shoham, in the Faculty of Computer Science.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my supervisor, Prof. Orna Grumberg, for being both a wonderful supervisor and an incredible person. Thank you for the weekly inspiring meetings, for constantly helping me improve my work, and for encouraging me to explore and learn what research is. Thank you for always having an open door, and for making the studying experience so fun. It has been a huge honor to work with you and learn from you.

I would like to thank my co-supervisor Dr. Sharon Shoham for her immense part of this work. Thank you for your dedication, and for the ability to make every meeting fruitful with new ideas and sharp thought. It was always enjoyable to meet you both in Tel Aviv and the Technion. You are a great advisor and a great person, and I am very grateful for all your help and guidance.

I would like to thank my parents, Lina and Lev, for their endless love and support. Thank you for teaching me how to always aim high, and being there for me at every stage of my life. Finally, I would like to thank my girlfriend Veronica. You have been escorting me through this period both on good days when I had progress and on worse more frustrating days. You have been always supportive and helped me to go forward, and I will be forever grateful for that.

The generous financial help of the Technion is gratefully acknowledged.
# Contents

List of Figures

Abstract

1 Introduction
   1.1 Related Work ........................................ 5
   1.2 Organization ........................................... 6

2 Preliminaries ............................................. 9

3 Reduction to Sequential Verification .................... 13
   3.1 From Concurrent to Sequential Programs ............. 13
   3.2 Interface Between Main and the Environment ......... 14
   3.3 What’s Next ............................................. 14

4 Analyzing the Main Thread ................................ 17
   4.1 The Structure of $P_M$ ................................ 17
   4.2 Representation of $P$ Within $P_M$ ................... 20
   4.3 Initial Construction of $P_M$ ......................... 25
   4.4 Iteration of the MainThreadCheck Algorithm ........ 26
   4.5 Analyzing a Potentially Spurious Violating Path .... 28
   4.6 Generalizing an Environment Query .................. 31

5 Soundness and Progress of the Main Thread Analysis .... 33
   5.1 Soundness .............................................. 33
   5.2 Progress and Termination ............................. 43

6 Answering Environment Queries ............................. 49
   6.1 Sequential Program for Answering Environment Queries 49
   6.2 Correctness of Environment Query .................... 51

7 Extending the Algorithm .................................... 59
   7.1 Extending to Multiple Threads ....................... 59
   7.2 Environment Queries with Multiple Threads ........ 61
7.3 Extending Assertions to all Threads of $P$ .......................... 67

8 Optimizations ........................................ 69
  8.1 General Optimization ................................... 69
  8.2 Optimizations for Generalizing Environment Information ............ 70
  8.3 Multiple Threads Optimizations .................................. 72

9 Experimental Results .................................... 75

10 Conclusion and Future Work ................................ 79
  10.1 Conclusion ................................................. 79
  10.2 Future Work .................................................. 79
## List of Figures

4.1 A typical computation of $P_M$ is a periodic repetition of extended steps consisting of: original commands of $t_M$, newly added assertion and the $env\_move$ function. 20  
4.2 Peterson’s mutual exclusion algorithm for two threads $t0$ and $t1$. 27  
4.3 The sequential program $P_0$ after adding environment move calls. 27  
4.4 (a) If $Reach_{(t_E)}(\alpha, \beta) = FALSE$, we search for more general $\alpha'$ and $\beta'$ which restrict the environment transition; (b) If $Reach_{(t_E)}(\alpha, \beta) = \psi \neq FALSE$, then we know that $\psi$ leads to $\beta$ and that $\psi \land \alpha \neq FALSE$. 31  
4.5 The sequential program $P_0$ after a few iterations of Algorithm 4.1. 31  
4.6 The $env\_move$ function of $P_0$: initially (without highlighted lines); and after one refinement (with highlighted lines). 31  
5.1 The partition and partitioning states of a computation $\rho$. Note that $r_3$ is allowed to have inner cut-point states, as it is a computation of $t_E$. 39  
6.1 The sequential program $P_1$ for computing $Reach_{(t_E)}(\alpha, \beta)$ for $\alpha = \neg cs1 \land claim0 \land (\neg claim1 \lor turn = 0)$ and $\beta = cs1$. For the convenience of presentation, we split the assume$(\alpha)$ to three assume commands. 52  
7.1 A variation of the fib_bench examples from the SV-Comp concurrency benchmark with three threads. 67  
7.2 The sequential program $P_2$ with both try_start and env_move’ calls, for answering $Reach_{\{t_2,t_3\}}(\alpha_1, \beta_1)$ with $\alpha_1 \triangleq (a == 1 \land b == 1 \land c == 1)$ and $\beta_1 \triangleq (a + 2b > 15)$. 67  
9.1 Run times [secs] for all four tools for verifying concurrent dynamic programming algorithms. 77  
9.2 Run times [secs] for fib_bench programs with ring topology vs. chain topology. 78
Abstract

Verification of concurrent programs is known to be extremely difficult. On top of the challenges inherent in verifying sequential programs, it adds the need to consider a high (typically unbounded) number of thread interleavings. In this work, we utilize the plethora of work on verification of sequential programs for the purpose of verifying concurrent programs. We introduce a technique which reduces the verification of a concurrent program to a series of verification tasks of sequential programs, without explicitly encoding all the possible interleavings. Our approach is modular in the sense that each sequential verification task roughly corresponds to the verification of a single thread, with some additional information about the environment in which it operates. Information regarding the environment is gathered during the run of the algorithm, by need.

A unique aspect of our approach is that it exploits a hierarchical structure of the program in which one of the threads, considered “main”, is being verified as a sequential program. Its verification process initiates queries to its “environment” (which may contain multiple threads). Those queries are answered by sequential verification, if the environment consists of a single thread, or, otherwise, by applying the same hierarchical algorithm on the environment.

Our technique is fully automatic, and allows us to use any off-the-shelf sequential model-checker. We implemented our technique in a tool called CoMuS and evaluated it against established tools for concurrent verification. Our experiments show that it works particularly well on hierarchically structured programs.
Chapter 1

Introduction

Verification of concurrent programs is known to be extremely hard. On top of the challenges inherent to verifying sequential programs, it adds the need to consider a high (typically unbounded) number of thread interleavings. For such programs, it is very appealing to exploit their modular structure in verification.

Usually, however, a property of the whole system cannot be partitioned into a set of properties that are local to the individual threads. Hence, pure modular verification methods, in which each thread is verified in isolation are not useful in practice. Thus, proving the property on a single thread requires some knowledge about its interaction with its environment.

In this work we develop a new approach, which utilizes the plethora of work on verification of sequential programs for the purpose of modularly verifying the safety of concurrent programs. Our technique automatically reduces the verification of a concurrent program to a series of verification tasks of sequential programs. This allows us to benefit from any past, as well as future, progress in techniques for sequential verification.

Our approach is modular in the sense that each sequential verification task roughly corresponds to the verification of a single thread, with some additional information about the environment in which it operates. This information is automatically and lazily discovered during the run of the algorithm, when needed.

A unique aspect of our approach is that it exploits a hierarchical view of the program in which one of the threads, $t_M$, is considered “main”, and all other threads are considered its “environment”. We analyze $t_M$ using sequential verification, where, for soundness, all interferences from the environment are abstracted (over-approximated) by a function $\text{env\_move}$, which is called by $t_M$ whenever a context switch should be considered. Initially, $\text{env\_move}$ havoc all shared variables; it is gradually refined during the run of the algorithm.

When the sequential model-checker discovers a violation of safety in $t_M$, it also returns a path leading to the violation. The path may include calls to $\text{env\_move}$, in which case the violation may be spurious (due to the over-approximation). Therefore, the algorithm initiates queries to the environment of $t_M$ whose goal is to check whether certain interferences, as observed on the violating path, are feasible in the environment. Whenever an interference turns out to be infeasible, the $\text{env\_move}$ function is refined to exclude it. Eventually, $\text{env\_move}$
becomes precise enough to enable full verification of the desired property on the augmented $t_M$. Alternatively, it can reveal a real (non-spurious) counterexample in $t_M$.

The queries are checked on the environment (that may consist of multiple threads) in the same modular manner. Thus we obtain a hierarchical modular verification. The hierarchical characteristics of our method guarantee that along the algorithm, each thread learns about the next threads in the hierarchy, and is provided with assumptions from former threads in the hierarchy to guide its learning.

Our technique is fully automatic and performs unbounded verification, i.e., it can both prove safety and find bugs in a concurrent program. It works on the level of program code. The information gathered about the environment is accumulated in the code of $t_M$ by means of assertions, and assumptions within the env.move function.

The fact that our algorithm generates standard sequential programs in its intermediate checks allows us to use any off-the-shelf sequential model-checker. In particular, we can handle concurrent programs with an infinite state-space, provided that the sequential model checker supports such programs.

We implemented our technique in a prototype called Concurrent to Multiple Sequential (CoMuS) and evaluated it against established tools for unbounded verification of concurrent C programs. We use SeaHorn [22] to model check sequential programs. SeaHorn receives C programs, annotated with assertions, and checks whether an assertion can be violated. If so, it produces a trace leading to a violated assertion. Otherwise, it announces that no violation occurs.

While our approach is designed to work on any concurrent program, our experiments show that it works particularly well on programs in which the threads are arranged as a chain, $t_1, t_2, \ldots, t_k$, where thread $t_i$ depends only on its immediate successor $t_{i+1}$ in the chain. This induces a natural hierarchical structure in which $t_1$ is the main thread with environment $t_2, \ldots, t_k$; thread $t_2$ is the main thread in the environment, and so on. This structure often occurs in concurrent implementations of dynamic programming [2] algorithms.

To summarize, the main contributions of our work are as follows:

- We present a new modular verification approach that reduces the verification of a concurrent program to a series of verification tasks of sequential programs. Any off-the-shelf model checker for sequential programs can thus be used.
- Our approach exploits a hierarchical view, where each thread learns about the next threads in the hierarchy, and is provided with assumptions from former threads to guide its learning.
- The needed information on a thread’s environment is gathered in the code, automatically and lazily, during the run of the algorithm.
- We implemented our approach and showed that as the number of threads grows, it outperforms existing tools on programs that have a hierarchical structure, such as concurrent
implementations of dynamic programming algorithms.

1.1 Related Work

The idea of performing code transformation and using any off-the-shelf model checker appeared in [42, 43, 30]. However, they translate the concurrent program to a single nondeterministic sequential program, and model the scheduler as well. In contrast, our technique exploits the modular structure of the program. [38] also transforms the concurrent program into a single sequential one. However, their approach is not sound, as they can miss some behaviors leading to an error.

The fundamental concept of using modular reasoning is known as the Assume-Guarantee paradigm [34]. The main idea is to split the specification into two: one part describing the desired behavior of one module, and the second part describes an assumption on the module’s environment, under which the first property should hold. The system can then be proved safe if the module satisfies the desired property under the given assumption about the environment, and the environment is proved to satisfy the assumption (independently). The assume guarantee paradigm has been a key aspect in many works in the world of verification and model checking [23], [31], [32], [12]. Our work is inspired from the Assume-Guarantee paradigm. Assumptions about the environment under which we try to prove the desired property are gathered within our env.move function. We start by generating simple assumptions about the environment (which are correct by construction), and strengthen them with new assumptions after we were able to prove that the environment indeed satisfies them.

In the rest of this section, we address unbounded modular techniques for proving safety properties of concurrent programs. Other related problems in the field of concurrent verification include proving termination (e.g. [9, 35]) and using bounded model checking, where the bound can address different parameters, such as the number of context switches [25, 43], the number of write operations [42] or the number of loop iterations [39, 1, 45].

The work most closely related to ours is [20, 21]. There, an automatic modular verification framework is described, which uses predicate abstraction of both states and environment transitions. Our env.move function is also used to abstract environment transitions, and we also use predicates, in the form of assertion in the code, to reason about states leading to an error. [20, 21] also iteratively try to prove the program safe, and gradually refine this abstraction by checking possible witnesses of errors. However, they treat all threads symmetrically, whereas our approach exploits a hierarchical view of the program. In particular, we single out a ”main” thread from its ”environment” threads. The main thread initiates queries to its environment in order to prove or disprove a violation. The environment threads are also treated hierarchically, in a similar manner. In addition, their exploration of a single thread (with the information learned from its environment) uses abstract reachability trees, which are inherent to their technique. We, on the other hand, represent each thread (augmented with information from its environment) as a stand-alone C program. Thus, we can use any off-the-shelf model checker to address the ”sequential part” of the verification problem, utilizing possible advancements in the field of
sequential model checking.

The works in [13, 26, 16] suggest to apply rely-guarantee reasoning for concurrent (or asynchronous) programs, while the different sections of the program can be verified sequentially. However, their technique requires human effort to specify the rely-guarantee conditions. [24] also suggests a human-involving approach, where verification is modular, but the user is responsible for providing the abstraction for each thread. In our approach we infer assumptions about the environment automatically, and augment the code with assumptions and assertions to represent them.

[14] suggests a modular algorithm with rely-guarantee reasoning and automatic condition inference. [27] formalizes the algorithm in the framework of abstract interpretation. However, their algorithm requires finite state systems, and its inferred conditions only refer to changes in global variables. Hence, they fail to prove properties where local variables are necessary for the proof. This can be overcome by treating all variables as global, but this would strip the algorithm from its modularity. In our approach, reasoning about local variables is allowed, when we learn that they are necessary for verification. Such variables are then turned into global variables, but their behavior is abstracted in the threads that require them, preserving modularity. [8] also fights the incompetence of modular proofs by exposing local variables as global, according to counterexamples. However, their approach uses BDDs and suits finite state systems. Similar to [20], they also treat threads symmetrically. Our approach is applicable to infinite state systems and uses a guided search to derive cross-thread information.

Our queries resemble queries in learning-based compositional verification [7, 32], which are also answered by a model checker. Our hierarchical recursive approach resembles the n-way decomposition handled in [32]. However, these works represent programs, assumptions and specification as LTSs, and although extended to deal with shared memory in [41] these algorithms are suitable for finite state systems.

Several works such as [15, 18, 44, 37], tackle the interleaving explosion problem by performing a thread interleaving reduction. [44] combines partial order reduction [17] with the impact algorithm [28], whereas [37] identifies reducible blocks for compositional verification. These approaches are complementary to ours, as our first step is performing an interleaving reduction (to identify cut-points for env.move calls).

### 1.2 Organization

The rest of this work is organized as follows. Chapter 2 presents the background and necessary definitions concerning the semantics and analysis of both sequential and concurrent programs. Chapter 3 presents the key concepts of our methodology, and formally defines environment queries. Chapter 4 focuses on the sequential program constructed to analyze $t_M$, and the $env.move$ function that is used to over-approximate computations of the environment. Chapter 5 formally proves the correctness of the aforementioned analysis. The correctness of the proofs relies on the existence of a method for answering environment queries. Chapter 6 describes how such environment queries are answered in case the environment consists of a single thread. The
chapter also includes a proof for the correctness of this construction. Chapter 7 describes how our method can be extended to multiple threads. Chapter 8 describes a list of optimizations used by our tool. Chapter 9 provides an experimental evaluation, and shows that our technique is useful for programs which have a hierarchical structure. Chapter 10 summarizes this work.
Chapter 2

Preliminaries

Sequential Programs. A sequential program \( P \) is defined by a control flow graph whose nodes are a set of program locations \( L \) (also called labels), and whose edges \( E \) are a subset of \( L \times L \). The program has an initial label, denoted \( l_{\text{init}} \in L \). Each node \( l \) is associated with a command \( c \in \text{cmds} \), denoted \( \text{cmd}(l) \), which can be an assignment or an if command, as well as \text{havoc}, \text{assume} and \text{assert} (explained below). Intuitively, we think of standard C programs (that may contain loops as well), which can be trivially compiled to such control flow graphs. The program may also include non-recursive functions, which will be handled by inlining.

The program is defined over a set of variables \( V \). Conditions in the program are quantifier-free first-order-logic formulas over \( V \). A special variable \( pc \notin V \), ranging over \( L \), indicates the program location. A state \( s \) of the sequential program \( P \), is a pair \((l,\sigma)\) where \( l \in L \) is the value of \( pc \) and \( \sigma \) is a valuation of \( V \). Variables may have unbounded domains, resulting in a potentially infinite state-space. We also assume the existence of a special error state, denoted \( \epsilon = (l_\epsilon, \bot) \). We denote by \( l(s) \) and \( \sigma(s) \) the first and second components (resp.) of a state \( s = (l, \sigma) \). Given a valuation \( \sigma \) over \( V \) and a set of variables \( U \), we denote by \( \sigma|_U \) the restriction of \( \sigma \) to the variables in \( U \). That is, \( \sigma|_U(v) = \sigma(v) \) for every \( v \in V \cap U \), and \( \sigma|_U(v) \) is undefined for any other \( v \). Note that we do not assume that \( U \subseteq V \). If \( c \) is a command or a condition over some of the program variables, we denote by \( \text{Vars}(c) \) the set of variables appearing in \( c \). We denote \( \sigma|_c = \sigma|_{\text{Vars}(c)} \).

A program may include an initialization condition in the form of a valuation \( \sigma_{\text{init}} \) of a set of variables \( U \subseteq V \). We denote by \( \sigma_{\text{init}}(v) \) the initial value of a variable \( v \in U \). The set of initial states consists of all states \((l_{\text{init}}, \sigma)\) where \( \sigma|_U = \sigma_{\text{init}} \). We denote by \( \phi_{\text{init}} \) the initialization formula \( \phi_{\text{init}} \triangleq \bigwedge_{v \in U} [v = \sigma_{\text{init}}(v)] \). It is possible to express more complex initial conditions, e.g. \((v_1 > 0 \land v_2 + v_3 = 7)\), via explicit assume commands in the code, as described below.

For a state \( s = (l, \sigma) \), let \( \text{cmd}(s) = \text{cmd}(l) \). We denote \( \text{next}(s) = \{s' \mid s' \text{ can be obtained from } s \text{ using } \text{cmd}(s)\} \). This set is defined according to the command \( \text{cmd}(s) \). In particular, \( s' \in \text{next}(s) \) implies that \((l(s), l(s'))\) is an edge in \( P \). The definition of \( \text{next}(s) \) for assignments and if commands is standard. A \text{v=havoc}() command assigns a non-deterministic value to the variable \( v \). An assume \((b)\) command is used to disregard any computation in which the command

\[\text{havoc} \text{ commands are non-deterministic, hence } s' \text{ is not unique.}\]
b does not hold. It does nothing, otherwise. Formally, if \( s = (l, \sigma) \) and \( cmd(s) = \text{assume} (b) \), then \( \sigma \vdash b \Rightarrow next(s) = \{(l', \sigma)\} \) where \( l' \) is the label of the next command to be executed after the \text{assume} command (i.e., \( l' \) is the single label such that \( (l, l') \in E \)), and \( \sigma \not\vdash b \Rightarrow next(s) = \emptyset \). An \text{assert} (b) command moves to the error state if \( b \) is violated, and does nothing otherwise. Formally, if \( s = (l, \sigma) \) and \( cmd(s) = \text{assert} (b) \) for some condition \( b \), then \( \sigma \vdash b \Rightarrow next(s) = \{(l', \sigma)\} \) where \( l' \) is the label of the next command to be executed, and \( \sigma \not\vdash b \Rightarrow next(s) = \{\epsilon\} \).

A computation \( \rho \) of \( P \) is a sequence \( \rho = s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_n \) for some \( n \geq 0 \) s.t. for every two adjacent states \( s_i, s_{i+1} : s_{i+1} \in next(s_i) \). \( \rho \) is an initial computation in \( P \) if it starts from an initial state. \( \rho \) is a reachable computation in \( P \) if there exists an initial computation \( \rho' \) for which \( \rho \) is the suffix. The path of a computation \((l_0, \sigma_0) \rightarrow \ldots \rightarrow (l_n, \sigma_n)\) is the sequence of program locations \( l_0, \ldots, l_n \).

**Remark.** The examples appearing later in this work are C programs, where we use line numbers to denote the labels of the program.

**Preconditions and Postconditions.** Given a condition \( q \) over the program variables \( V \) and an edge \( e = (l, l') \), a precondition of \( q \) w.r.t. \( e \), denoted \( \text{pre}(e, q) \), is a condition \( p \) such that for every state \( s \), if \( \sigma(s) \vdash p \) and \( l(s) = l \) then there exists \( s' \in next(s) \) s.t. \( \sigma(s') \vdash q \) and \( l(s') = l' \). A precondition extends to a path \( \pi = l_0, \ldots, l_n \) with commands \( c_0, \ldots, c_{n-1} \) in the natural way. The weakest precondition of \( q \) w.r.t. \( e \) (resp., \( \pi \)) is a precondition that is implied by any other precondition, and can be computed in a standard way, according to the command \( cmd(l) \) (resp., \( c_0, \ldots, c_{n-1} \)) and the chosen target label \( l' \) (resp., \( l_1, \ldots, l_n \)) [11]. We denote it \( \text{wp}(e, q) \) (resp., \( \text{wp}(\pi, q) \)).

A path precondition \( p = \text{pre}(\pi, q) \) has the property that for every state \( s \) such that \( \sigma(s) \vdash p \), there exists a computation \( \rho \) whose path is \( \pi \) from \( s \) to some state \( s' \) such that \( \sigma(s') \vdash q \). A path weakest precondition \( p = \text{wp}(\pi, q) \) also satisfies the converse: if \( \rho \) is a computation whose path is \( \pi \) from some state \( s \) to a state \( s' \) s.t. \( \sigma(s') \vdash q \), then \( \sigma(s) \vdash p \).

A postcondition of a condition \( p \) w.r.t \( e = (l, l') \), denoted \( \text{post}(e, p) \), is any condition \( q \) such that if \( \sigma(s) \vdash p \), \( l(s) = l \) then for every \( s' \in next(s) \), if \( l(s') = l' \) then \( \sigma(s') \vdash q \). Postconditions can also be extended to paths \( \pi = l_0, \ldots, l_n \). We use \( \text{post}(\pi, p) \) to denote a postcondition of condition \( p \) w.r.t. path \( \pi \). A path postcondition \( q = \text{post}(\pi, p) \) has the property that for every computation \( \rho \) whose path is \( \pi \) from a state \( s \) s.t. \( \sigma(s) \vdash p \) to some state \( s' \), it holds that \( \sigma(s') \vdash q \).

**Concurrent Programs** A concurrent program \( P \) consists of multiple threads \( t_1, \ldots, t_m \), where each thread \( t_i \) has the same syntax as a sequential program over a set of variables \( V_i \) and a program location variable \( pc_i \). The threads communicate through shared variables, meaning that generally \( V_i, V_j \) are not disjoint for \( i \neq j \). Let \( V = \bigcup_{i=1}^{m} V_i \). A state \( s \) of the concurrent program \( P \) is a pair \( s = (\bar{l}, \sigma) \), where \( \sigma \) is a valuation of \( V \) and \( \bar{l} = (l_1, \ldots, l_m) \) with \( l_i \) being...
We define initial states as the set of all states \((\bar{t}^{\text{init}}, \sigma)\) where \(\sigma \models \phi_{\text{init}}\) and \(\bar{t}^{\text{init}} = (t_1^{\text{init}}, \ldots, t_m^{\text{init}})\) are the initial labels of the threads \(t_1, \ldots, t_m\) (resp.).

The execution of a concurrent program is interleaving, meaning that exactly one thread performs a command at each step, and the next thread to execute is chosen nondeterministically. We consider a sequentially consistent semantics in which the effect of a single command on the memory is immediate. For \(s = (\bar{t}, \sigma)\), let \(\text{cmd}(s, t_i)\) denote the command of thread \(t_i\) at label \(l_i\). We define \(\text{next}(s, t_i) = \{s' \mid s' \text{ can be obtained from } s \text{ after } t_i \text{ performs } \text{cmd}(s, t_i)\}\). A computation \(\rho\) of the concurrent program \(P\) is a sequence \(s_0 \xrightarrow{t_1^{(1)}} s_1 \xrightarrow{t_2^{(2)}} \ldots \xrightarrow{t_n^{(n)}} s_n\) s.t. for every two adjacent states \(s_i, s_{i+1}: t_{(i+1)}^{(i+1)} \in \{t_1, \ldots, t_m\}\) and \(s_{i+1} \in \text{next}(s_i, t_{(i+1)})\). We say that \(\rho\) is a computation of thread \(t\) in \(P\) if \(t_j^{(j)} = t\) for every \(1 \leq j \leq n\). Given a set of threads \(T \subseteq \{t_1, \ldots, t_m\}\), we say that \(\rho\) is a computation of \(T\) in \(P\) if \(t_j^{(j)} \in T\) for every \(1 \leq j \leq n\). We define initial and reachable computations as in the sequential case, but w.r.t. computations of the concurrent program.

**Remark.** Our technique also supports atomic synchronization operations by modeling them with atomic control commands. For example, \texttt{Lock(lock)} is modeled by atomic execution of \texttt{assume(lock = false); lock=true;} As explained later, our technique models context switches by explicit calls to a function, env\_move. Thus, we are able to guarantee that these commands are treated as an atomic operation (with no context switches allowed) by not including env\_move calls between them.

**Variable Classification** A variable is read by a thread \(t_i\) if it appears in a condition of any control structure (if, assume, assert) or on the right hand side of any assignment in \(t_i\). A variable is written by \(t_i\) if it appears on the left hand side of any assignment in \(t_i\). A variable \(v \in V\) is shared between two threads \(t_i, t_j\) if \(v \in V_i \cap V_j\). A variable \(v \in V_i\) is a local variable of \(t_i\) if \(v \not\in V_j\) for every \(j \neq i\).

**Safety.** A computation of a (sequential or concurrent) program is violating if it ends in the error state (i.e., the last step of the computation executes the command \texttt{assert(b)} at a state \(s\) such that \(\sigma(s) \not\models b\)). The computation is safe otherwise. A (sequential or concurrent) program is safe if it has no initial violating computations. In the case of a sequential program, we refer to the path of a violating computation as a violating path.

A **Sequential Model Checker** is a tool which receives a sequential program as input, and checks whether the program is safe. If it is, it returns “SAFE”. Otherwise, it returns a counterexample in the form of a violating path.

**Interleaving Reduction** An interleaving reduction analysis is a technique which identifies a set of labels, called cut-points, such that the original program is safe if and only if all the computations in which context-switches can only occur at cut-points are safe. This means that
at every state of the computation, at most one thread (which is the thread performing the current sequence of steps) can be at a label which is not a cut-point. When the next thread to move changes (i.e., a context switch occurs), all threads must be in a cut-point label.

More formally, let $CL_1 \subseteq L_1, \ldots, CL_m \subseteq L_m$ be the sets of cut-point labels for the threads $t_1, \ldots, t_m$ of a concurrent program $P$. A state $s$ satisfying $l(s, t_j) \in CL_j$ for every thread $t_j$ of $P$, is called a cut-point state. The program $P$ is safe if and only if it has no initial violating computation $\rho = s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} \ldots \xrightarrow{t_n} s_n$ in which every two adjacent transitions $s_i \xrightarrow{t_i} s_{i+1}$ satisfy the following two properties:

- If $t_i \neq t_{i+1}$, then $s_i$ is a cut-point state.
- If $t_i = t_{i+1}$, then for every thread $t_j \neq t_i$: $l(s_i, t_j) \in CL_j$ (and $l(s_i, t_i)$ is not restricted).

For the rest of this work, we will only consider computations in which context switches are restricted to cut-point states. For simplicity, we assume that all initial states are also cut-point states, i.e., $l_{init} \in CL_j$ for every thread $t_j$ of $P$. Several techniques for performing an interleaving reduction are described in [15, 18, 44, 37].
Chapter 3

Reduction to Sequential Verification

In this chapter we provide an overview of our methodology for verifying safety properties of concurrent programs, given via assertions. The main idea is to use a sequential model checker in order to verify the concurrent program. Our approach handles any number of threads. However, to simplify the presentation, we first describe our approach for a concurrent program that consists of two threads. In Chapter 7, we extend the presentation to any number of threads.

In the sequel, we fix a concurrent program $P$ with two threads. We refer to one as the main thread ($t_M$) and to the other as the environment thread ($t_E$), with variables $V_M$ and $V_E$ and program location variables $pc_M$ and $pc_E$, respectively. $V_M$ and $V_E$ might intersect. Let $V = V_M \cup V_E$. Given a state $s = (l, \sigma)$, we denote by $l_M(s)$ and $l_E(s)$ the values of $pc_M$ and $pc_E$ in a state $s$, respectively. We also denote by $l_{init}^M$ and $l_{init}^E$ the initial values of $pc_M$ and $pc_E$. For simplicity, we assume that the safety of $P$ is specified by assertions in $t_M$. Section 7.3 describes how to support assertions in all threads of $P$.

3.1 From Concurrent to Sequential Programs

Our algorithm generates and maintains a sequential program for each thread. Let $P_M$ and $P_E$ be the two sequential programs, with variables $\hat{V}_M \supseteq V_M$ and $\hat{V}_E \supseteq V_E$. Each sequential program might include variables of the other thread as well, together with additional auxiliary variables not in $V$. Our approach is asymmetric, meaning that $P_M$ and $P_E$ have different roles in the algorithm. $P_M$ is based on the code of $t_M$, and uses a designated function, $\text{env.move}$, to abstract computations of $t_E$. $P_E$ is based on the code of $t_E$, and is constructed in order to answer specific queries for information required by $P_M$, specified via assumptions and assertions. The algorithm iteratively applies model checking to each of these programs separately. In each iteration, the code of $P_M$ is gradually modified, as the algorithm learns new information about the environment, and the code of $P_E$ is adapted to answer the query of interest.
### 3.2 Interface Between Main and the Environment

In Chapter 4, we first describe the way our algorithm operates on $P_M$. During the analysis of $P_M$, information about the environment is retrieved using environment queries: Intuitively, an environment query receives two conditions, $\alpha$ and $\beta$, and checks whether there exists a reachable computation of $t_E$ in $P$ from $\alpha$ to $\beta$. The idea is to perform specific guided queries in $t_E$, to search for computations that might “help” $t_M$ to reach a violation. If such a computation exists, the environment query returns a formula $\psi$, which ensures that all states satisfying it can reach $\beta$ using $t_E$ only. We also require that $\alpha$ and $\psi$ overlap. In order to ensure the reachability of $\beta$, the formula $\psi$ might need to address local variables of $t_E$, as well as $pc_E$. These variables will then be added to $P_M$, and may be used for the input of future environment queries. If our algorithm can prove that there are no such computations of $t_E$, it returns $\psi = \text{FALSE}$. The formal definition follows.

**Definition 3.2.1** (Environment Query). An environment query $\text{Reach}(t_E)(\alpha, \beta)$ receives conditions $\alpha$ and $\beta$ over $V \cup \{pc_E\}$, and returns a formula $\psi$ over $V \cup \{pc_E\}$ such that:

1. If there exists a computation of $t_E$ in $P$ that is (1) reachable in $P$, (2) starts from a cut point state $s$ s.t. $s \models \alpha$, and (3) ends in a cut point state $s'$ s.t. $s' \models \beta$, then $\psi \land \alpha \neq \text{FALSE}$.
2. For every state $s$ s.t. $s \models \psi$, there exists a computation (not necessarily reachable) of $t_E$ in $P$ from $s$ to some $s'$ s.t. $s' \models \beta$.
3. $\psi \neq \text{FALSE} \Rightarrow \psi \land \alpha \neq \text{FALSE}$.

**Observation 3.2.2.**

1. We note that $\psi$ is not required to be precise, i.e., nothing is required of states $s$ s.t. $s \not\models \psi$.
2. If $\psi = \text{FALSE}$, the first property implies that there is no reachable computation of $t_E$ in $P$ between cut point states from $\alpha$ to $\beta$.
3. If $\alpha \land \beta \neq \text{FALSE}$, $\beta$ is always a valid result for $\text{Reach}(t_E)(\alpha, \beta)$, as a computation of length zero would satisfy the requirements.
4. The aforementioned computations do not have to be initial, (i.e., the first state $s$ of the computation is not necessarily an initial state).
5. The last property of Definition 3.2.1 is only needed for the progress of our main algorithm (see Lemma 5.2.4 of Section 5.2). For soundness, the first two properties suffice.

### 3.3 What’s Next

Chapter 4 focuses on $P_M$. It describes the structure of $P_M$, based mainly on the code of $t_M$, and the connection between states and computations of $P$ to states and computations of $P_M$. It also describes the $\text{env\_move}$ function, and formally defines in what sense it over-approximates
$t_E$. The key part of this chapter is Algorithm 4.1, which analyzes and refines $P_M$ to determine the safety of $P$. The correctness proof for Algorithm 4.1 appears in Chapter 5. The algorithm assumes the existence of a model checker that can determine the correctness of sequential programs (and provide counterexamples). That is, it reduces the verification of concurrent programs to a series of verification tasks of sequential programs.

An additional assumption of Algorithm 4.1 and its correctness proof in Chapter 5, is the existence of a method for answering environment queries. Chapter 6 describes how environment queries are answered in the case of a single environment thread $t_E$. The key elements of this chapter are $P_E$, the sequential program constructed according to $t_E$, and the try_start function, that is used in $P_E$ to over-approximate initial computations of $P$ in order to let $P_E$ simulate non-initial computations of $t_E$ that follow them. Chapter 6 also includes a proof for the correctness of this construction.

**Multiple Threads** The key ingredients used by our technique are (i) an env_move function that is used in $P_M$ to overapproximate finite computations (of any length) of $t_E$, and (ii) a try_start function as mentioned above. When $P$ has more than two threads, the environment of $t_M$ consists of multiple threads, hence environment queries are evaluated by a recursive application of the same approach. Since the computations we consider in the environment are not necessarily initial, the main thread of the environment should now include both the env_move function and the try_start function. For more details see Chapter 7.
Chapter 4

Analyzing the Main Thread

In this chapter we describe our algorithm for analyzing the main thread for the purpose of proving the concurrent program $P$ safe or unsafe (Algorithm 4.1). Algorithm 4.1 maintains a sequential program, $P_M$, over $\hat{V}_M \supseteq V_M$, which represents the composition of $t_M$ with an abstraction of $t_E$. The algorithm changes the code of $P_M$ iteratively, by adding new assumptions and assertions, as it learns new information about the environment.

The rest of this chapter is organized as follows: Section 4.1 describes general properties of $P_M$, that hold throughout Algorithm 4.1. Section 4.2 provides necessary formal definitions describing the relation between $P$ and $P_M$. Next, we start describing our algorithm. Section 4.3 describes the initialization steps of Algorithm 4.1, and presents a running example. After initialization, Algorithm 4.1 works in iterations. Section 4.4 describes the outline of each iteration, and explains when the algorithm can terminate with a final result (safe or unsafe). Section 4.5 details how the algorithm analyzes counterexamples provided by the sequential model-checker. This analysis is the core of every iteration of Algorithm 4.1. Section 4.6 briefly describes how the results of environment queries can be generalized.

4.1 The Structure of $P_M$

We now detail the general structure of $P_M$. The described properties are invariants, and hold throughout all modifications of $P_M$.

Variables of $P_M$ As mentioned before, $\hat{V}_M$ includes all variables of $t_M$, i.e., $\hat{V}_M \supseteq V_M$. $\hat{V}_M$ may also include variables in $V_E \setminus V_M$, as well as include $pc_E$ as an explicit variable. All other variables in $\hat{V}_M$ are auxiliary local variables required for ad-hoc technical purposes, and will be mentioned later (see (4) of Section 4.5). Other than the usage in (4) of Section 4.5, all commands in $P_M$ only address variables in $V \cup \{pc_E\}$. Thus, we can safely assume that conditions computed later in Algorithm 4.1 using paths of $P_M$ (weakest precondition and postcondition) are also over $V \cup \{pc_E\}$. 

17
Algorithm 4.1 Algorithm MainThreadCheck

1: \textbf{procedure} MainThreadCheck($t_M, t_E$) 
2: \hspace{1em} $P_M = \text{add } env\_move \text{ calls in } t_M \text{ and initialize } env\_move()$
3: \hspace{1em} \textbf{while} a violating path exists in $P_M$ do ~// using sequential MC 
4: \hspace{2em} Let $\hat{\pi} = l_0, \ldots, l_{n+1}$ be a violating path.
5: \hspace{2em} \textbf{if} there are no $env\_move$s in $\hat{\pi}$ \textbf{then}: 
6: \hspace{3em} \text{return} “Real Violation”
7: \hspace{2em} \textbf{end if}
8: \hspace{2em} Let $\hat{\pi} = l_{k+1}, \ldots, l_n$
9: \hspace{2em} $\beta = wp(\hat{\pi}_{\text{end}}, -b)$ \hspace{1em} // see (1) in Section 4.5
10: \hspace{2em} $\alpha = \text{post}(\hat{\pi}_{\text{start}}, \phi_{\text{init}})$ \hspace{1em} // see (2) in Section 4.5
11: \hspace{2em} Let $\psi = \text{Reach}(t_E)(\alpha, \beta)$ \hspace{1em} // environment query for $t_E$ (see Chapter 6)
12: \hspace{2em} \textbf{if} $\psi$ is $\text{FALSE}$ \textbf{then}
13: \hspace{3em} Let $\alpha', \beta'$ be as in (4) in Section 4.5.
14: \hspace{3em} $P_M = \text{RefineEnvMove}(P_M, \alpha', \beta')$
15: \hspace{2em} \textbf{else} \hspace{1em} // see (5) in Section 4.5 
16: \hspace{3em} Add assert($\neg\psi$) in $P_M$ at new label $\hat{l}'$, placed right before $\hat{l}_k$
17: \hspace{2em} \textbf{end if}
18: \hspace{1em} \textbf{end while}
19: \hspace{1em} \text{return} “Program is Safe”. 
20: \hspace{1em} \textbf{end procedure}

Initialization Condition of $P_M$ All variables in $\hat{V}_M \cap V$ appear in $P_M$ with the same declaration as in $P$, and the same (possible) initialization as in $P$. In addition, if $p \in \hat{V}_E$, it is initialized with $l_v^{\text{init}}$. Formally, if $P$ uses the initialization formula $\phi_{\text{init}} = \bigwedge_{v \in U}[v = \sigma_{\text{init}}(v)]$, then $\hat{\phi}_{\text{init}}$, the initialization formula of $P_M$, is $\hat{\phi}_{\text{init}} = \bigwedge_{v \in U \cap \hat{V}_M}[v = \sigma_{\text{init}}(v)]$ if $p \in \hat{V}_E$, and $\hat{\phi}_{\text{init}} = \bigwedge_{v \in U \cap \hat{V}_M}[v = \sigma_{\text{init}}(v)] \land (p \in \hat{V}_E)$ if $p \in \hat{V}_M$.

The $env\_move$ Function The abstraction of $t_E$ is achieved by introducing a new function, $env\_move$. Context switches from $t_M$ to $t_E$ are modeled explicitly by calls to $env\_move$. The body of $env\_move$ changes during the run of Algorithm 4.1. However, it always has the property that it over-approximates the set of finite (possibly of length zero) computations of $t_E$ in $P$, that are reachable in $P$ (this is formalized by Definition 4.2.5). The $env\_move$ function is called at every cut-point location in $t_M$, as determined by an interleaving reduction analysis.

Commands of $P_M$ The code of $P_M$ always consists of the original code of $t_M$, with the following changes:

- The $env\_move$ function (which contains assumptions about the environment) is part of $P_M$.
- Calls to $env\_move$ are included at every chosen cut-point. These are added during initialization. Exactly one $env\_move$ call is added at each cut-point, meaning that there is at least one original command of $t_M$ between every two $env\_move$ calls in $P_M$. 


• New assertions may be included (only) directly before an `env.move` call. There may be several such new assertions before a single `env.move` call. These are added during the run of Algorithm 4.1).

**Observation 4.1.1.** The last two properties imply that the next command after `env.move` must be an original command of $t_M$. Clearly, by the second property above, it cannot be another `env.move` call. Further, when new assertions are added, they always immediately precede an `env.move` call (but after numerous such additions, there may be several subsequent newly added assertions before a particular `env.move` call). Therefore, if the next command after the `env.move` were some newly added `assert(b)`, the next commands afterwards must have been several (possibly zero) additional new assertions (part of the same sequence as `assert(b)`), followed by the `env.move` call before which `assert(b)` was added. This would have meant no original command of $t_M$ between two consecutive `env.move` calls, and it contradicts the second property.

**Computations of $P_M$** There are no additional commands in $P_M$ other than the ones listed above. Original commands of $t_M$ are also neither modified nor reordered. In particular, the control flow of $t_M$ is preserved within $P_M$. Hence, any computation of $P_M$ can be viewed as a periodic repetition of the following “extended steps”, where each step consists of:

1. A sequence of original commands of $t_M$, between cut-point locations, with no inner cut-point location.
2. A sequence of new assertions accumulated before an `env.move` call.
3. An execution of the `env.move` function.

Further, for every two such adjacent “extended steps”, the label reached in $t_M$ after the last original command (of $t_M$) of the former extended step, must be the same as the label in $t_M$ of the first original command in the latter extended step. Figure 4.1 illustrates such a typical computation of $P_M$.

**Observation 4.1.2.** Note that the set of new assertions before an `env.move` call may be empty. Specifically, after initialization - none of the `env.move` calls have new assertions preceding them. Additionally, since we include the initial label of each thread in the set of cut-point labels, there will be an `env.move` call in $P_M$ before the first original command of $t_M$, and possibly additional new assertions before that `env.move` call.

To address the original label in $t_M$, Algorithm 4.1 maintains a mapping, denoted $Lab$, that maps each label of $P_M$ (except those inside `env.move`) to the corresponding label in $t_M$.

**Definition 4.1.3 (Lab).** Let $\hat{l}$ be a label of $P_M$, not inside `env.move`, with $cmd(\hat{l}) = c$. We define $Lab(\hat{l})$ recursively as follows:

• If $c$ is an original command of $t_M$, appearing at label $l$ in $t_M$, then $Lab(\hat{l}) = l$. 

19
Figure 4.1: A typical computation of $P_M$ is a periodic repetition of extended steps consisting of: original commands of $t_M$, newly added assertion and the `env_move` function.

- If $c$ is an `env_move` call or a new assert, $\text{Lab}(\hat{l}) = \text{Lab}(\hat{l}')$, where $\hat{l}'$ is the next label in $P_M$ after the `env_move` or the assert command (resp.).

$\text{Lab}$ is well defined, as the next label after `asserts` or `env_move` calls is uniquely defined, and a sequence of such commands cannot form a loop, without introducing at least a single command originally from $t_M$. Technically, for each label of $P_M$, the mapping holds the label of the next original command of $t_M$ to be executed. Clearly, this definition is not injective, as multiple labels of asserts and `env_move` calls can be mapped to the same label of an original command. For example, consider the sequential program $P_0$ in Figure 4.3, which is based on $t_0$ from Figure 4.2. In this example, line 10 of $P_0$ with the command `turn = 1`, as well as the preceding line 9 with an `env_move` call, are both mapped to line 8 of $t_0$ with the original `turn = 1` command. Thus, $\text{Lab}(9) = \text{Lab}(10) = 8$.

### 4.2 Representation of $P$ Within $P_M$

In this section, we describe the relation between states of the sequential program $P_M$, and states of the concurrent program $P$. We also formalize the relation between computations of $t_M$ in $P$, and computations in $P_M$ that are restricted to original commands of $t_M$ (i.e., between consecutive cut-points).

The purpose of $P_M$ is to combine a precise representation of $t_M$, with an over-approximation of $t_E$ (represented by the `env_move` function). Together, this combination provides an over-approximation of the computations of $P$. Every state of $P_M$ can be viewed as an abstract state,
representing several states of \( P \). The formal relation is given by the following definition:

**Definition 4.2.1 (Extend).** Let \( \hat{s} = (\hat{l}, \hat{\sigma}) \) be a state of \( P_M \), s.t. \( \hat{l} \) is not inside \textit{env\_move}. We define the set \( \text{Extend}(\hat{s}) \) to be the set of all states \( s = (l, \sigma) \) of \( P \) such that:

- \( l_M(s) = \text{Lab}(\hat{l}) \)
- If \( pc \in \hat{V}_M \) then \( l_E(s) = \hat{\sigma}(pc) \)
- For every \( v \in V: v \in \hat{V}_M \Rightarrow \sigma(v) = \hat{\sigma}(v) \)

Intuitively, the states appearing in \( \text{Extend}(\hat{s}) \) are all the states of \( P \) that agree with \( \hat{s} \) on the program location of \( t_M \) (and of \( t_E \) if \( pc \in \hat{V}_M \)), and agree with \( \hat{s} \) on all the variables in \( V \cap \hat{V}_M \).

**Observation 4.2.2.** Note that the set \( \text{Extend}(\hat{s}) \) is never empty for any state \( \hat{s} \) of \( P_M \). This is because any extension of \( \sigma(s) \) to the variables in \( V \setminus \hat{V}_M \) will form a legal state \( s \) of \( P \) (not necessarily reachable) which is in \( \text{Extend}(\hat{s}) \).

**Observation 4.2.3.** If \( s \in \text{Extend}(\hat{s}) \), then for every formula \( \gamma \) over \( \hat{V}_M \cap (V \cup \{pc\}) \), it holds that \( s \models \gamma \iff \sigma(\hat{s}) \models \gamma \).

This is true, since Definition 4.2.1 implies that \( s \) and \( \hat{s} \) have identical values on all the variables in \( \gamma \).

Next, we would like to extend our observation to initial states using the following lemma:

**Lemma 4.2.4.** Let \( \hat{s} \) be an initial state of \( P_M \). Then there exists an initial state \( s \) of \( P \), such that \( s \in \text{Extend}(\hat{s}) \).

**Proof.** Let \( \phi_{\text{init}} \triangleq \bigwedge_{v \in U} [v = \sigma_{\text{init}}(v)] \) be the initialization formula of \( P \). We define a state \( s \) as follows:

- \( l_M(s) = l_M^{\text{init}} \).
- \( l_E(s) = l_E^{\text{init}} \).
- For every \( v \in U: \sigma(s)(v) = \sigma_{\text{init}}(v) \).
- For \( v \in \hat{V}_M \setminus U: \sigma(s)(v) = \sigma(\hat{s})(v) \).
- If \( v \not\in \hat{V}_M \) and \( v \not\in U: \sigma(s)(v) \) can be chosen arbitrarily.

\(^1\)The difference in notation (i.e., “\( s \)” vs. “\( \sigma(\hat{s}) \)” results from the special status of the variable \( pc \). It is not part of \( \sigma(s) \), but it is a part of \( s \), with its value indicated by \( l_E(s) \). In \( P_M \), on the other hand, it is a regular variable, and is part of \( \sigma(\hat{s}) \).
By the first three properties, \( s \) is clearly an initial state of \( P \). We need to show that \( s \in \text{Extend}(\hat{s}) \).

The code of \( P_M \) starts with a (possibly empty) sequence of new assertions, an \texttt{env.move}, and then the first original command \( c \) at label \( \text{init} \) of \( t_M \) (see Observation 4.1.2). Therefore, \( \text{cmd}(\hat{s}) \) is either such a new assertion or an \texttt{env.move} call, and there is no other original command of \( t_M \) between \( \text{cmd}(\hat{s}) \) and \( c \). Hence, by Definition 4.1.3, \( \text{Lab}(l(\hat{s})) = \text{init} \).

Since \( \hat{s} \) is initial, \( \sigma(\hat{s}) = \hat{\phi}_\text{init} \). Thus, if \( pc_E \in \overline{V_M} \), then \( \sigma(\hat{s})(pc_E) = l(\hat{s}) \). Similarly, if \( v \in \overline{V_M} \cap U \), then \( \sigma(\hat{s})(v) = \sigma(\text{init})(v) = \sigma(v)(v) \). For \( v \in \overline{V_M} \setminus U \), \( \sigma(\hat{s})(v) = \sigma(s)(v) \) by the construction of \( s \). Thus, \( s \in \text{Extend}(\hat{s}) \). \qed

After defining the compatibility between states of \( P \) and states of \( P_M \), we can now address the compatibility of computations. First, we define in what sense the \texttt{env.move} function over-approximates computations of \( t_E \). Intuitively, the \texttt{env.move} function over-approximates the set of finite reachable computations of \( t_E \) in terms of their input-output relation. Formally, this means the following.

**Definition 4.2.5 (Over-approximation).** We say that \texttt{env.move} over-approximates the computations of \( t_E \) in \( P \) if for every reachable (and possibly empty) computation \( \rho = s \xrightarrow{\text{init}} s' \) of \( t_E \) in \( P \) from a cut-point state \( s \) to a cut-point state \( s' \), and for every state \( \hat{s} \) of \( P_M \) s.t. \( s \in \text{Extend}(\hat{s}) \), and \( \text{cmd}(\hat{s}) \) is an \texttt{env.move} call, there exists a computation \( \hat{\rho} = \hat{s} \xrightarrow{\cdot} \cdot \xrightarrow{\cdot} \cdot \xrightarrow{\cdot} s' \) of \( P_M \) s.t.

- \( l(s') \) is the next command in \( P_M \) after \texttt{env.move}, and for every inner state \( \hat{s}'' \) in \( \hat{\rho} \), \( l(\hat{s}'') \) is a label inside \texttt{env.move} (i.e., \( \hat{\rho} \) is a complete single execution of \texttt{env.move})
- \( s' \in \text{Extend}(\hat{s}') \)

The definition implies that if \( t_E \) can move (in any finite number of steps) from a state \( s \) satisfying some condition \( \alpha \) to a state \( s' \) satisfying some condition \( \beta \), \texttt{env.move} should also allow reaching a state satisfying \( \beta \), when it is called from a state satisfying \( \alpha \). In order to prove that \texttt{env.move} always over-approximates the computations of \( t_E \) in \( P \), we need to consider the possible refinements and changes applied to \texttt{env.move} by Algorithm 4.1. This is proved in Lemma 5.1.1, after the complete presentation of the algorithm in this chapter. We will later see (Lemma 5.1.3) that in addition to this over-approximation, \( P_M \) also under-approximates the set of states leading to an error, using the new assertions. This under-approximation also incorporates partial information about \( t_E \).

As opposed to \( t_E \), the representation of \( t_M \) within \( P_M \) is precise. In fact, this property is based on the general structure of \( P_M \), as described in this section. Hence, we can prove the following two useful lemmas, formalizing the meaning of “precise representation” of \( t_M \) within \( P_M \). Lemma 4.2.6 maps computations of \( P_M \) that do not use \texttt{env.move}, to representative computations of \( t_M \) in \( P \). Lemma 4.2.7 maps computations \( \rho \) of \( t_M \) in \( P \) to computations \( \hat{\rho} \) in \( P_M \) that represent them, where a subtle point is that \( \hat{\rho} \) may correspond to a prefix \( \rho' \) of \( \rho \),
followed by an error state. This may happen, in case that Algorithm 4.1 already learned that the last state of \( \rho' \) leads to an error. The soundness proof at Section 5.1 uses both of these lemmas (as well as additional properties of the algorithm, presented later) to complete the correctness argument of our algorithm.

**Lemma 4.2.6 (PM representation I).** Let \( \hat{\rho} = s \rightarrow \cdots \rightarrow \hat{s}' \) be a computation of \( P_M \), along a path in which all commands are not inside env_move and are not env_move calls, and such that \( \hat{s}' \neq \epsilon \). Then for every state \( s \in \text{Extend}(\hat{s}) \) there exists a computation \( \rho \) of \( t_M \) in \( P \) from \( s \) to some state \( s' \) s.t. \( s' \in \text{Extend}(\hat{s}') \).

**Proof.** Let \( k \) be the length of the computation \( \hat{\rho} \). The proof is by induction on \( k \).

**Base Case:** Assume \( k = 0 \). Let \( s \) be a state of \( P \). If we choose \( s' = s \), there is a zero length computation (of \( t_M \)) in \( P \) from \( s \) to \( s' \) (from \( s \) to itself). Since \( k = 0 \), \( \hat{s}' = \hat{s} \), and therefore \( s \in \text{Extend}(\hat{s}) \) implies \( s' \in \text{Extend}(\hat{s}') \).

**Induction Step:** Let \( k > 0 \). Assume that the lemma holds for computations in \( P_M \) of length \((k-1)\), and let \( \hat{\rho} = s \rightarrow \cdots \rightarrow \hat{s}'' \rightarrow \hat{s}' \) be a computation of length \( k \) in \( P_M \), as described above. Then \( \hat{\rho}' = \hat{s} \rightarrow \cdots \rightarrow \hat{s}'' \) is a computation of length \((k-1)\), for which the induction hypothesis holds. Let \( s \in \text{Extend}(\hat{s}) \). By the induction hypothesis, there exists some state \( s'' \in \text{Extend}(\hat{s}'') \) and a computation \( \rho' \) of \( t_M \) in \( P \) from \( s \) to \( s'' \). We define a state \( s' = (l', \sigma') \) as follows:

- \( l_M(s') = \text{Lab}(l(\hat{s}')) \).
- \( l_E(s') = l_E(s'') \).
- \( \forall v \in V: \)
  - If \( v \in \overline{V_M} \) then \( \sigma'(v) = \sigma(\hat{s}')(v) \).
  - If \( v \notin \overline{V_M} \) then \( \sigma'(v) = \sigma(s'')(v) \).

It is sufficient to show that there exists a computation \( \rho'' \) of \( t_M \) in \( P \) from \( s'' \) to \( s' \) and that \( s' \in \text{Extend}(\hat{s}') \), since \( \rho \triangleq \rho' \cdot \rho'' \) will then be a computation from \( s \) to \( s' \), as required. Note that in order to prove that \( s' \in \text{Extend}(\hat{s}') \), we only need to show that if \( \rho_{C_E} \in \overline{V_M} \) then \( l_E(s') = \sigma(\hat{s}')((\rho_{C_E}) \) as the other two requirements of Definition 4.2.1 are met immediately by the definition of \( s' \).

Let \( c = \text{cmd}(\hat{s}'') \). \( c \) is the command used by \( \hat{\rho} \) to move from \( s'' \) to \( \hat{s}' \). Since \( c \) cannot be an env_move call and is also not inside env_move, there are two possible cases:

- \( c \) is an original command of \( t_M \).
- \( c \) is a new assertion in \( P_M \).
1. Assume first that \( c \) is an original command of \( t_M \). Since original commands of \( t_M \) cannot change \( \text{pc}\mathcal{E} \) in \( P_M \), we get \( \sigma(\hat{s}'')(\text{pc}\mathcal{E}) = \sigma(\hat{s}'')(\text{pc}\mathcal{E}). \) Since \( s'' \in \text{Extend}(\hat{s}'') \), \( \text{pc}\mathcal{E} \in \overline{V}_M \) implies \( l_E(s'') = \sigma(\hat{s}'')(\text{pc}\mathcal{E}). \) Combining this with the definition of \( s' \), we get \( l_E(s') = l_E(s'') \), and all together \( l_E(s') = \sigma(s')(\text{pc}\mathcal{E}). \) It is now sufficient to show that \( s' \in \text{next}(s'', t_M) \) (a computation of length one from \( s'' \) to \( s' \)).

Let \( V_{\text{act}} = \text{Vars}(c) \). The command \( c \) can only change the variables in \( V_{\text{act}} \) and the possible modifications are independent of variables not in \( V_{\text{act}} \). Since \( c \) is in \( P_M \), \( V_{\text{act}} \subseteq \overline{V}_M \) and hence by the definition of \( s' \), \( \sigma'|_{V_{\text{act}}} = \sigma(\hat{s}')|_{V_{\text{act}}}. \) Since \( s'' \in \text{Extend}(s'') \), we also get \( \sigma(s'')|_{V_{\text{act}}} = \sigma(s'')|_{V_{\text{act}}}. \) Since \( s' \in \text{next}(s'') \), it means that \( \sigma(s'')|_{V_{\text{act}}} \) can be obtained from \( \sigma(s'')|_{V_{\text{act}}} \) by performing \( c \), and hence \( \sigma'|_{V_{\text{act}}} \) can be obtained from \( \sigma(s'')|_{V_{\text{act}}} \) by performing \( c \). For a variable \( v \notin V_{\text{act}} \), \( \sigma'(v) = \sigma(s'')(v) \) (by the definition of \( s' \) if \( v \notin \overline{V}_M \), and because \( \sigma(s'')(v) = \sigma(s'')(v) \) for \( v \in \overline{V}_M \setminus V_{\text{act}} \), as \( c \) cannot change variables outside \( V_{\text{act}} \). Hence, \( \sigma' \) can be obtained from \( \sigma(s'') \).

As for the labels, \( l_E(s') = l_E(s'') \) by the definition of \( s' \). We also need to show that \( l_M(s') \) is the label reached after performing \( c \) from \( s'' \). Given the state \( s'' \), the label of the next command depends on \( l_M(s'') \), the location of \( s'' \) within \( t_M \), and of \( \sigma(s'')|_{V_{\text{act}}}. \) The label of the next command is unique given this valuation (although an if-command may move to several possible labels, it can only have one target given a specific valuation of \( V_{\text{act}} \). Since \( s'' \in \text{Extend}(\hat{s}'') \), it holds that \( l_M(s'') = \text{Lab}(l(\hat{s}'')) \) and \( \sigma(s'')|_{V_{\text{act}}} = \sigma(s'')|_{V_{\text{act}}}. \) Hence, the label reached in \( t_M \) after performing \( c \) from \( s'' \), is also given by \( \text{Lab}(l(\hat{s}'')) \), where \( s' \) is the state obtained after performing \( c \) on \( \hat{s}'' \). Luckily, this is exactly how \( l_M(s') \) was defined.

2. Otherwise, \( c \) is a newly added assertion command. Since, \( s' \neq \epsilon \), the assertion is not violated and hence, \( \sigma(s'') = \sigma(s'). \) In particular, \( \text{pc}\mathcal{E} \in \overline{V}_M \) implies \( \sigma(s'')(\text{pc}\mathcal{E}) = \sigma(\hat{s}')|_{\text{pc}\mathcal{E}} \) as before, and hence again we have \( l_E(s') = l_E(s'') = \sigma(\hat{s}')|_{\text{pc}\mathcal{E}} = \sigma(\hat{s}')|_{\text{pc}\mathcal{E}}. \)

It is now sufficient to show that \( s' = s'' \) (a computation of length zero). By Definition 4.2.1, \( l_M(s'') = \text{Lab}(l(\hat{s}'')) \), which equals to \( \text{Lab}(l(\hat{s}')) \) by Definition 4.1.3. With the definition of \( s' \) we get \( l_M(s') = l_M(s'') \). The rest of the equalities follow immediately from the definition of \( s' \), the equality \( \sigma(s'') = \sigma(s') \) and the assumption \( s'' \in \text{Extend}(\hat{s}'') \).

**Lemma 4.2.7** (*P_M* representation II). *Let \( \rho \) be a computation of \( t_M \) in \( P \) from a state \( s \) to a state \( s' \) such that there are no inner cut-point states within \( \rho \) (but \( s \) and \( s' \) may be cut-point states). Let \( \hat{s} \) be a state of \( P_M \) such that \( s \in \text{Extend}(\hat{s}) \), and \( \text{cmd}(\hat{s}) \) is an original command of \( t_M \). Then there exists a computation \( \hat{\rho} \) of \( P_M \) from \( \hat{s} \) to a state \( \hat{s}' \) such that either \( \hat{s}' = \epsilon \), or the following conditions hold:

- \( \hat{s}' \in \text{Extend}(\hat{s}) \)

24
• If \( s' \) is a cut-point state then \( \text{cmd}(s') \) is an \text{env\_move} call.

• If \( s' \) is not a cut-point state then \( \text{cmd}(s') \) is an original command of \( t_M \) (and by the first requirement \( \text{cmd}(s') = \text{cmd}(s') \)).

**Proof Sketch.** The full proof of this lemma is similar to the previous one, and is omitted. As before, the main idea is that performing an original command of \( t_M \) on some state \( s \) of \( P \), has “the same” effect as performing it on a state \( \hat{s} \) of \( P_M \) satisfying \( s \in \text{Extend}(\hat{s}) \). We can construct \( \hat{\rho} \) by adding each command from \( \rho \) to \( \hat{\rho} \). Since \( V_M \subseteq \hat{V}_M \), and \( s \in \text{Extend}(\hat{s}) \), it is possible to perform the same series of commands in \( P_M \) as well. The two more subtle points in the proof are the following:

• The formed sequence of commands, \( \hat{\rho} \), indeed forms a computation in \( P_M \), because \( P_M \) preserves the control flow of \( t_M \), and because there are no inner cut-point states in \( \rho \). Therefore, after performing a command from \( \rho \) in \( P_M \), the next command to be executed is also an original command of \( t_M \), and cannot be an \text{env\_move} call or a newly added assertion. Thus, we can continue with the construction of \( \hat{\rho} \) for every command in \( \rho \).

• The last state \( \hat{s}' \) may be \( \epsilon \). If \( s' \) is a cut-point state, then the last step of \( \hat{\rho} \) should lead to an \text{env\_move} call. However, some newly added assertions, not appearing in \( t_M \), may appear before that \text{env\_move} call in \( P_M \). If \( s' \) contradicts one of these assertions, the computation in \( P_M \) will move to \( \epsilon \) (before reaching the \text{env\_move} call). We will later see (Lemma 5.1.3) that this can only happen when the relevant state \( s' \) is a state from which an error is reachable. Algorithm 4.1 plants such new assertion about future errors, when it learns the conditions that guarantee them. This is explained more thoroughly in Section 4.4.

If \( s' \) satisfies all the assertions before the \text{env\_move} call, we can simply add all these assertions to \( \hat{\rho} \), making \( \text{cmd}(s') \) an \text{env\_move} call as needed. This is legal, as satisfied assertions do not change \( \sigma(\hat{s}') \). If \( s' \) is not a cut-point state at all, \( \hat{\rho} \) would not reach an \text{env\_move} call (nor the new assertions added before such calls), making \( \text{cmd}(\hat{s}') \) an original command of \( t_M \).

\[ \square \]

### 4.3 Initial Construction of \( P_M \)

Algorithm 4.1 starts by constructing the initial version of \( P_M \), based on the code of \( t_M \). To do so, it adds explicit calls to \text{env\_move} in every cut point label of \( t_M \). In addition, the algorithm constructs the initial \text{env\_move} function, which havocs (i.e., assigns a non deterministic value to) every shared variable of \( t_E \) and \( t_M \) that is written by \( t_E \). This function will gradually be refined to represent the environment in a more precise way.
Example 4.3.1. We use the Peterson’s algorithm [33] for mutual exclusion, presented in Figure 4.2, as a running example. The algorithm contains a busy-wait loop in both threads, where each thread enters its critical section (i.e. leaves the while loop) only after the turn variable indicates that it is its turn to enter, or the other thread gave up on its claim to enter the critical section. In order to specify the safety property (mutual exclusion), we use additional variables cs0, cs1 which indicate that t0 and t1 (resp.) are in their critical sections. We would like to verify that t0 and t1 cannot be in their critical section at the same time, i.e ¬cs0 ∨ ¬cs1 always holds. The safety property is specified by the assert(!cs1) commands at Line 13 of Figure 4.2. We could have also used the full assertion assert(!cs0 || !cs1) at every location in t0. However, the assert(!cs1) at Line 13 suffices, as cs0 is always true there, and always false everywhere else (In fact, our algorithm performs an initial static analysis that makes this simplification automatic).

Assume that t0 was chosen as the main thread and t1 as the environment thread. We generate a sequential program P0, based on the code of t0: we add env_moves at every cut point, as determined by our interleaving reduction analysis. The result is illustrated by Figure 4.3. In our case, the cut-points are after all commands except for those in line 16 and 19, where t0 changes the local variable cs0, and except for the while(true) command at Line 7, which does not read any variable.

The initial env_move only havoccs all variables of P0 that are written by t1, i.e., claim1, turn, cs1 (see Figure 4.6).

4.4 Iteration of the MainThreadCheck Algorithm

Each iteration of Algorithm 4.1 starts by applying a sequential model checker to check whether there exists a violating path (that may involve calls to env_move) in PM (Line 3). If not, we conclude that the concurrent program is safe (Line 20), as the env_move function over-approximates the computations of the environment. If an assertion violation is detected in PM, the model checker returns a counterexample in the form of a violating path. If there are no env_move calls in the path, (Line 5), it means that the path represents a genuine violation obtained by a computation of the original main thread, and hence the program is unsafe (Line 6).

Otherwise, the violation relies on environment moves, and as such it might be spurious. We therefore analyze this counterexample as described in Section 4.5. The purpose of the analysis is to check whether t_E indeed enables the environment transitions used along the path. If so, we find “promises of error” for the violated assertion at earlier stages along the path and add them as new assertions in PM. Intuitively speaking, a “promise of error” is a property ensuring that t_E can make a sequence of steps that will allow t_M to violate its assertion. Such a property may

2In fact, since the condition claim1 && turn != 0 in line 12 of Figure 4.3 is not evaluated atomically in C programs, another env_move is required after reading claim1 and before reading turn. This can be solved by rewriting the program s.t. it first assigns the values of claim1 and turn to two new local variables, calls env_move between these two assignments, and only evaluates the new local variables to check whether the condition holds. We omit this here for simplicity.
bool claim0 = false, claim1 = false;
int turn;

void t0() {
    while (true) {
        claim0 = true;
        turn = 1;
        while (claim1 && turn != 0) {
            }
        cs0 = true;
        // CRITICAL_SECTION
        assert(!claim1);
        cs0 = false;
        claim0 = false;
    }
}

void t1() {
    while (true) {
        claim1 = true;
        turn = 0;
        while (claim0 && turn != 1) {
            }
        cs1 = true;
        // CRITICAL_SECTION
        cs1 = false;
        claim1 = false;
    }
}

Figure 4.2: Peterson’s mutual exclusion algorithm for two threads t0 and t1.

depend on both threads, and hence it is defined over \( V \cup \{P_C E\} \) (\( P_C M \) is given implicitly by the label of the assertion in \( P_M \)). Formally, we have the following definition:

**Definition 4.4.1.** Let \( \psi, \psi’ \) be formulas over \( V \cup \{P_C E\} \) and let \( l, l’ \) be labels of \( t_M \). We say that \( (l, \psi) \) is a promise of \( (l’, \psi’) \) if for every state \( s \) of \( P \) s.t. \( l_M(s) = l \) and \( s \models \psi \) there exists a computation in \( P \) starting from \( s \) to a state \( s’ \) s.t. \( l_M(s’) = l’ \) and \( s’ \models \psi’ \).

If \( (l, \psi) \) is a promise of \( (l’, \neg b) \) and \( cmd(l’) = assert(b) \), then we say that \( (l, \psi) \) is a promise of error.

A promise (and a promise of error) \( (l, \psi) \) is defined “from the point of view” of \( t_M \). That is, if \( \psi \) holds when \( t_M \) is at a specific label \( l \), then \( (l’, \psi’’) \) (or an error) can be reached. Note that the definition refers to computations of \( P \), and is independent of our construction of \( P_M \) and \( P_E \).

Note further that the definition is transitive. Specifically, if \( (l, \psi) \) is a promise of \( (l’, \psi’) \) and \( (l’, \psi’’) \) is a promise of error, then \( (l, \psi) \) is also a promise of error. The proof follows.

**Lemma 4.4.2** (Transitivity of promises of error). Let \( (l, \psi) \) be a promise of \( (l’, \psi’) \), and let \( (l’, \psi’’) \) be a promise of error. Then \( (l, \psi) \) is also a promise of error.

**Proof.** Since \( (l’, \psi’’) \) is a promise of error, there exists a label \( l'' \in L \) s.t. \( cmd(l'') = assert(b) \) and \( (l', \psi'') \) is a promise of \( (l'', \neg b) \). It is sufficient to show that \( (l, \psi) \) is also a promise of \( (l'', \neg b) \). Let \( s \) be a state of \( P \) s.t. \( l_M(s) = l \) and \( s \models \psi \). By Definition 4.4.1, there exists a computation \( \rho \) in \( P \) from \( s \) to a state \( s' \) s.t. \( l_M(s') = l' \) and \( s' \models \psi' \). Again, by Definition 4.4.1,
there exists another computation $\rho'$ in $P$ from $s'$ to a state $s''$ s.t. $l_M(s'') = l''$ and $s'' \models \neg b$. The concatenation of these two computations, $\rho'' = \rho \cdot \rho'$, is a computation from $s$ to $s''$ and hence $(l, \psi)$ is indeed a promise of $(l'', \neg b)$. 

**Outcome** Each iteration of Algorithm 4.1 ends with one of these four scenarios:

1. There is no violation in $P_M$ and hence $P$ is safe (Line 20).
2. The algorithm terminates having found a genuine counterexample for $P$ (Line 6).
3. The obtained counterexample is found to be spurious since an execution of $\text{env} \_\text{move}$ along the path is proved to be infeasible. The counterexample is eliminated by refining the $\text{env} \_\text{move}$ function (Line 15, also see item (4) in the next section).
4. Spuriousness of the counterexample remains undetermined, but a new promise of error is generated before the last $\text{env} \_\text{move}$ call in the violating path of $P_M$. We augment $P_M$ with a new assertion, representing this promise of error (Line 17).

The analysis of a potentially spurious violating path of $P_M$, as well as the generation of new promises of error (step (5)) and the refinement of $\text{env} \_\text{move}$ when the counterexample is found to be spurious (step (4)), are explained in detail in Section 4.5.

### 4.5 Analyzing a Potentially Spurious Violating Path

This section thoroughly explains the analysis of a potentially spurious violating path in $P_M$, i.e., a path that contains at least one $\text{env} \_\text{move}$ call, obtained in an iteration of Algorithm 4.1. Let $\hat{\pi} = \hat{l}_0, \ldots, \hat{l}_{n+1}$ be such a violating path of $P_M$ returned by the sequential model checker. Since $\hat{\pi}$ is a violating path, we know that $\hat{l}_{n+1} = \hat{l}_e$ and that $\text{cmd}(\hat{l}_n) = \text{assert}(b)$, for some condition $b$. Let $\hat{l}_e$, for some $0 \leq k \leq (n - 1)$, be the label of the last $\text{env} \_\text{move}$ call in $\hat{\pi}$. For convenience, we assume that labels within $\text{env} \_\text{move}$ are omitted from $\hat{\pi}$. That is, $\hat{l}_{k+1}$ is the label of the next command in $P_M$ after the $\text{env} \_\text{move}$.

We perform the following steps, illustrated by Figure 4.4:

1. **Computing Condition After the Environment Step:** We compute (backwards) the weakest precondition of $\neg b$ w.r.t. the path $\hat{\pi}_{\text{end}} = \hat{l}_{k+1}, \ldots, \hat{l}_n$ to obtain a formula $\beta = \text{wp}(\hat{\pi}_{\text{end}}, \neg b)$ (Line 10 of Algorithm 4.1). Recall that $\beta$ has the property that for every state $\hat{s}$ of $P_M$, $\sigma(\hat{s}) \models \beta$ iff there exists a computation $\hat{\rho}$ in $P_M$ starting from $\hat{s}$ whose path is $\hat{\pi}_{\text{end}}$, that reaches a state $\hat{s}'$ s.t. $\sigma(\hat{s}') \models \neg b$. Also, note that there are no $\text{env} \_\text{move}$ calls in $\hat{\pi}_{\text{end}}$.

2. **Computing Condition Before the Environment Step:** We compute (forward) a postcondition $\alpha = \text{post}(\hat{\pi}_{\text{start}}, \hat{\phi}_{\text{init}})$ starting from $\hat{\phi}_{\text{init}}$ for the path $\hat{\pi}_{\text{start}} = \hat{l}_0, \ldots, \hat{l}_k$. Note that $\alpha$ is not necessarily a strongest postcondition. Therefore, in order to ensure progress, we need to make sure that if $\hat{\pi}_{\text{start}}$ ends with a suffix of asserts, $\text{assert}(c_1), \ldots, \text{assert}(c_m)$, they are all taken into account in our postcondition computation (e.g., by conjoining each $c_i$ with
post(\hat{\pi}_{\text{start}}, \hat{\phi}_{\text{init}})). Formally, this means that \( \alpha \Rightarrow c_i \) for every \( 1 \leq i \leq m \). This requirement is used by Lemma 5.2.4 (which is part of the progress proof for Algorithm 4.1).

Recall that \( \alpha \) has the property that for every computation \( \hat{\rho} \) from a state \( \hat{s} \) of \( P_M \) to state \( \hat{s}' \) whose path is \( \hat{\pi}_{\text{start}} \): if \( \sigma(\hat{s}) \models \hat{\phi}_{\text{init}} \) then \( \sigma(\hat{s}') \equiv \alpha \). Note that this property is not compromised by our progress requirement, as the last commands of every such computation are the sequence assert\((c_1), \ldots, \text{assert}(c_m)\). If \( \sigma(\hat{s}') \neq c_i \) for some \( 1 \leq i \leq m \), \( \hat{\rho} \) cannot pass assert\((c_i)\) and reach the env.move call.

We strive to compute a postcondition which is “as precise as possible”. However, soundness does not rely on this and we can choose \( \alpha \) as an arbitrary postcondition. For progress, the minimal necessary requirement is the one mentioned above.

(3) Environment Query: We compute \( \psi = \text{Reach}_{(t_E)}(\alpha, \beta) \) (Line 12).

Example 4.5.1. Figure 4.5 presents a prefix of \( P_M \) after a few iterations of the algorithm, before the first refinement of env.move (i.e., \( P_M \) still uses the initial env.move function). The previous iterations found new promises of error, and augmented \( P_M \) with new assertions. Consider the initial conditions from Figure 4.2, i.e., \( \phi_{\text{init}} \triangleq [\text{claim0} = \text{claim1} = cs1 = cs0 = false] \). Assume that our sequential model checker found the violation given by the next path: 2, 3, 4, 5, 6, 7, 8, 9, reaching and violating assert\((!cs1 || (\text{claim1} \&\& \text{turn} != 0))\) at Line 9 (i.e., \( b = [\neg cs1 \lor (\text{claim1} \land \text{turn} \neq 0)] \)).

To check whether the last env.move call at Line 7 represents a real computation of \( t_1 \), we compute the weakest precondition of the condition \( \neg b \triangleq cs1 \land (\neg \text{claim1} \lor \text{turn} = 0) \), taken from the violated assertion at Line 9, w.r.t. the path \( \hat{\pi}_{\text{end}} = 8, 9 \). The result is \( \beta = \wp(\hat{\pi}_{\text{end}}, \neg b) = (cs1 \land \neg \text{claim1}) \). The computation of \( \alpha = \text{post}(\hat{\pi}_{\text{start}}, \hat{\phi}_{\text{init}}) \) for the path \( \hat{\pi}_{\text{start}} = 2, 3, 4, 5, 6 \) yields \( \alpha = [\neg cs0 \land \text{claim0} \land (\neg cs1 \lor \text{claim1})] \). Note the conjunction with \((\neg cs1 \lor \text{claim1})\), which is the condition inside the last assert of \( \hat{\pi}_{\text{start}} \). We then generate an environment query \( \text{Reach}_{(t_E)}(\alpha, \beta) \).

(4) Refining the env.move Function: If \( \psi = \text{FALSE} \) (Line 13) it means that there is no reachable computation of \( t_E \) in \( P \) from a state \( s \) such that \( s \models \alpha \) to a state \( s' \) such that \( s' \models \beta \) (due to the properties of an environment query in Definition 3.2.1). We invoke a generalization method (see Section 4.6) to obtain two formulas \( \alpha', \beta' \) such that \( \alpha \Rightarrow \alpha', \beta \Rightarrow \beta' \) and still \( \text{Reach}_{(t_E)}(\alpha', \beta') = \text{FALSE} \) (Line 14). Finally, we refine env.move to eliminate the environment transition from \( \alpha' \) to \( \beta' \) (Line 15). Figure 4.4(a) illustrates this step.

The refinement is done by introducing in env.move, after the variables are havocked, the command \((\text{if } (\alpha'(W.\text{old})) \text{ assume}(\neg \beta'))\), where \( W.\text{old} \) are the values of the variables before they are havocked in env.move (these values are copied by env.move, using additional auxiliary variables, to allow evaluating \( \alpha' \) on the values of the variables before env.move is called). The commands ensure that if the condition \( \alpha' \) is met when entering the env.move function, then condition \( \beta \) is satisfied when it exits. Since reachable computations from \( \alpha' \) to \( \beta' \)
were proven by the environment query to be infeasible in $t_E$, we are ensured that $\text{env} \_ \text{move}$ remains an over-approximation of the reachable computations of $t_E$.

**Example 4.5.2.** Figure 4.6 presents the $\text{env} \_ \text{move}$ function before and after the refinement step resulting from the reachability query described in Example 4.5.1, with $\alpha = [\neg cs0 \land \text{claim0} \land (\neg cs1 \lor \text{claim1})]$ and $\beta = (\text{cs1} \land \neg \text{claim1})$. The refinement step adds the two highlighted lines to the initial $\text{env} \_ \text{move}$ function. The call to $\text{Reach}(t_E)(\alpha, \beta)$ in this example results in $\psi = \text{FALSE}$. After generalization is applied (see (4) in Section 4.5, and Section 4.6), we obtain two formulas $\alpha' = \text{TRUE}, \beta' = \beta$ which indeed satisfy $\text{Reach}(t_E)(\alpha', \beta') = \text{FALSE}$, $\alpha \Rightarrow \alpha'$ and $\beta \Rightarrow \beta'$. Since $\text{Reach}(t_E)(\alpha', \beta') = \text{FALSE}$, there are no reachable computations of $t_E$ reaching a state satisfying $\beta' = \text{cs1} \land \neg \text{claim1}$. We therefore augment $\text{env} \_ \text{move}$ with a new constraint $(\text{if } (\alpha'[W/\text{W0}]) \text{ assume } (\neg \beta'))$, derived from this observation. Since $\alpha' = \text{True}$, this constraint is simplified to $(\text{if } (\text{true}) \text{ assume } (\neg \text{cs1} || \text{claim1}))$.

(5) **Adding Assertions:** If $\psi \neq \text{FALSE}$, then for every state $s$ satisfying $\psi$ there exists a computation of $t_E$ in $P$ from $s$ to a state $s'$ satisfying $\beta$. Since $\beta = \text{wp}(\hat{\pi}_{\text{end}}, \neg b)$ (see (1)), it is guaranteed that this computation can be extended (in $t_M$) along the path $\pi_{\text{end}}$, which does not use any environment moves, to reach a state $s''$ that violates the assertion $\text{assert}(b)$. This is illustrated in Figure 4.4(b). We therefore conclude that if $\psi$ is satisfied before the $\text{env} \_ \text{move}$ at location $\hat{l}_k$, a genuine violation can be reached, making $(\text{Lab}(\hat{l}_k), \psi)$ a promise of error. Therefore, we add a new label $\hat{l}'$ with $\text{cmd}(\hat{l}') = \text{assert}(\neg \psi)$ right before $\hat{l}_k$ (Line 17). The $\text{Lab}$ function is updated to make sure that each command is mapped to the same label as before, and $\text{Lab}(\hat{l}') = \text{Lab}(\hat{l}_k)$, making $(\text{Lab}(\hat{l}'), \psi)$ a promise of error as well. In addition, if $\psi$ includes a variable $v$ that is not in $\overline{V}_M$ (e.g., $\text{pc}_E$), then $v$ is added to $\overline{V}_M$, its declaration (and initialization, if exists) is added to $P_M$, and $\text{env} \_ \text{move}$ is extended to havoc $v$ as well (if it is written by $t_E$).

**Remark.** In both steps (4) and (5), the algorithm learns new information about the environment and is ensured to make progress. The formal proof appears in Section 5.2.

Intuitively, after the refinement in (4), the path $\hat{\pi}$ (or any transformation of $\hat{\pi}$ by adding new inner assertions, which may be added by Algorithm 4.1 in future iterations) is eliminated completely. This is because every computation that uses $\hat{\pi}_{\text{start}}$ would reach a state satisfying $\alpha$. Then, the refined $\text{env} \_ \text{move}$ forces the computation to reach $\neg \beta$, which means that the computation cannot reach and violate $\text{assert}(b)$ using $\hat{\pi}_{\text{end}}$. Note, however, that the blocked path $\hat{\pi}$ is a path of $P_M$. It might be the case that another violating path $\hat{\pi}'$ is found, which projects to the same path of $t_M$ as $\hat{\pi}$. For example, if $\text{assert}(b)$ is part of a sequence of newly added assertions before some $\text{env} \_ \text{move}$ call, $\hat{\pi}'$ can be identical to $\hat{\pi}$, except for the last assertion (i.e., $\hat{\pi}'$ will not violate $\text{assert}(b)$).

In (5), the analysis is more subtle. First, after adding the new assertion, the algorithm can search for computations violating that assertion (and having one less $\text{env} \_ \text{move}$ call). Nevertheless, the algorithm may also have to re-examine the path $\hat{\pi}$ (extended with the new
(a) If $\text{Reach}(t_E)(\alpha, \beta) = \text{FALSE}$, we search for more general $\alpha'$ and $\beta'$ which restrict the environment transition; (b) If $\text{Reach}(t_E)(\alpha, \beta) = \psi \neq \text{FALSE}$, then we know that $\psi$ leads to $\beta$ and that $\psi \land \alpha \neq \text{FALSE}$.

Figure 4.4: (a) If $\text{Reach}(t_E)(\alpha, \beta) = \text{FALSE}$, we search for more general $\alpha'$ and $\beta'$ which restrict the environment transition; (b) If $\text{Reach}(t_E)(\alpha, \beta) = \psi \neq \text{FALSE}$, then we know that $\psi$ leads to $\beta$ and that $\psi \land \alpha \neq \text{FALSE}$.

Figure 4.5: The sequential program $P_0$ after a few iterations of Algorithm 4.1.

void $P_0()$
{
    assert (!cs1) || claim1;
    env_move();
    while (true) {
        assert (!cs1) || claim1);
        env_move();
        turn = 1;
        assert (!cs1) || (claim1 && turn != 0));
        ...
    }
}

Figure 4.5: The sequential program $P_0$ after a few iterations of Algorithm 4.1.

Figure 4.6: The $\text{env\_move}$ function of $P_0$: initially (without highlighted lines); and after one refinement (with highlighted lines).

assertion) again in subsequent iterations. However, it is ensured that in all computations whose path is $\hat{\pi}$ (extended with the new assertion, and possibly other inner assertions added by Algorithm 4.1), the new assertion holds.

4.6 Generalizing an Environment Query

Given formulas $\alpha, \beta$ s.t. $\text{Reach}(t_E)(\alpha, \beta) = \text{FALSE}$, we wish to generalize them into $\alpha', \beta'$ s.t. $\text{Reach}(t_E)(\alpha', \beta') = \text{False}$ and $\alpha \Rightarrow \alpha'$ and $\beta \Rightarrow \beta'$. We start by representing $\alpha$ and $\beta$ in negation normal form (NNF) ([40]). Note that, if a formula $\gamma$ is in NNF, then for every subformula (except atomic negations) $\delta$ of $\gamma$ and for every formula $\delta'$ s.t $\delta \Rightarrow \delta'$, it holds that $\gamma \Rightarrow \gamma[\delta/\delta']$ (monotonicity). Then generalization is performed by iteratively choosing subformulas $\delta$ of $\alpha$ (or $\beta$), replacing $\delta$ by a generalization $\delta' \neq \delta$ s.t. $\delta \Rightarrow \delta'$, and computing $\text{Reach}(t_E)(\alpha, \beta)$. For example, we can attempt to generalize $\delta = \delta_1 \land \delta_2$ to $\delta_1$ or to $\delta_2$. If the result of $\text{Reach}(t_E)(\alpha, \beta)$ after generalization is still $\text{FALSE}$, the process continues. Otherwise, $\alpha$ (or $\beta$) is reverted and another generalization is attempted.
Chapter 5

Soundness and Progress of the Main Thread Analysis

In this chapter, we provide the correctness proof for Algorithm 4.1. First, we prove that the algorithm is sound. That is, (i) if it terminates, and returns that $P$ is safe, then $P$ is indeed safe, and (ii) if it returns that $P$ is unsafe, then $P$ has an initial violating computation. This is shown by Theorem 5.1.

Next, we prove that Algorithm 4.1 makes progress. Namely, it never stagnates. Instead, it continues learning new information through every iteration. This is proved by Lemmas 5.2.2 and 5.2.4. We also prove that for programs with a finite state space, termination is guaranteed. Theorem 5.2 provides this proof.

The proofs use one implicit assumption. Namely, they assume that there exists some mechanism for answering environment queries correctly. The mechanism we use is described and proved correct in Chapter 6.

5.1 Soundness

Our algorithm for verifying the concurrent program $P$ terminates when either (i) all the assertions in $P_M$ are proven safe (i.e., neither the original error nor all the new promises of error can be reached in $P_M$), in which case Algorithm 4.1 returns “Program is Safe”, or (ii) a violation of some assertion in $P_M$, which indicates either the original error or a promise of error, is reached without any $\text{env\_move}$ calls, in which case Algorithm 4.1 returns “Real Violation”. Theorem 5.1 below summarizes the soundness of the algorithm. In order to prove it we first need Lemmas 5.1.1, 5.1.3 and 5.1.6:

**Lemma 5.1.1 (Soundness of $\text{env\_move}$).** During the run of Algorithm 4.1, $\text{env\_move}$ always over-approximates the computations of $t_E$ in $P$ (see Definition 4.2.5).

**Proof.** The proof is by induction on the order of refinements of $P_M$, and specifically, the $\text{env\_move}$ function. We start by proving that the initial $\text{env\_move}$ satisfies the lemma. We then consider the two possible modifications of $P_M$: refining $\text{env\_move}$, and adding a new...
assert (¬ψ) command, outside of env.move. Though the latter, by itself, cannot affect any computation within env.move, it may introduce new variables (from $s_E$) to $P_M$, if such a variable $v$, not already in $\tilde{V}_M$, appears in $ψ$. This new variable affects the definition of Extend (Definition 4.2.1), and may also require an additional $v=havoc()$ within env.move. Thus, the proof consists of the following three stages:

1. The initial env.move satisfies the lemma.

2. If env.move satisfies the lemma and a variable $v$ is added to $P_M$, the new env.move function of the new program $P_M$ also satisfies the lemma.

3. If env.move satisfies the lemma and is augmented with if $(α′[W_copy/W])$ assume $(¬β′)$ commands, the new env.move function also satisfies the lemma.

Let $ρ$ be a computation of $t_E$ in $P$ from $s$ to $s'$ which satisfies the conditions of Definition 4.2.5.

**Base Case:** Initially, the env.move function only havocs all variables in $\tilde{V}_M$ that are written by $t_E$ (anywhere in the program). Let $v_1=havoc(), v_2=havoc(), \ldots, v_n=havoc()$ be the series of havocs inside env.move. Since a havoc command only sets a non-deterministic value to a single variable (and then moves to the next command), we can construct a computation in $P_M$, $\tilde{ρ} = ̄s_0 \rightarrow ̄s_1 \rightarrow ̄s_2 \cdots \rightarrow ̄s_n$ starting from $̄s = ̄s_0$ and ending at $̄s' = ̄s_n$ such that $̄s_i$ is obtained from $̄s_{i-1}$ by performing $v_i=havoc()$, for every $1 \leq i \leq n$.

Since a havoc command can assign any non-deterministic value, we can choose a computation in which $v_i=havoc()$ assigns $σ(̄s_i)(v_i)$ with $σ(̄s')(v_i)$ (or with $l_E(̄s')$ if $v_i = pc_E$). Since $v_i=havoc()$ does not change variables other than $v_i$, we get $σ(̄s_i)(v) = σ(̄s_{i-1})(v)$ for every $v \neq v_i$. This yields a computation from $̄s$ to $̄s'$, which is clearly a complete single execution of env.move. It is left to show that $̄s' \in Extend(̄s')$.

Since $s \in Extend(̄s), l_M(s) = Lab(l(̄s))$. Since cmdl($̄s$) is an env.move call, Lab(l($̄s$)) is defined as the label of the next command in $P_M$ after the env.move, which is exactly Lab(l($̄s'$)). Since $ρ$ is a computation of $t_E$ in $P$, it does not change pc$M$, and hence we get $l_M(s) = l_M(̄s')$. Joining all three equalities yields $l_M(̄s') = Lab(l(̄s'))$ as required.

Finally, let $v \in ̄V_M$. If $v$ is not written by $t_E$, then the value of $v$ in $s$ and $s'$ is identical. It also does not appear as one of the havocked variables, hence it’s value in $̄s$ and $̄s'$ is identical. Since $s \in Extend(̄s)$, we get the desired requirement about $v$ for $s'$ and $s'$ as well. If $v$ is written by $t_E$, then $v$ appears as one of the havocked variables. Its value is set exactly once to $σ(̄s')(v)$ (or to $l_E(̄s')$ if $v = pc_E$) by the $v=havoc()$ command, and it is unchanged by all other havoc commands. Hence, $σ(̄s')(v) = σ(̄s')(v)$ if $v \in V$ and $σ(̄s')(v) = l_E(̄s')$ if $v = pc_E$, which concludes the proof that $̄s' \in Extend(̄s')$.

**Induction Step:** For the next two parts, let $P_M$ denote the sequential program before refinement, and $P'_M$ the sequential program after refinement. Let env.move denote the env.move function of $P'_M$. Let Extend denote the mapping Extend for states of $P'_M$. 34
We need to show that given a state \( \hat{s}_r \) of \( P'_M \), s.t. \( \text{cmd}(\hat{s}_r) \) is an env.move call and \( s \in \text{Extend}_r(\hat{s}_r) \), there exists a computation \( \hat{\rho}_r = \hat{s}_r \rightarrow \cdots \rightarrow \hat{s}'_r \) in \( P'_M \), which is a complete single execution of env.move, s.t. \( s' \in \text{Extend}_r(\hat{s}'_r) \). By the induction hypothesis, we know that for every state \( \hat{s} \) of \( P_M \), s.t. \( \text{cmd}(\hat{s}) \) is an env.move call and \( s \in \text{Extend}(\hat{s}) \), there exists a computation \( \hat{\rho} = \hat{s} \rightarrow \cdots \rightarrow \hat{s}' \) in \( P_M \), which is a complete single execution of env.move, s.t. \( s' \in \text{Extend}(\hat{s}') \).

**Adding a New Variable to \( P_M \)** Assume first that \( P'_M \) was obtained from \( P_M \) after introducing a new variable \( v \). Given a state \( \hat{s}_r \) of \( P'_M \), as described above, let \( \hat{s} \) be the state of \( P_M \) obtained from \( \hat{s}_r \) by removing the value of \( v \) from \( \sigma(\hat{s}_r) \). Clearly, since \( s \in \text{Extend}_r(\hat{s}_r) \), then \( s \in \text{Extend}(\hat{s}) \), since removing a variable cannot compromise any of the conditions of Definition 4.2.1. Since \( l(\hat{s}) \) is also copied from \( \hat{s}_r \), then \( \text{cmd}(\hat{s}) \) is an env.move call. Thus, there exists a computation \( \hat{\rho} = \hat{s} \rightarrow \cdots \rightarrow \hat{s}_n \) in \( P_M \) from \( \hat{s} = s_0 \) to some \( \hat{s}' = \hat{s}_n \), as guaranteed by the induction hypothesis above. There are two cases to consider:

- **If \( v \) is not written by \( t_E \), then \( \text{env.move}_r = \text{env.move} \).** Since \( \rho \) is a computation of \( t_E \), it holds that \( \sigma(s')(v) = \sigma(s)(v) \) (or \( l_E(s') = l_E(s) \) if \( v = pCE \)). Let \( \hat{s}_0, \ldots, \hat{s}_n \) be a sequence of states of \( P'_M \), obtained from \( \hat{s}_0, \ldots, \hat{s}_n \) by adding the value of \( v \) to \( \sigma(\hat{s}_i) \) and setting it to \( \sigma(\hat{s}_i')(v) = \sigma(\hat{s}_i)(v) \), for \( 0 \leq i \leq n \). Note that in particular, it means that \( \hat{s}_0 = \hat{s}_r \).

Since all the commands in \( \hat{\rho} \) do not use \( v \) (which does not appear at all in \( P_M \)), and since \( \sigma(\hat{s}_{i-1}') = \sigma(\hat{s}_i') \) for every \( 1 \leq i \leq n \), we can use the exact same commands as in \( \hat{\rho} \) to form a computation \( \hat{\rho}_r = \hat{s}_0' \rightarrow \cdots \rightarrow \hat{s}_n' \). Since \( \text{env.move}_r = \text{env.move} \), this is also a complete single execution of env.move in \( P'_M \). It is left to show that \( s' \in \text{Extend}_r(\hat{s}_n') \).

Since \( s'_n \in \text{Extend}(\hat{s}_n) \), and \( s'_n \) only differs from \( \hat{s}_n \) by the appearance of \( v \) to \( \sigma(\hat{s}_n') \), we only need to check the condition in Definition 4.2.1 concerning \( v \). And indeed, since \( v \) is not changed during \( \hat{\rho}_r \) and since \( s \in \text{Extend}(\hat{s}_r) \), we get that \( \sigma(\hat{s}_n')(v) = \sigma(s')(v) = \sigma(s)(v) = \sigma(s')(v) \) (or \( l_E(s) = l_E(s') \)), if \( v = pCE \), as required.

- **If \( v \) is written by \( t_E \), then a new \( v = \text{havoc()} \) command is added at the beginning of env.move.** Let \( \hat{s}_0', \ldots, \hat{s}_n' \) be a sequence of states of \( P'_M \), obtained from \( \hat{s}_0, \ldots, \hat{s}_n \) by adding the value of \( v \) to \( \sigma(\hat{s}_i) \) and setting it to \( \sigma(\hat{s}_i')(v) = \sigma(s')(v) \) (or \( l_E(s') \), if \( v = pCE \)), for \( 0 \leq i \leq n \). We also change \( l(\hat{s}_0) \) to point to the second command in env.move, immediately after the new \( v = \text{havoc()} \). Note that \( \sigma(\hat{s}_0) \) and \( \sigma(\hat{s}_0') \) agree on all variables, except for, possibly, \( v \). Further, since \( \text{cmd}(\hat{s}_r) \) is an env.move call and \( l(\hat{s}_0') \) is immediately after the first command in env.move, there exists a transition in \( P'_M \) which moves from \( \hat{s}_r \) to \( \hat{s}_0' \), by using the \( v = \text{havoc()} \) command at the beginning of env.move, which sets the value of \( v \) to \( \sigma(\hat{s}_0')(v) = \sigma(s')(v) \) (or \( l_E(s') \) if \( v = pCE \)).

Let \( \hat{\rho}_r \) be a computation in \( P_M \), defined as follows: the computation starts by moving from \( \hat{s}_r \) to \( \hat{s}_0' \), as described above. The computation continues from \( \hat{s}_0' \), using the transitions \( \hat{s}_0' \rightarrow \cdots \rightarrow \hat{s}_n' \), and the same commands as in \( \hat{\rho} \). As before, this is a legal computation
since \( \hat{\rho} \) is a legal computation in \( P_M \), the commands of \( \hat{\rho} \) do not refer to \( v \) and all the states \( \hat{s}_0^r, \ldots, \hat{s}_n^r \) agree on \( \sigma(\hat{s}_i^r)(v) \). Since \( \hat{\rho} \) is a complete execution of \( \text{env}_r \), and \( \text{env}_r \) only differs from \( \text{env}_r \) by the addition of the initial \( v = \text{havoc}() \) command (which was used to reach from \( \hat{s}_r \) to \( \hat{s}_0^r \)) at the beginning of \( \text{env}_r \), then \( \hat{\rho}_r \) is a complete single execution of \( \text{env}_r \) from \( \hat{s}_r \) to \( \hat{s}_n^r \).

Again, it is left to show that \( s' \in \text{Extend}_r(\hat{s}_n^r) \). As before, we only need to check the condition for Definition 4.2.1 concerning \( v \), and indeed, \( \sigma(\hat{s}_n^r)(v) = \sigma(s')(v) \) (or \( = l_E(s') \), if \( v = \text{pc}_E \)), by construction.

**Refining env**

env is only refined after an environment query \( \text{Reach}(l_E)(\alpha', \beta') \) returned FALSE, as described in the beginning of this section. The refinement step adds the commands if (\( \alpha'[W_\text{copy}/W] \) assume(\( \neg\beta' \)) at the end of \( \text{env}_r \). Given a state \( \hat{s}_r \) of \( P_M \) as described above, let \( \hat{s} \) be the state of \( P_M \) identical to \( \hat{s}_r \). Clearly, since, \( s \in \text{Extend}_r(\hat{s}_r) \), then \( s \in \text{Extend}(\hat{s}) \). Since \( l(\hat{s}) \) is also copied from \( \hat{s}_r \), then \( \text{cmd}(\hat{s}) \) is an env call. Thus, there exists a computation \( \hat{\rho} = \hat{s}_0 \rightarrow \hat{s}_1 \rightarrow \cdots \rightarrow \hat{s}_n \in P_M \) from \( \hat{s} = \hat{s}_0 \) to some \( \hat{s}' = \hat{s}_n \), as guaranteed by the induction hypothesis above.

Let \( \hat{s}_0^r, \ldots, \hat{s}_n^r \) be a sequence of states of \( P_M^r \), identical to \( \hat{s}_0, \ldots, \hat{s}_n \), except for the value of \( l(\hat{s}_n^r) \), which points to the beginning of the new if (\( \alpha'[W_\text{copy}/W] \) command. Note that in particular, \( \hat{s}_r = \hat{s}_0^r \). Since \( P_M \) and \( P_M^r \) have the same variables, and since \( \text{env}_r \) and \( \text{env}_r \) share the same prefix until \( l(\hat{s}_n^r) \), the computation \( \hat{\rho}_r = \hat{s}_0^r \rightarrow \cdots \rightarrow \hat{s}_n^r \) is a valid computation in \( P_M^r \) from \( \hat{s}_r \), at the beginning of \( \text{env}_r \), to \( l(\hat{s}_n^r) \).

Since \( \text{Reach}(l_E)(\alpha', \beta') \) returned FALSE, by Definition 3.2.1, one of the conditions \( s \models \alpha' \) or \( s' \models \beta' \) cannot hold. That is, \( s \models \alpha' \) implies \( s' \models \neg\beta' \). Since \( s \in \text{Extend}(\hat{s}) \) and \( s' \in \text{Extend}(\hat{s}) \), it follows by Observation 4.2.3, that if \( \sigma(\hat{s}) = \alpha' \) then \( \sigma(\hat{s}') = \neg\beta' \). Since \( \sigma(\hat{s}_r) = \sigma(\hat{s}) \) and \( \sigma(\hat{s}_n^r) = \sigma(\hat{s}') \), then if \( \sigma(\hat{s}_r) = \alpha' \) then \( \sigma(\hat{s}_n^r) = \neg\beta' \).

The condition \( \alpha'[W_\text{copy}/W] \) uses auxiliary variables that are local to \( \text{env}_r \) and reflects the values of \( \hat{s}_r \)'s variables, when entering \( \text{env}_r \). The condition \( \neg\beta' \) refers to the current state variables. Consider a computation \( \hat{\rho}_r \), which is a concatenation of \( \hat{\rho}_r \) with the two new commands of \( \text{env}_r \).

If \( \sigma(\hat{s}_r) = \alpha' \), then \( \sigma(\hat{s}_n^r) = \neg\beta' \), which means that performing the two commands from \( \hat{s}_n^r \) will result in entering the true branch of the if command and then passing the assumption assume(\( \neg\beta' \)) (since the condition holds). If \( \sigma(\hat{s}_r) \neq \alpha' \), the next step from \( \hat{s}_n^r \) will be to skip the new if command (and the assume that follows). In both cases, \( \hat{\rho}_r \) reaches the end of \( \text{env}_r \), without changing \( \sigma(\hat{s}_n^r) \). Hence, we achieve a computation from \( \hat{s}_r \) to some new state \( \hat{s}_t' \) at the end of \( \text{env}_r \).

Finally, by the same arguments as in the base case, we get \( l_M(s') = l_M(s) = \text{Lab}(l(\hat{s}_r)) = \text{Lab}(l(\hat{s}_r')) \), and since \( \sigma(\hat{s}_r') = \sigma(\hat{s}_n^r) = \sigma(s') \) and \( s' \in \text{Extend}(\hat{s}') \), it also holds that \( s' \in \text{Extend}(\hat{s}_t') \), as required.

When \( \rho \) is an empty computation (i.e., \( s = s' \)) then the lemma receives the following form:
Corollary 5.1.2. If \( s \) is a reachable cut-point state of \( P \), then for any state \( \hat{s} \) of \( P_M \) such that \( s \in \text{Extend}(\hat{s}) \) and \( \text{cmd}(\hat{s}) \) is an \text{env.move} call, there exists a computation consisting of a complete single execution of \text{env.move} to a state \( \hat{s}' \) such that \( s \in \text{Extend}(\hat{s}') \).

Lemma 5.1.3 (Assertions are promises of error). Let \( \hat{l}_a \) be a label of \( P_M \) with \( \text{cmd}(\hat{l}_a) = \text{assert}(b) \) for some condition \( b \). Then \( (\text{Lab}(\hat{l}_a), \neg b) \) is a promise of error.

Proof. The proof is by induction on the order in which assertions are added to \( P_M \).

Base Case: Assume that \( \text{assert}(b) \) is an original assertion of \( t_M \) in \( P_M \). According to Definition 4.1.3, \( \text{Lab}(\hat{l}_a) = l_a \), where \( l_a \) is the label of \( t_M \), for which \( \text{cmd}(l_a) \) is the original \( \text{assert}(b) \). Let \( s \) be some state of \( P \) such that \( l_M(s) = l_a \) and \( s \models \neg b \). Let \( \rho \) be a zero length computation in \( P \), starting from \( s \) (and hence ending at \( s \)). Hence, by Definition 4.4.1, \( (l_a, \neg b) \) is a promise of \( (l_a, \neg b) \), and since \( \text{cmd}(l_a) = \text{assert}(b) \), then \( (l_a, \neg b) \) is a promise of error.

Induction Step: Assume that the lemma holds for all assertions in \( P_M \), and that a new assertion is now added. An assertion is only added at step (5) of Section 4.5, after an environment query \( \textit{Reach}_{\text{end}}(\alpha, \beta) \) returned \( \psi \neq \text{FALSE} \). The added assertion is of the form \( \text{assert}(\neg \psi) \) and it is added at a new label \( \hat{l}_\psi \), right before the \text{env.move} call for which the environment query was applied. Let \( l_\psi = \text{Lab}(\hat{l}_\psi) \) We would like to show that \( (l_\psi, \psi) \) is a promise of error.

Let \( s \) be a state of \( P \) such that \( l_M(s) = l_\psi \) and \( s \models \psi \). By Definition 3.2.1, there exists a computation \( \rho \) of \( t_E \) in \( P \) from \( s \) to some state \( s' \) such that \( s' \models \beta \). Since this computation is a computation of \( t_E \), \( pc_M \) is left unchanged at every stage in \( \rho \), hence \( l_M(s') = l_M(s) = l_\psi \). Therefore, \( (l_\psi, \psi) \) is a promise of \( (l_\psi, \beta) \).

An environment query is invoked only after a violating path, containing at least one \text{env.move} call, is found in \( P_M \). Let \( \hat{\pi}_{\text{end}} \) be the suffix of the found path, starting after the last \text{env.move}, as defined in step (1) of Section 4.5. The path ends with some label \( \hat{l}_n \) having \( \text{cmd}(\hat{l}_n) = \text{assert}(b) \) for some condition \( b \). By the induction hypothesis, \( (\text{Lab}(\hat{l}_n), \neg b) \) is a promise of error.

We now wish to show that \( (l_\psi, \beta) \) is a promise of \( (\text{Lab}(\hat{l}_n), \neg b) \). Let \( \hat{l}_k \) be the label in \( P_M \), of the last \text{env.move} call in the found path, and \( \hat{l}_{k+1} \) the label of the next command in \( P_M \) after the \text{env.move} (not inside the function). Since the newly added assertion at location \( \hat{l}_\psi \) is not an original command of \( t_M \) and it is added right before an \text{env.move} call, by Definition 4.1.3 \( l_\psi = \text{Lab}(\hat{l}_\psi) = \text{Lab}(\hat{l}_k) \). Using the same argument for the \text{env.move} call at \( \hat{l}_k \), we get that \( \text{Lab}(\hat{l}_{k+1}) = \text{Lab}(\hat{l}_k) \), and hence \( l_\psi = \text{Lab}(\hat{l}_{k+1}) \).

Let \( s_\beta \) be some state of \( P \) such that \( l_M(s_\beta) = l_\psi \) and \( s_\beta \models \beta \). Let \( \hat{s}_\beta \) be a state of \( P_M \) such that:

- \( l(\hat{s}_\beta) = \hat{l}_{k+1} \)
- \( \forall v \in (V \cap \overrightarrow{M}) \colon \sigma(\hat{s}_\beta)(v) = \sigma(s_\beta)(v) \)
- If \( pc_E \in \overrightarrow{M} \) then \( \sigma(\hat{s}_\beta)(pc_E) = l_E(s_\beta) \)
Clearly \( s_\beta \in \text{Extend}(\hat{s}_\beta) \), and by Observation 4.2.3, \( \hat{s}_\beta \models \beta \). According to step (1) in Section 4.5, and the definition of a weakest precondition, for every state \( \hat{s} \) of \( P_M \) s.t. \( \sigma(\hat{s}) \models \beta \), and specifically for \( \hat{s}_\beta \), there exists a computation in \( P_M \) that passes through \( \hat{s}_{\text{end}} \) (and therefore ends at \( \hat{t}_n \)) and reaches a state \( \hat{s}_{\text{end}} \) such that \( \sigma(\hat{s}_{\text{end}}) \models \neg b \) (and \( l(\hat{s}_{\text{end}}) = \hat{l}_n \)).

Since \( \hat{s}_{\text{end}} \) does not use \( \text{env}_{\text{moves}} \), it follows from Lemma 4.2.6, that there exists a computation in \( P \) from \( s_\beta \) to a state \( s_{\text{end}} \in \text{Extend}(\hat{s}_{\text{end}}) \). Again, by Observation 4.2.3, \( s_{\text{end}} \models \neg b \). Since \( s_{\text{end}} \in \text{Extend}(\hat{s}_{\text{end}}) \), it also holds that \( l_M(s_{\text{end}}) = \text{Lab}(l(\hat{s}_{\text{end}})) = \text{Lab}(\hat{l}_n) \).

We conclude that indeed \((l_\psi, \beta)\) is a promise of \((\text{Lab}(\hat{l}_n), \neg b)\). Since \((\text{Lab}(\hat{l}_n), \neg b)\) is a promise of error, by Lemma 4.4.2, \((l_\psi, \beta)\) is also a promise of error. Since \((l_\psi, \psi)\) is a promise of \((l_\psi, \beta)\), we can use Lemma 4.4.2 again, to conclude that \((l_\psi, \psi)\) is a promise of error. \( \square \)

Essentially, lemma 5.1.3 describes the manner in which \( P_M \) under-approximates the set of states leading to an error in \( P \). Each assertion provides an annotation for a set of states of \( P \), leading to an error. Any state of \( P_M \) that can reach and violate an assertion without using \( \text{env}_{\text{move}} \) calls, represents concrete states of \( P \) that can reach an error (possibly with the help of the environment).

The complementary direction, describes how \( P_M \) also over-approximates the computations of \( P \). Computations passing through safe states only (i.e., states from which an error in not reachable), are fully over-approximated. However, computations in \( P \) passing through states from which an error is reachable may be pruned in \( P_M \) by an assertion. To describe this formally, we need the following definition and lemma.

**Definition 5.1.4 (Computation Partitioning).** Let \( \rho \) be a computation in \( P \) from a state \( s \) to a state \( s' \). The **partition** of \( \rho \) is a series of computations \( r_1, \ldots, r_k \), called segments, such that

- \( s \) is the first state of \( r_1 \), \( s' \) is the last state of \( r_k \) and for every \( 1 \leq i \leq (k - 1) \), the last state of \( r_i \) is the first state of \( r_{i+1} \). Hence, the concatenation \( r_1 \cdot r_2 \cdots r_k \), connecting \( r_i \) with \( r_{i+1} \) using their overlapping state, yields \( \rho \).

- For \( 1 \leq i \leq k \), \( r_i \) is either a computation of \( t_M \) or a computation of \( t_E \).

- If \( r_i \) is a computation of \( t_M \), then it has no inner cut-point states (but there is no restriction for the first and last state of \( r_i \)).

- Each \( r_i \) is maximal. That is, if \( r_i \) is a computation of \( t_E \) then both of its neighbors \( r_{i-1} \) and \( r_{i+1} \) (if defined) must be computations of \( t_M \). If \( r_i, r_{i+1} \) are two adjacent computations of \( t_M \), then the state connecting them is a cut-point state.

If \( r_1, \ldots, r_k \), are the partition of a computation \( \rho \), then the **partitioning states** of \( \rho \), \( s_0, s_1, \ldots, s_k \) are the boundary states of the partition. That is, \( s_0 \) is the first state of \( \rho \) and for \( 1 \leq i \leq k \), \( s_i \) is the last state of \( r_i \).

Figure 5.1 illustrates the definition above.
Figure 5.1: The partition and partitioning states of a computation \( \rho \). Note that \( r_3 \) is allowed to have inner cut-point states, as it is a computation of \( t_E \).

**Observation 5.1.5.** Note that due to the maximality requirement, and the fact that initial locations are always considered as cut-points, all the partitioning states, except for, maybe, \( s_k \), are cut-point states.

**Lemma 5.1.6.** Let \( \rho_e = s \rightarrow \cdots \rightarrow s' \rightarrow \epsilon \) be an initial violating computation in \( P \), and let \( \rho \) be the prefix of \( \rho_e \) from \( s \) to \( s' \) (i.e., \( \text{cmd}(s') = \text{assert}(b) \) and \( \sigma(s') \not\equiv b \)). Let \( r_1, \ldots, r_k \) be the partition of \( \rho \), and \( s_0, s_1, \ldots, s_k \) the partitioning states of \( \rho \). Then there exists a sequence of states \( \hat{s}_0, \ldots, \hat{s}_j \) of \( P_M \) such that

1. \( j \leq k \)
2. \( \hat{s}_0 \) is reachable in \( P_M \) from an initial state.
3. For \( 1 \leq i \leq j \): \( \hat{s}_i \) is reachable in \( P_M \) from \( \hat{s}_{i-1} \)
4. For \( 0 \leq i \leq j \): \( s_i \in \text{Extend}(\hat{s}_i) \)
5. If \( i < j \), and \( r_{i+1} \) is a segment of \( t_M \), then \( \text{cmd}(\hat{s}_i) \) is an original command of \( t_M \)
6. If \( i < j \), and \( r_{i+1} \) is a segment of \( t_E \), then \( \text{cmd}(\hat{s}_i) \) is an \text{env.move} call
7. There exists a computation in \( P_M \) from \( \hat{s}_j \) to an error state

The main idea of this very technical lemma is that computations of \( P \) have a representation in \( P_M \). Given the partition of a computation \( \rho \) into segments, a segment of \( t_M \) will be represented by the exact same commands appearing in \( P_M \) (Lemma 4.2.7). Segments of \( t_E \) will be represented by the \text{env.move} function (Lemma 5.1.1). It might be the case that \( \rho \) contains two adjacent segments of \( t_M \), in which case their representative computations in \( P_M \) will have an \text{env.move} call between them. We will use Corollary 5.1.2 to “skip” the \text{env.move} while preserving the valuation of the connecting state.

However, the representative computation of \( \rho \) in \( P_M \) may encounter a promise of error, annotated by an assertion in \( P_M \). In this case, the representative computation will be pruned and move to an error state at an early stage. Thus, the representative computation will only represent a prefix of \( \rho \). Our goal is to show that when \( \rho_e \) is an initial violating computation
then the representative computation is always either pruned as described, or reaches the original assertion and violates it.

**Proof.** The proof is by construction. At each step, we construct a new state \( \hat{s}_i \) satisfying Items 3 and 4. Then, we show that either an error state is reachable from that state (Item 7), or it satisfies Items 5 and 6, and we can continue the construction of the next state \( \hat{s}_{i+1} \). However, while trying to construct \( \hat{s}_{i+1} \), we may discover that \( \hat{s}_i \) reaches an error before reaching the desired \( \hat{s}_{i+1} \). In that case, we can simply choose \( j = i \). I.e., \( \hat{s}_j \) will satisfy Item 7, as well as Items 5 and 6 (even without the condition \( i < j \))

We also show that \( \hat{s}_0 \) satisfies Item 2, and that if the construction reaches \( j = k \), then (Item 7) must hold, i.e., an error state is reachable (ensuring Item 1).

**Base Case - \( i = 0 \):** We start by constructing the first state \( \hat{s}_0 \). We first construct a state \( \hat{s} \) of \( P_M \) as follows, and show that \( \hat{s} \) is an initial state of \( P_M \):

- \( l(\hat{s}) = \bar{\text{init}} \) (initial label of \( P_M \))
- For every \( v \in V \cap \bar{V}_M \): \( \sigma(\hat{s})(v) = \sigma(s)(v) \)
- If \( p \in E \in \bar{V}_M \), then \( \sigma(\hat{s})(p \in E) = l_E(s) \)

Let \( \phi_{\text{init}} = \land_{v \in U} [v = \sigma_{\text{init}}(v)] \) be the initialization formula of \( P \), for some \( U \subseteq V \). Since \( s \) is initial in \( P \), it holds that \( \sigma(s)|_U = \sigma_{\text{init}} \), and \( l_E(s) = l_{\text{init}}^E \). By the definition of \( \hat{s} \), \( \sigma(\hat{s})|_{V \cap \bar{V}_M} = \sigma(s)|_{V \cap \bar{V}_M} \). Since \( U \subseteq V \), then clearly \( \sigma(\hat{s})|_{U \cap \bar{V}_M} = \sigma(s)|_{U \cap \bar{V}_M} = \sigma_{\text{init}}|_{U \cap \bar{V}_M} \). Further, if \( p \in E \in \bar{V}_M \), then \( \sigma(\hat{s})(p \in E) = l_E(s) = l_{\text{init}}^E \). Hence, \( \hat{s} \models \phi_{\text{init}} \), and is an initial state of \( P_M \).

By Definition 4.1.3, \( \text{Lab}(\hat{l}) \) is mapped to the label in \( t_M \), associated with the first original command of \( t_M \) in \( P_M \) (see Observation 4.1.2). That is, \( \text{Lab}(l(\hat{s})) = l_{\text{init}}^M = l_M(s) \). By Definition 4.2.1, \( s \in \text{Extend}(\hat{s}) \).

Since all initial labels are considered as cut-points, the code of \( P_M \) starts with an \texttt{env.move} call, before the first original command of \( t_M \), \( \text{cmd}(s) \), and possibly some new assertions (not originally from \( t_M \)) prior to that \texttt{env.move} call. If \( \sigma(\hat{s}) \) violates any of these new assertions, then \( \hat{s} \) leads to an error, hence we can pick \( \hat{s}_0 = \hat{s} \) and complete the proof with \( j = 0 \). Otherwise, the computation starting from \( \hat{s} \) reaches a state \( \hat{s}' \), after all new assertions, for which \( \text{cmd}(\hat{s}') \) is the \texttt{env.move} call. Since satisfied assertions do not change the valuation of a state, we have \( \sigma(\hat{s}) = \sigma(\hat{s}') \). Further, since none of the assertions is an original command of \( t_M \), we have \( \text{Lab}(l(\hat{s})) = \text{Lab}(l(\hat{s}')) \). Hence, since \( s \in \text{Extend}(\hat{s}) \), it also holds that \( s \in \text{Extend}(\hat{s}') \). \( s \) is initial in \( P \), and therefore reachable. Thus by Corollary 5.1.2, there exists a computation in \( P_M \) which uses the entire \texttt{env.move} function, from \( \hat{s}' \) to another state \( \hat{s}'' \), satisfying \( s \in \text{Extend}(\hat{s}'') \). \( \text{cmd}(\hat{s}'') \), the next command after the \texttt{env.move}, is the first command of \( t_M \), i.e. \( \text{cmd}(s) \).

We consider three possible cases:
1. If \( k = 0, \rho \) is an empty computation, and since \( \rho_0 \) is a violation, then \( \text{cmd}(s''') = \text{cmd}(s)\{\text{assert}(b) \) for some condition \( b \) s.t. \( \sigma(s) \neq b \). Since \( s \in \text{Extend}(s''') \), then \( \sigma(s''') \neq b \) as well. Hence we can choose \( j = 0 \), and \( 0 = s'' \). \( s'' \) is reachable from \( s \) (through \( (s') \) and the \text{env\_move} function), and reaches an error, as required (Item 7).

2. If \( k > 0 \) and \( r_1 \) is a segment of \( t_E \), we can pick \( \hat{s}_0 = s' \), as \( s' \) is reachable from \( s \), \( s \in \text{Extend}(s') \) and \( \text{cmd}(s') \) is an \text{env\_move} call (Item 6).

3. If \( k > 0 \) and \( r_1 \) is a segment of \( t_M \), we can choose \( \hat{s}_0 = s'' \), as \( s'' \) is reachable from \( s \), \( s \in \text{Extend}(s'') \) and \( \text{cmd}(s'') \) an original command of \( t_M \). (Item 5).

The latter two cases show that \( \hat{s}_0 \) satisfies Items 5 and 6, even in the case where \( j > 0 \). Thus, we can continue and construct the next state \( \hat{s}_1 \). Of course, if during this construction we learn that an error is reachable from \( \hat{s}_0 \), we can choose \( j = 0 \) and conclude the construction, without constructing \( \hat{s}_1 \).

**General Step:** Let \( 0 < i < k \), and let \( \hat{s}_{i-1} \) be the last constructed state. If we did not choose \( j = (i - 1) \) during the previous construction step, then \( \hat{s}_{i-1} \) satisfies Items (3 - 6).

If \( r_i \) is a segment of \( t_M \), \( \text{cmd}(\hat{s}_{i-1}) \) is an original command of \( t_M \) (Item 5). \( r_i \) has no inner cut-point states, but \( s_i \) itself is a cut-point state (see Observation 5.1.5). Using Lemma 4.2.7 for \( r_i \), there exists a computation in \( P_M \) from \( \hat{s}_{i-1} \) to some state \( s' \) that is either \( \epsilon \), or satisfies that \( s_i \in \text{Extend}(s') \) and \( \text{cmd}(s') \) is an \text{env\_move} call. If \( s' = \epsilon \), we can now choose \( j = (i - 1) \) and complete the construction. Otherwise, if \( r_{i+1} \) is a segment of \( t_E \), we choose \( \hat{s}_i = s' \), and it satisfies Items (3 - 6). Next, assume that \( r_{i+1} \) is a segment of \( t_M \). By Corollary 5.1.2, there exists a computation \( \hat{\rho} \) in \( P_M \) from \( s' \), consisting of a single complete execution of \text{env\_move}, to a state \( s'' \), also satisfying \( s_i \in \text{Extend}(s'') \). Since \( \hat{\rho} \) passes through the entire \text{env\_move} function, \( \text{cmd}(s'') \) is the next command after the \text{env\_move}, i.e., an original command of \( t_M \) (see Observation 4.1.1). We then choose \( \hat{s}_i = s'' \), and again it satisfies Items (3 - 6).

If, on the other hand, \( r_i \) is a segment of \( t_E \), then \( r_{i+1} \) must be a segment of \( t_M \), due to the maximality of the partition. Also, by Item 6, \( \text{cmd}(\hat{s}_{i-1}) \) is an \text{env\_move} call. Then by Lemma 5.1.1, there exists a computation in \( P_M \), consisting of a complete single execution of \text{env\_move} from \( \hat{s}_{i-1} \) to some state \( s' \) satisfying \( s_i \in \text{Extend}(s') \). Therefore, \( \text{cmd}(s') \) is the next command after \text{env\_move}, i.e., an original command of \( t_M \), and we can choose \( \hat{s}_i = s' \) and complete the construction of \( \hat{s}_i \).

**Last Step:** \( i = k \) We now show that if \( \hat{s}_k \) is constructed, \( \epsilon \) is definitely reachable from \( \hat{s}_k \) (Item 7) and we can complete the construction. Assume that in the previous construction step, we did not choose \( j = (k - 1) \), and \( \hat{s}_{k-1} \) was constructed satisfying Items (3 - 6).

Similar to the construction step, if \( r_k \) is a segment of \( t_M \), \( \text{cmd}(\hat{s}_{k-1}) \) is an original command of \( t_M \). Again, by Lemma 4.2.7, there exists a computation \( \hat{\rho} \) in \( P_M \) from \( \hat{s}_{k-1} \) to either \( \epsilon \) or to a state \( s' \) such that \( s_k \in \text{Extend}(s') \) and \( \text{cmd}(s') \) is either an \text{env\_move} call or an original
Theorem 5.1. If Algorithm 4.1 returns “Safe” then the concurrent program $P$ has no initial violating computation; If it returns “Real violation” then $P$ has an initial violating computation.

Intuitively, the first claim follows since env.move always over-approximates the reachable finite computations of $t_{E}$ (see Lemma 5.1.1). The second follows from the properties of an environment query (see Definition 3.2.1) and from the use of promises of errors (Definition 4.4.1). Formally, we use the lemmas above:

Proof. 1. The algorithm returns “Safe” only when $P_{M}$ is safe. However, if $P$ is not safe, by Lemma 5.1.6, there exists a sequence of states in $P_{M}$ such that the first is reachable from an initial state, every other state in the sequence is reachable from the previous, and the last state can reach $\epsilon$ in $P_{M}$. This yields an initial violating computation in $P_{M}$. Therefore, Algorithm 4.1 cannot return “Safe”.

2. The algorithm returns "Real violation” only when there exists an initial violating computation $\hat{\rho} = \hat{s} \rightarrow \cdots \rightarrow \hat{s}' \rightarrow \epsilon$ that does not use any env.move. Since the last step of the computation leads to $\epsilon$, $cmd(\hat{s}') = assert(b)$, for some condition $b$, such that $\sigma(\hat{s}') \models \neg b$. By Lemma 4.2.4, there exists an initial state $s$ of $P$ s.t. $s \in \text{Extend}(\hat{s})$. $\hat{s}' \neq \epsilon$, hence by Lemma 4.2.6, there exists a computation $\rho$ in $P$ from $s$ to some state $s' \in \text{Extend}(\hat{s})$, $s$ is initial in $P$, and therefore, $\rho$ is an initial computation in $P$.

Since, $s' \in \text{Extend}(\hat{s})$, it holds that $l_{M}(s') = Lab(l(\hat{s}'))$. Further, since $\sigma(\hat{s}') \models \neg b$, then $s' \models \neg b$ (see Observation 4.2.3). By Lemma 5.1.3, $( Lab(l(\hat{s}')) , \neg b )$ is a promise of error. Hence, there exists a computation $\rho'$ from $s'$ in $P$ which violates an assertion in $P$. The concatenation of $\rho$ with $\rho'$ yields an initial violating computation in $P$.

$\square$
5.2 Progress and Termination

While termination is not guaranteed for programs over infinite domains, the algorithm is ensured to make progress in the following sense. Each iteration either refines env_move (step (4) in Section 4.5), thus pruning a violating computation of PM, or generates new promises of errors at earlier stages along the violating path (step (5) in Section 4.5). Each refinement of env_move makes it more precise w.r.t. the real environment, and hence advances the algorithm in the direction of verifying the program, in case it is safe. When a new promise of error is introduced, it is sufficient to find a path reaching it, rather than a path reaching the original error. Hence, it advances the algorithm in the direction of finding a bug, in case the program is unsafe. With each iteration, we can search for paths with less env_move calls, until we encounter a violating path in PM with no env_move calls at all (or prove that no such violating path exists). Further, the addition of the new promise of error guarantees that even though the same violating path may recur, the new assertion restricts the computations that can be observed along this path.

In order to formally describe progress, and prove termination for finite state programs, we consider two approximation sets. The ERR set is a set of states used to under-approximate the set of states known to lead to a real violation of safety. We can deduce such states from promises of error in PM. This set will strictly increase with every addition of a new assertion to PM. The MOV set consists of pairs of states, that over-approximate the start and end state of all reachable computations of tE. We can deduce such states from the computations allowed by the env_move function. This set will strictly decrease with every refinement of the env_move function.

The next lemmas show that each iteration of the algorithm (except the last one) refines one of the sets (strictly increases ERR or strictly decreases MOV) and leaves the other unchanged. This would prove termination for the finite case, as described in Theorem 5.2, as both sets ERR and MOV are bounded.

Definition 5.2.1. We define the following approximation sets:

- ERR is the sets of all states s in P such that there exists a state ˆs of PM, satisfying (i) s ∈ Extend(ˆs) (ii) cmd(ˆs) is a newly added assertion assert(b) in PM, and (iii) s ⊨ ¬b
- MOV is the set of all pairs (s, s′), where s, s′ are states of P such that there exist states ˆs, ˆs′ of PM such that s ∈ Extend(ˆs), s′ ∈ Extend(ˆs′), and there exists a complete single execution of env_move from ˆs to ˆs′ (see Definition 4.2.5).

Lemma 5.2.2 (Progress when Reach(tE)(α, β) = FALSE). Let ERR and MOV be the sets as in Definition 5.2.1 at the beginning of a new iteration of Algorithm 4.1, in which Reach(tE)(α, β) (in line 12) returned FALSE, and let ERRr and MOVr be the corresponding sets at the end of the iteration. Then ERRr = ERR and MOVr ⊆ MOV

Proof. If Reach(tE)(α, β) returns FALSE, then Algorithm 4.1 refines env_move, and concludes the iteration. Since this does not change the code of PM outside the env_move function (and
specifically does not add, remove or change any exiting assertion), it is clear that \( \text{ERR}_r = \text{ERR}_f \).

As for MOV, the refinement step augments \( \text{env}\_\text{move} \) with if \( (\alpha'[W\_\text{copy}/W]) \) assume \( \neg \beta' \) commands at the end of the function. Let \( P'_M \) be the program obtained from \( P_M \) following this refinement step, and let \( \text{env}\_\text{move}_r \) be the \( \text{env}\_\text{move} \) function of \( P'_M \).

We first show that \( \text{MOV}_r \subseteq \text{MOV} \). Let \((s, s') \in \text{MOV}_r \) and let \( \hat{s}_r, \hat{s}'_r \) be the corresponding states, and \( \hat{\rho}_r \) be the computation from \( \hat{s}_r \) to \( \hat{s}'_r \) as described in Definition 5.2.1. \( \hat{\rho}_r \) is a complete execution of \( \text{env}\_\text{move}_r \) (who only has a single exit point, at its end), hence it must have a prefix \( \beta'_r \) consisting of the entire \( \text{env}\_\text{move}_r \) function, except the newly added commands. Let \( \hat{s}'''_r \) be the last state of \( \hat{\rho}_r \). Consider a computation \( \hat{\rho} \) in \( P_M \) starting from a state \( \hat{s} \), identical to \( \hat{s}_r \), which uses the same sequence of actions (same commands and chooses the same value for havoc commands) as in \( \hat{\rho}_r \). Since the initial states are identical, such a computation will follow the same path and reach a state \( \hat{s}' \) with \( \sigma(\hat{s}') = \sigma(\hat{s}'''_r) \). The only difference between \( \hat{s}' \) and \( \hat{s}'''_r \) is the label. While \( l(\hat{s}'''_r) \) points to the newly added if command, \( l(\hat{s}') \) points immediately after the \( \text{env}\_\text{move} \) function (same as \( \hat{s}'_r \)), and hence satisfies \( l(\hat{s}') = l(\hat{s}'_r) \). Since the suffix of \( \hat{\rho}_r \) reached \( \hat{s}'_r \) from \( \hat{s}''_r \) only through if and assume commands, it also holds that \( \sigma(\hat{s}_r') = \sigma(\hat{s}'_r) \), and hence \( \sigma(\hat{s}') = \sigma(\hat{s}'_r) \), which together with \( l(\hat{s}') = l(\hat{s}'_r) \) leads to \( \hat{s}' \) = \( \hat{s}'_r \).

Therefore, \( \hat{s}' \in \text{Extend}(\hat{s}') \), and we know that \( s \in \text{Extend}(\hat{s}) \) (because \( \hat{s} = \hat{s}_r \)), and that \( \hat{\rho} \) is a complete single execution of \( \text{env}\_\text{move} \) from \( \hat{s} \) to \( \hat{s}' \). By Definition 5.2.1, this proves \((s, s') \in \text{MOV} \).

Finally, we wish to show that \( \text{MOV}_r \neq \text{MOV} \). The iteration can only reach line 12 (where \( \text{Reach}(t_E)(\alpha, \beta) \) is called), if an initial violating computation, \( \hat{\rho} \), was found in \( P_M \), containing at least one \( \text{env}\_\text{move} \) call. Let \( \hat{s}_\alpha \) be the state of that computation before the last \( \text{env}\_\text{move} \) call, and let \( \hat{s}_\beta \) be the state of the same computation immediately after the \( \text{env}\_\text{move} \). By the definition of a postcondition and a weakest precondition, and of \( \alpha \) and \( \beta \) in Algorithm 4.1, it holds that \( \sigma(\hat{s}_\alpha) \vdash \alpha \) and \( \sigma(\hat{s}_\beta) \vdash \beta \).

Clearly, the sub-computation of \( \hat{\rho} \) from \( \hat{s}_\alpha \) to \( \hat{s}_\beta \) is a complete single execution of \( \text{env}\_\text{move} \). According to Observation 4.2.2, there exist some \( s \in \text{Extend}(\hat{s}_\alpha) \) and \( s' \in \text{Extend}(\hat{s}_\beta) \), which means that \((s, s') \in \text{MOV} \). We would like to show that \((s, s') \notin \text{MOV}_r \).

Let \( \alpha' \), \( \beta' \) be the formulas used for refinement. The chosen formulas must satisfy \( \alpha \Rightarrow \alpha' \), \( \beta \Rightarrow \beta' \). Since \( s \in \text{Extend}(\hat{s}_\alpha) \) and \( \sigma(\hat{s}_\alpha) \vdash \alpha \), then \( s \vdash \alpha \) (see Observation 4.2.3), and since \( \alpha \Rightarrow \alpha' \): \( s \vdash \alpha' \). Similarly, \( s' \vdash \beta' \).

Assume to the contrary that \((s, s') \in \text{MOV}_r \). That is, there exist some \( \hat{s}_r, \hat{s}'_r \) of \( P'_M \), such that \( s \in \text{Extend}(\hat{s}_r) \) and \( s' \in \text{Extend}(\hat{s}'_r) \) and a computation \( \hat{\rho}_r \) from \( \hat{s}_r \) to \( \hat{s}'_r \), consisting of a complete single execution of \( \text{env}\_\text{move}_r \). In particular, \( \hat{\rho}_r \) reaches (and passes) the newly added if \( (\alpha'[W\_\text{copy}/W]) \) assume \( \neg \beta' \) at the end of the \( \text{env}\_\text{move} \) function. Since \( s \in \text{Extend}(\hat{s}_r) \) and \( s' \in \text{Extend}(\hat{s}'_r) \), by Observation 4.2.3, we get that \( \sigma(\hat{s}_r) \vdash \alpha' \) and \( \sigma(\hat{s}'_r) \vdash \beta' \).

Using auxiliary variables to hold old values, the condition if \( (\alpha'[W\_\text{copy}/W]) \) refers to the

\(^1\)Note that the definition of \( \text{ERR} \) uses general (and not necessarily reachable) states of \( P_M \). Though the set of reachable states in \( \text{ERR} \) may decrease due to the pruning of computations using \( \text{env}\_\text{move} \), the set of general states in \( \text{ERR} \) does not change
values of the variables of the first state that entered the \texttt{env\_move}\_r function, i.e., \(s_r\). Since \(s_r \models \alpha\), the computation enters the true branch of the if statement and reaches the \texttt{assume}(\neg \beta') at some state \(s_r'\). Since the computation reaches the end of the function, it must hold that \(\sigma(s_r') \models \neg \beta'\). The next (and final) state in \(\hat{\rho}_r\) is \(s_r''\). However, passed \texttt{assume} commands do not change variables, hence \(\sigma(s_r'') = \sigma(s_r')\), which contradicts \(\sigma(s_r') \models \beta'\).

\(\square\)

When \(\text{Reach}_{(t_E)}(\alpha, \beta)\) returns \(\psi \neq \text{FALSE}\), the new formula might contain new variables from \(V\), which were not previously in \(\hat{V}_M\). These variables should then be added to \(\hat{V}_M\), and if they are written by \(t_E\), they should also be havocked inside \texttt{env\_move}. We first show that adding such variables does not change our approximation sets, and proceed to prove the progress lemma for the case where \(\text{Reach}_{(t_E)}(\alpha, \beta) \neq \text{FALSE}\).

**Lemma 5.2.3 (No regress when adding new variable).** Let \(\text{ERR}\) and \(\text{MOV}\) be the approximation sets as in Definition 5.2.1 for \(P_M\), and assume a new variable, \(v\), from \(t_E\) is added to \(\hat{V}_M\), creating a new program \(P_M'\) with variables \(\hat{V}_M'\). Let \(\text{ERR}_r\) and \(\text{MOV}_r\) be the corresponding sets in \(P_M'\). Then \(\text{ERR}_r = \text{ERR}\) and \(\text{MOV}_r = \text{MOV}\).

**Proof.** Adding \(v\) does not change any of the existing assertions, and does not add new assertions, hence it is clear that \(\text{ERR}_r = \text{ERR}\). Let \texttt{env\_move}\_r denote the \texttt{env\_move} function of \(P_M'\). If \(v\) is not written by \(t_E\), no modification is made to the code of \(P_M\), except adding a declaration for \(v\), i.e., adding it to \(\hat{V}_M\), and maybe initializing \(v\). In particular \texttt{env\_move}\_r=\texttt{env\_move}\_r. Hence, \(\text{MOV}_r = \text{MOV}\) in this case as well.

If \(v\) is written by \(t_E\), a new \texttt{havoc} statement is added for this variable inside \texttt{env\_move}\_r, and a new local auxiliary variable is added to \texttt{env\_move}\_r to hold \(v\)’s initial value.

We first show that \(\text{MOV} \subseteq \text{MOV}_r\). Let \((s, s') \in \text{MOV}\), and let \(\hat{s}, \hat{s}'\) be the corresponding states, and \(\hat{\rho}\) be the computation from \(\hat{s}\) to \(\hat{s}'\) as described in Definition 5.2.1. Let \(\hat{s}_r\) be a state of \(P_M'\) identical to \(\hat{s}\) w.r.t all mutual variable (\(\hat{V}_M\)) and with location \(l(\hat{s}_r) = l(\hat{s})\), i.e., pointing to the same \texttt{env\_move} call. For the variable \(v\), we define \(\sigma(\hat{s}_r)(v) = \sigma(s)(v)\) (or \(\sigma(\hat{s}_r)(v) = l_E(s)\) if \(v = \text{pc}\)). Clearly, since \(s \in \text{Extend}(\hat{s})\), we also get \(s \in \text{Extend}(\hat{s}_r)\).

Consider the following two-step computation \(\hat{\rho}_r\) in \(P_M'\) starting from \(\hat{s}_r\). The first step of the computation would be to assign a copy of \(v\) to a new local auxiliary variable \(v\_\text{COPY}\). The second step will be a \(v = \text{havoc}\()\) command, setting the value of \(v\) to \(\sigma(s')(v)\) (or \(l_E(s')\) if \(v = \text{pc}\)). Let \(s_r''\) be the end state of \(\hat{\rho}_r\). As only \(v\) and \(v\_\text{COPY}\) were changed by \(\hat{\rho}_r\), \(\sigma(s_r'')(\hat{V}_M') = \sigma(s_r)\) for \(\hat{V}_M\).

The rest of the commands inside \texttt{env\_move}\_r (i.e., all commands except for those used by \(\hat{\rho}_r\)) are exactly the same commands as in the entire \texttt{env\_move} function of \(P_M\). Further, \(\sigma(s_r'')|_{\hat{V}_M'} = \sigma(s)\), and the computation \(\hat{\rho}\) only uses variables from \(\hat{V}_M\). Hence, there exists a computation \(\hat{\rho}'\) from \(s_r''\) to some state \(\hat{s}'\) which uses the exact same actions as \(\hat{\rho}\) (same commands, and chooses the same values for \texttt{havocs}), and reaches a state, \(s_r''\), with the same values for \(\hat{V}_M\) as \(s\) (i.e., \(\sigma(s_r'')|_{\hat{V}_M'} = \sigma(s')\)), and the same location \(l(s_r'') = l(s')\), at the end of the \texttt{env\_move} function. Further, since none of the commands in \(\hat{\rho}\) changes \(v\), it must hold that

45
\[
\sigma(s'_r)(v) = \sigma(s''_r)(v) = \sigma(s')(v) \text{ (or } t_E(s') \text{ if } v = p_{CE}). \n\]
Therefore, since \( s' \in \text{Extend}(s') \) then it also holds that \( s' \in \text{Extend}(s'_r) \). Let \( \hat{\rho}_r \) be the concatenation \( \hat{\rho}_r = \hat{\rho}'_r \cdot \hat{\rho}''_r \). Then \( \hat{\rho}_r \) is a complete single execution of \( \text{env}_\text{move}_r \), which means that \((s, s') \in \text{MOV}_r \).

The proof of the converse, \( \text{MOV} \supseteq \text{MOV}_r \), is similar, and is only briefly sketched. For this direction, we start with a computation \( \hat{\rho}_r = \hat{s}_r \rightarrow \cdots \rightarrow \hat{s}'_r \) of \( \text{env}_\text{move}_r \) and omit the two first new commands. We obtain a computation \( \hat{\rho}'_r \) which does not use \( v \) and passes through the same commands as in \( \text{env}_\text{move} \). Then we need to show that it corresponds to a computation \( \hat{\rho} \) in \( P_M \) from a state \( \hat{s} \) “matching” \( \hat{s}_r \) to a state \( \hat{s}' \) “matching” \( \hat{s}'_r \). Since \( v \) does not appear in \( \hat{V}_M \), its value does not influence the definition of \( \text{Extend} \) w.r.t. \( P_M \), and hence we will have \( s \in \text{Extend}(\hat{s}) \) and \( s' \in \text{Extend}(\hat{s}') \). \( \square \)

**Lemma 5.2.4** (Progress when \( \text{Reach}(t_E)(\alpha, \beta) \neq \text{FALSE} \)). Let \( \text{ERR} \) and \( \text{MOV} \) be the approximation sets as in Definition 5.2.1 at the beginning of a new iteration of Algorithm 4.1, in which \( \text{Reach}(t_E)(\alpha, \beta) \) (in line 12) returned \( \psi \neq \text{FALSE} \), and let \( \text{ERR}_r \) and \( \text{MOV}_r \) be the corresponding sets at the end of the iteration. Then \( \text{ERR}_r \supseteq \text{ERR} \) and \( \text{MOV}_r = \text{MOV} \).

**Proof.** If \( \text{Reach}(t_E)(\alpha, \beta) \) returns \( \psi \neq \text{FALSE} \), then Algorithm 4.1 adds a new \( \text{assert}(\neg \psi) \) to the code of \( P_M \), and concludes the iteration. We can assume w.l.g. that all the variables in \( \psi \) are already in \( \hat{V}_M \), since if not, by Lemma 5.2.3 they can be added without changing the sets \( \text{ERR} \) and \( \text{MOV} \).

Adding \( \text{assert}(\neg \psi) \) does not change the \( \text{env}_\text{move} \) function, does not change any \( \text{env}_\text{move} \) call, and does not change original commands of \( t_M \). If \((s, s') \in \text{MOV} \), then the corresponding states \((\hat{s}, \hat{s}') \) from Definition 5.2.1 are such that \( \text{cmd}(\hat{s}) \) is an \( \text{env}_\text{move} \) call, and \( \text{cmd}(\hat{s}') \) is a command appearing immediately after an \( \text{env}_\text{move} \). According to Observation 4.1.1, it must be an original command of \( t_M \). Hence, all computations from such \( \hat{s} \) to such \( \hat{s}' \) do not pass through newly added assertions, and hence are not affected by the addition of another new assertion. Therefore, \( \text{MOV}_r = \text{MOV} \).\(^2\)

For \( \text{ERR} \), we first show that \( \text{ERR}_r \supseteq \text{ERR} \). Let \( s \in \text{ERR} \) and let \( \hat{s} \) be the corresponding state from Definition 5.2.1, such that \( s \in \text{Extend}(\hat{s}) \), \( \text{cmd}(\hat{s}) \) is a new assertion \( \text{assert}(b) \) in \( P_M \) and \( s \models \neg b \). The existence of \( \hat{s} \) and the relevant \( \text{assert}(b) \), is not affected by adding \( \text{assert}(\neg \psi) \), hence \( s \in \text{ERR}_r \) as well.

Finally, we need to show that \( \text{ERR}_r \neq \text{ERR} \). Let \( P_M^r \) be the program obtained from \( P_M \) after adding the new assertion. The command \( \text{assert}(\neg \psi) \) is added directly before an \( \text{env}_\text{move} \) call. Let \( \text{assert}(c_1), \ldots, \text{assert}(c_m) \) be the (possibly empty) sequence of assertions preceding this \( \text{env}_\text{move} \) call in \( P_M \), and let \( \gamma = \bigwedge_{1 \leq i \leq m} c_i \). According to step (2) of Section 4.5, \( \alpha \Rightarrow \gamma \), and by the last property of Definition 3.2.1, \( \alpha \land \psi \neq \text{FALSE} \).

Hence, there exists a valuation \( \hat{\sigma}_r \) of \( \hat{V}_M \) such that \( \hat{\sigma}_r \models (\psi \land \alpha) \), and since \( \alpha \Rightarrow \gamma \), it holds that \( \hat{\sigma}_r \models (\psi \land \gamma) \). Let \( \hat{s}_r = \ldots (l'_r, \hat{\sigma}_r) \), where \( l'_r \) is the new label associated with the newly added \( \text{assert}(\neg \psi) \) in \( P_M^r \). According to Observation 4.2.2, there exists some \( s \in \text{Extend}(\hat{s}_r) \), and by Observation 4.2.3, \( s \models \psi \land \gamma \). By Definition 5.2.1, we get that \( s \in \text{ERR}_r \).

\(^2\)As in lemma 5.2.2, this is correct since we do not restrict ourselves to reachable computations.
We would like to show that $s \not\in \text{ERR}$. Assume to the contrary that $s \in \text{ERR}$, and let $\hat{s}$ be a state of $P_M$ such that $s \in \text{Extend}(\hat{s})$, $\text{cmd}(\hat{s})$ is some newly added $\text{assert}(b)$ in $P_M$, and $s \models \neg b$.

Since $s \in \text{Extend}(\hat{s}_r)$ and $s \in \text{Extend}(\hat{s})$, it holds that $\text{Lab}(l(\hat{s})) = l_M(s) = \text{Lab}(l(\hat{s}_r))$. $l(\hat{s}_r) = \hat{l}'_r$ is the new label associated with the new $\text{assert}(\neg \psi)$ in $P_M$, right before an $\text{env}_\text{move}$ call. Hence, $\text{Lab}(\hat{l}'_r)$ is mapped to the location in $t_M$ of the first command after that $\text{env}_\text{move}$ (which is an original command of $t_M$ by Observation 4.1.1). Let $l' = \text{Lab}(\hat{l}'_r)$. The only new assertions in $P_M$ whose location can also be mapped to $l'$ are $\text{assert}(c_1), \ldots, \text{assert}(c_m)$.

Hence, $\text{cmd}(\hat{s}) = \text{assert}(c_i)$ for some $1 \leq i \leq m$, which means that $s \models \neg c_i$. But, $\gamma = \bigwedge_{1 \leq i \leq m} c_i$, and this contradicts $s \models \gamma$.

\[ \square \]

**Theorem 5.2.** If $P$ is a finite-state program, then Algorithm 4.1 terminates.

**Proof.** Let $N$ be the number of states in $P$. Given $P_M$, we define the function $f(P_M) = (N - |\text{ERR}|) + |\text{MOV}|$. The number of states in ERR is bounded by $N$, the total number of states of $P$, hence $f(P_M)$ is always non-negative. If $\text{Reach}(t_E)(\alpha, \beta)$ returns $\text{FALSE}$ at some iteration, then by lemma 5.2.2, $|\text{ERR}|$ is left unchanged, while $|\text{MOV}|$ decreases, causing $f(P_M)$ to decrease. If $\text{Reach}(t_E)(\alpha, \beta)$ returns $\psi \neq \text{FALSE}$, then by lemma 5.2.4, $|\text{MOV}|$ is left unchanged, and $|\text{ERR}|$ increases, causing $f(P_M)$ to decrease again.

Since $f(P_M)$ is bounded from below by 0, it can only decrease a finite number of times. Hence, Algorithm 4.1 can only reach the call to $\text{Reach}(t_E)(\alpha, \beta)$ finitely often. The only cases in which Algorithm 4.1 does not reach the environment query, is either when there is no initial violating computation in $P_M$ - in which case the algorithm terminates, proving the program safe, or when there exists an initial violating computation in $P_M$ without any $\text{env}_\text{move}$ calls - in which case the algorithm terminates, proving the program unsafe. Therefore, termination is always guaranteed.

\[ \square \]

**Note.** The proof of termination here relies on the successful termination of all sequential model checking calls performed by our algorithm. Clearly, if our sequential model checker does not terminate at some stage during the algorithm, our algorithm will continue waiting for its result and will not terminate as well.
Chapter 6

Answering Environment Queries

In this chapter, we complete the description of our approach for verifying concurrent programs that consist of two threads, by presenting a technique for answering environment queries (Definition 3.2.1). Intuitively, the purpose of an environment query $\text{Reach}(t_E)(\alpha, \beta)$ is to check whether there exists a reachable computation $\rho$ of $t_E$ in $P$ from a state $s \models \alpha$ to a state $s' \models \beta$. This computation may involve any finite number of steps of $t_E$, executed without interference of $t_M$. Section 6.1 describes how we can generate a sequential program and use a sequential model checker to answer this question, while Section 6.2 formally proves that this construction indeed satisfies the properties of Definition 3.2.1.

Note. A careful examination of the exact properties of Definition 3.2.1 reveals that an environment query does not completely determine the existence of a reachable computation of $t_E$ from $\alpha$ to $\beta$. Although, $\psi = \text{FALSE}$ guarantees that there are no such computations, there might be no such computation when $\psi \neq \text{FALSE}$ as well, if all the states satisfying $\psi \land \alpha$ are unreachable.

6.1 Sequential Program for Answering Environment Queries

If $\alpha \land \beta \neq \text{FALSE}$, we simply return $\beta$, which is a valid answer, as it represents a computation of length zero. Otherwise, we wish to apply a sequential model checker on $t_E$ in order to reveal such computations, or conclude there are none. However, the computation $\rho$ may not be initial. That is, $\rho$ may start from an arbitrary label $l$ of $t_E$ with non-initial values to the variables, while our sequential model checker can only search for violating paths starting from an initial state. Hence we construct a modified sequential program $P_E$, based on the code of $t_E$, which also represents (over-approximates) non-initial, but reachable, computations $\rho$ of $t_E$ in $P$.

We observe that a reachable computation $\rho$ is the continuation of a computation $\rho'$ which is initial in $P$, but is not restricted to $t_E$ alone. Before starting $\rho$, $t_M$ was allowed to run and set different values to its variables (possibly several times) and this could have also affected the control flow of $t_E$ before starting $\rho$. Therefore, we add in $P_E$ calls to a new function, $\text{try.start}$, which models the runs of $t_M$ until the start of $\rho$, by assigning non-deterministic values to the variables written by $t_M$. The calls to $\text{try.start}$ are added in all cut-points of $t_E$. 

49
The `try_start` Function  The `try_start` function is responsible for non-deterministically\(^1\) setting the start point of \(\rho\), where context switches to \(t_M\) are no longer allowed. It does so by setting a new `start` variable to `TRUE` (provided that its value is not yet `TRUE`). We refer to the latter call as the activation `try_start`. `start` is initialized to `FALSE` in \(P_E\). As long as `start` remains `FALSE` (i.e., prior to the activation call), `try_start` havoc all the variables in \(P_E\), written by \(t_M\). When `start` is set to `TRUE`, we add an `assume(\alpha)` command after the havoc commands, as this is the state chosen to start the computation \(\rho\). Whenever `start` is already `TRUE`, `try_start` exits without performing any havoc commands, ensuring that \(\rho\) indeed only uses transitions of `true-branch` is never taken.

Recall that \(\alpha\) and \(\beta\) may also refer to `pc_E`. To address this, we explicitly add `pc_E` to \(\tilde{V}_E\) as an auxiliary variable (which is different than the implicit `pc` variable of \(P_E\)). We need to make sure that whenever \(\alpha\) or \(\beta\) are evaluated, the value of `pc_E` corresponds to the label of the next original command of \(t_E\) to be executed (similar to the mapping in Definition 4.1.3). To this end, `try_start` receives the original location (in \(t_E\)) in which it is called as a parameter, and updates the explicit `pc_E` variable.

We also add assertions of the form `assert(!\text{start} \lor \neg\beta)`, after every call to `try_start` in \(P_E\). Hence, a violating path, if found, is such that it reached `\text{start} \land \beta`. That is, it captures a computation in which `start` was set to `TRUE` at some point (in which \(\alpha\) was satisfied) and reached \(\beta\). As before, \(\beta\) may refer to `pc_E`, but the new assertions appear exactly after the `try_start` calls, which update `pc_E`.

Returning Result  If a violating path is not found, we conclude that a computation as above does not exist and hence return `\text{Reach}(t_E)(\alpha, \beta) = \text{FALSE}`. If a violating path \(\hat{\pi} = \hat{l}_0, \ldots, \hat{l}_{n+1}\) is found (with \(\hat{l}_{n+1} = l\)), let \(\hat{l}_k\) be the location of the first command after the activation `try_start(l)` for some \(0 \leq k \leq n\), and \(l\) the program location that was passed to the activation `try_start`. Further, let \(\hat{\pi}_{\text{end}}\) be the sub-path of \(\hat{\pi}\) from \(\hat{l}_k\) to \(\hat{l}_{n}\). We compute the weakest precondition of \(\beta\) w.r.t.

\(\text{the path } \hat{\pi}_{\text{end}}\). When `\text{start} = \text{TRUE}`\(^2\), the `try_start` function does nothing except updating the explicit `pc_E` variable of \(P_E\). Hence, a computation passing through \(\hat{\pi}_{\text{end}}\) is essentially a computation of \(t_E\) in \(P\).

However, \(\hat{\pi}_{\text{end}}\) is nevertheless a path of \(P_E\), and as such, its commands may refer to the auxiliary variable `start`\(^2\). We wish to obtain a result \(\psi\) over \(V \cup \{pc_E\}\) only. Since we know that after the activation `try_start`, `start` always has the value `TRUE`, we can replace `start` with `TRUE` in the weakest precondition formula. Hence, the chosen result would be \(\psi = (pc_E = l) \land wp(\hat{\pi}_{\text{end}}, \beta)[\text{start}/\text{TRUE}]\).

The computed \(\psi\) satisfies the desired requirement: For every state \(s\) of \(P\) with \(l_E(s) = l\) s.t. \(s \models wp(\hat{\pi}_{\text{end}}, \beta)\), there exists a computation \(\rho\) of \(t_E\) starting from \(s\) which follows a path corresponding to \(\hat{\pi}_{\text{end}}\) in \(P\) and will reach a state \(s'\) satisfying \(\beta\). This is described more formally

\(^1\)A non-deterministic choice is modeled by using an additional variable, havocking it and then checking its value.

\(^2\)In particular, the `if (!\text{start})` condition itself inside `try_start` will be part of the path, although its true-branch is never taken
is Section 6.2.

Note. As mentioned in the beginning of this chapter, it is not guaranteed that $\rho$ is reachable, as in the prefix we only used an abstraction of $t_M$. However, it satisfies the requirements of Definition 3.2.1.

Example 6.1.1. Figure 6.1 describes the code of $P_1$, used to answer the environment query $\text{Reach}_{(t_E)}(\alpha, \beta)$ for $\alpha = \neg cs1 \land \text{claim0} \land (\neg \text{claim1} \lor \text{turn} = 0)$ and $\beta = cs1$. The $\text{try\_start}$ function is called at every computed cut-point, followed by $\text{assert} (\neg \text{start} \lor \neg \beta)$, i.e. $\text{assert} (!\text{start} \mid \mid !cs1)$. However, since $cs1$ is only written by $t_1$, we can use static analysis to learn $cs1$’s value at different locations. In our case $cs1$ is false everywhere except between lines 12 and 15, hence the assertion $\text{assert} (!\text{start} \mid \mid !cs1)$ immediately holds and can be dropped.

The parameters passed to $\text{try\_start}$ are the program locations of the next command to be executed, as can be seen in Figure 4.2. For as long as $\text{start}$ is false, every call to $\text{try\_start}$ will havoc $\text{claim0}$ and $\text{turn}$, as these are the variables written by $t_0$. When the $TRUE$ branch of the $\text{if}(\ast)$ is chosen, we set $\text{start}$ to true and add assumptions for our precondition $\neg cs1 \land \text{claim0} \land (\neg \text{claim1} \lor \text{turn} = 0)$.

If the activation $\text{try\_start}$ is in lines 2, 5, 16 or 18, $t_1$ will be stuck in the busy wait loop, since we assume $\text{claim0}$, whose value does not change, and $\text{turn}$ is set to 0 before the loop in line 6. If the activation $\text{try\_start}$ is in lines 7 or 9, where $\text{claim1}$ (who is only written by $t_1$) is always true, the assumptions $\text{assume} (\neg \text{claim1} \mid \mid \text{turn} == 0)$ and $\text{assume} (\neg \text{claim0})$ imply $\text{claim0} \land \text{turn} \neq 1$, hence there are no computations leaving the loop. If the activation $\text{try\_start}$ is in line 13, all the computations are disregarded, since $\text{assume} (\neg cs1)$ conflicts with $cs1$’s actual value at this location, which is always true.

Finally, the only activation $\text{try\_start}$ that can lead to a violation is the one in line 11, which is followed by the command $cs1=\text{true}$. The path $\hat{\pi}_{ terminates}$, starting after the activation $\text{try\_start}$ and reaching the violation, consists of the single command in line 12 (omitting the “non-activating” $\text{try\_start}$ call). The original location in $t_E$ in which this computation starts, is Line 24 (see Figure 4.2), which is exactly the location passed to the activation $\text{try\_start}$. Hence, we compute $\text{wp}(\hat{\pi}_{ terminates}, cs1)[\text{start}/TRUE] = TRUE$, and thus $\text{Reach}_{(t_E)}(\alpha, \beta)$ returns $\psi \triangleq (pc_E = L_{24})$.

### 6.2 Correctness of Environment Query

In this section, we prove the correctness of our technique for answering environment queries. Namely, we prove that the returned formula $\psi$, as obtained by the aforementioned technique, indeed satisfies the properties of Definition 3.2.1. To this end, we need to formally describe the connection between $t_E$ and $P_E$, and between states of $P_E$ and states of the original program $P$. Evidently, such connections would be very similar to those between $P_M$ and $P$. Hence some proofs do not contain the same amount of detail as we used for the case of $P_M$, and others are only sketched, while emphasizing the relevant differences.
\begin{verbatim}
bool claim0 = false, claim1 = false;
bool cs1 = false;
int turn;
bool start = false;

void try_start(int pc_aux) {
    pc_tl = pc_aux;
    if (start) {
        claim0 = havoc_bool();
        turn = havoc_int();
        if (turn) {
            start = true;
            assume(!claim0);
            assume(claim1);
        } else {
            start = false;
            assume(!claim1 || turn == 0);
        }
    }
}
\end{verbatim}

\begin{verbatim}
void P1() {
    try_start(L19);  // Label 19 in Figure 4.2
    while (true) {
        claim1 = true;
        try_start(L21);
        turn = 0;
        try_start(L22);
        while (claim0 && turn != 1) {
            try_start(L23);
        }
        try_start(L24);
        cs1 = true;
        try_start(L25);
        assert(!start || !cs1);
        cs1 = false;
        try_start(L27);
        claim1 = false;
        try_start(L28);
    }
}
\end{verbatim}

Figure 6.1: The sequential program $P_3$ for computing $Reach_{t_E}(\alpha, \beta)$ for $\alpha = \neg cs1 \land claim0 \land (\neg claim1 \lor turn = 0)$ and $\beta = cs1$. For the convenience of presentation, we split the $assume(\alpha)$ to three $assume$ commands.

**General Structure**  
$P_E$ is constructed for the purpose of answering an environment query $Reach_{t_E}(\alpha, \beta)$. The code of $P_E$ is based on the code of $t_E$. Technically, it contains original commands of $t_E$, new assertions, calls to the $try_start$, and the $try_start$ function itself. Since assertions and function calls are only inserted between commands and do not change the control flow (unlike if commands), the control flow of $t_E$ within $P_E$ is preserved. This is similar to the case of $P_M$, where except original commands of $t_M$ we had new assertions, env_move calls and the env_move function.

To argue about the relation of $P_E$ to $t_E$, we need to formally define a mapping between labels of $P_E$ to labels of $t_E$. Essentially, we map each label of $P_E$ to the label (in $t_E$) of the next original command of $t_E$ to be executed in $P_E$. The definition is very similar to Definition 4.1.3 for $P_M$.

**Definition 6.2.1** ($\text{Lab}_E$). Let $\hat{l}$ be a label of $P_E$, not inside $try_start$, with $cmd(\hat{l}) = c$. We define $\text{Lab}_E(\hat{l})$ recursively as follows:

- If $c$ is an original command of $t_E$, appearing at label $l$ in $t_E$, then $\text{Lab}_E(\hat{l}) = l$.
- Otherwise, $c$ is either an assert or a try_start call. We define $\text{Lab}_E(\hat{l}) = \text{Lab}_E(\hat{l}')$, where $\hat{l}'$ is the next label in $P_E$ (not inside $try_start$) after the assert or the try_start (resp.).

We can now define the relation between states of $P_E$ and states of $P$, similarly to Definition 4.2.1.

**Definition 6.2.2** ($\text{Extend}_E$). Let $\hat{s} = (\hat{l}, \sigma)$ be a state of $P_E$, s.t. $\hat{l}$ is not inside $try_start$. We define the set $\text{Extend}_E(\hat{s})$ to be the set of all states $s = (l, \sigma)$ of $P$ such that:

- $l_E(s) = \text{Lab}_E(l(\hat{s}))$. 

---

52
• For every $v \in V$: $v \in \hat{V}_E \Rightarrow \sigma(s)(v) = \sigma(\hat{s})(v)$.

Observation 6.2.3.  1. Unlike Definition 4.2.1, the definition above does not make any requirement for the value $\hat{\sigma}(pc_E)$. Each location in $P_E$ is uniquely mapped to a location in $t_E$ using $Lab_E$. However, $P_E$ does contain an explicit variable for $pc_E$. The purpose of this variable is to allow the query parameters $\alpha$ and $\beta$ to argue about the label of $t_E$. Hence, it is important that $pc_E$ will have an updated value at every place where we use one of the input formulas ($\alpha$ or $\beta$) in which it may appear. This is indeed the case, as $\alpha$ appears only inside the $\text{try}_{\text{start}}$ function, whose first command is used to update $pc_E$, and $\beta$ appears in assertions, which are placed immediately after $\text{try}_{\text{start}}$ calls (and with no original command of $t_E$ between them). The original commands of $t_E$ do not address this variable.

2. Similar to Observation 4.2.2, $\text{Extend}_{E}(\hat{s})$ is also never empty.

3. We would have also liked to claim, similar to Observation 4.2.3, that if $s \in \text{Extend}_{E}(\hat{s})$, then for every formula $\gamma$ over $\hat{V}_E \cap (V \cup \{pc_E\})$, it holds that $s \models \gamma \iff \sigma(\hat{s}) \models \gamma$. However, as mentioned above, the value of $pc_E$ is only updated (and corresponds to $l_{E}(s)$) at specific locations of $P_E$. Hence, if $\gamma$ contains $pc_E$, this general claim may be incorrect. However, the claim is correct in all cases where $pc_E$ has an updated value. In particular, within $\text{try}_{\text{start}}$ and in assertions.

Next, we have a series of lemmas describing the properties of $P_E$ and the connection between $P_E$ and $t_E$. The correctness proof for our mechanism of answering environment queries is concluded by Theorem 6.1.

First, we wish to prove that computations of $P_E$ in which the $\text{start}$ variable is $\text{TRUE}$, correspond to computations of $t_E$. This is formalized by Lemma 6.2.7. To prove Lemma 6.2.7, we first need the following three technical lemmas.

Lemma 6.2.4 (Start activation). Let $\hat{\rho}$ be a computation in $P_E$ from a state $\hat{s}$ to a state $\hat{s}' \neq \epsilon$, such that $\sigma(\hat{s})(\text{start}) = \text{TRUE}$. Then $\sigma(\hat{s}'')(\text{start}) = \text{TRUE}$ for every $\hat{s}''$ in $\hat{\rho}$.

Proof. Assume to the contrary that the lemma is false, then there exists some $\hat{s}''$ in $\hat{\rho}$, which is the first state in $\hat{\rho}$ such that $\sigma(\hat{s}'')(\text{start}) \neq \text{TRUE}$. $\hat{s}'' \neq \hat{s}$, the first state in $\hat{\rho}$, since $\sigma(\hat{s})(\text{start}) = \text{TRUE}$. Let $\hat{s}_p$ be the state preceding $\hat{s}''$ in $\hat{\rho}$. Since $\hat{s}''$ is the first state such that $\sigma(\hat{s}'')(\text{start}) \neq \text{TRUE}$, $\text{cmd}(\hat{s}_p)$ must change the value of $\text{start}$ (from $\text{TRUE}$ to $\sigma(\hat{s}'')(\text{start}) = \text{FALSE}$).

However, according to the construction described in Section 6.1, the only commands in $P_E$ that change the value of $\text{start}$ (after initialization) are $\text{start} = \text{TRUE}$ inside the $\text{try}_{\text{start}}$ function, and violated assertions (which transition to $\epsilon$, in which the value of $\text{start}$ is undefined). None of the commands in $\hat{\rho}$ can be a violated assertion, as this would lead the computation to $\epsilon$, which is a final state. This is impossible, since the last state of $\hat{\rho}$ satisfies $\hat{s}' \neq \epsilon$. Hence, $\text{cmd}(\hat{s}_p) = "\text{start} = \text{TRUE}"$, and since $\hat{s}''$ is obtained from $\hat{s}_p$ by
With the help of the previous two lemmas, we can now also argue about computations that

\[ \text{cmd}(\hat{s}_p) = "\text{start} = \text{TRUE}" \], it must hold that \( \sigma(\hat{s}'')\text{(start)} = \text{TRUE} \), which is a contradiction.

Lemma 6.2.5 (State preservation in \( P_E \)). Let \( \hat{\rho} \) be a computation in \( P_E \) from a state \( \hat{s} \) to a state \( \hat{s}' \neq \epsilon \), such that \( \sigma(\hat{s})\text{(start)} = \text{TRUE} \). Assume that \( l(\hat{s}) \) and \( l(\hat{s}') \) are not inside \( \text{try}\_\text{start} \), and that \( \hat{\rho} \) has no original commands of \( t_E \). Then for every state \( s \) of \( P \): \( s \in \text{Extend}_E(\hat{s}) \iff s \in \text{Extend}_E(\hat{s}') \).

Proof. The only other commands in \( P_E \) which are not original commands of \( t_E \) are \( \text{try}\_\text{start} \) calls and executions (i.e., commands within \( \text{try}\_\text{start} \), and assertions). Since, \( \text{Lab}_E \) maps a label of \( P_E \) to the label associated with the next original command of \( t_E \), and there is no such command between \( \hat{s} \) and \( \hat{s}' \), it follows that \( \text{Lab}_E(l(\hat{s})) = \text{Lab}_E(l(\hat{s}')) \). The definition of \( \text{Extend}_E \) only depends on the mapping \( \text{Lab}_E \), and the values of variables in \( V_E \cap V \). Hence, it is sufficient to show that \( \sigma(\hat{s})(v) = \sigma(\hat{s}')\text{(v)} \) for every \( v \) in \( V \). Since \( \hat{s}' \neq \epsilon \), none of the assertions in \( \hat{\rho} \) can be violated. Hence, all of the assertions in \( \hat{\rho} \) do not change the valuation \( \sigma \).

Further, since \( \sigma(\hat{s})\text{(start)} = \text{TRUE} \), by Lemma 6.2.4, \( \sigma(\hat{s}'')\text{(start)} = \text{TRUE} \) for every \( \hat{s}'' \) in \( \hat{\rho} \). When \( \text{start} = \text{TRUE} \), the \( \text{try}\_\text{start} \) function, only updates the value of \( \text{pc}_E \) (which is not in \( V \)) and exits without changing any other variable.

Hence, none of the commands within \( \hat{\rho} \) change \( \sigma(v) \) for every \( v \) in \( V \). Therefore, \( \sigma(\hat{s})(v) = \sigma(\hat{s}')\text{(v)} \) for every \( v \) in \( V \) as required.

With the help of the previous two lemmas, we can now also argue about computations that include original commands of \( t_E \) as well.

Lemma 6.2.6 (Single step of \( t_E \) mapping). Let \( \hat{\rho} \) be a computation in \( P_E \) from \( \hat{s} \) to \( \hat{s}' \neq \epsilon \), such that \( \sigma(\hat{s})\text{(start)} = \text{TRUE} \) and \( l(\hat{s}) \), \( l(\hat{s}') \) are not inside \( \text{try}\_\text{start} \). Assume also that \( \text{cmd}(\hat{s}) \) is an original command of \( t_E \) and the rest of the commands in \( \hat{\rho} \) are not. Then for every state \( s \in \text{Extend}_E(\hat{s}) \) there exists a state \( s' \in \text{Extend}_E(\hat{s}') \) such that \( s' \in \text{next}(s,t_E) \).

Proof. Let \( c = \text{cmd}(\hat{s}) \). If \( s \in \text{Extend}_E(\hat{s}) \), then \( \text{Lab}_E(l(\hat{s})) = l_E(s) \). Since \( c \) is an original command of \( t_E \), it follows by the definition of \( \text{Lab}_E \), that \( c \) is the same command as in label \( l_E(s) \) in \( t_E \), i.e., \( c = \text{cmd}(s,t_E) \). Hence, we can apply \( c \) on \( s \) in \( t_E \).

Additionally, \( \text{Vars}(c) \subseteq V_E \subseteq \hat{V}_E \), and since \( s \in \text{Extend}_E(\hat{s}) \), it follows that \( \sigma(s)|_c = \sigma(\hat{s})|_c \). Hence, any modification that \( c \) can apply on \( \sigma(\hat{s}) \), can also be applied to \( \sigma(s) \). Additionally, the control flow of \( t_E \) within \( P_E \) is preserved (as the only new commands are \( \text{try}\_\text{starts} \) and assertions), and any condition within \( c \) that may affect branching, holds or is violated in \( \hat{s} \) iff it holds or is violated in \( s \) (resp.).

Hence, if \( \hat{s}'' \) is the second state in \( \hat{\rho} \) (obtained after applying \( c \) to \( \hat{s} \)), then there exists \( s' \) in \( P \) obtained by applying \( c = \text{cmd}(s,t_E) \) to \( s \) (i.e., \( s' \in \text{next}(s,t_E) \)), which satisfies \( s' \in \text{Extend}_E(\hat{s}'') \).

By Lemma 6.2.4, \( \sigma(\hat{s}'')\text{(start)} = \text{TRUE} \). Since \( c \) is an original command of \( t_E \), \( l(\hat{s}'') \) cannot be inside \( \text{try}\_\text{start} \) (but it can be a \( \text{try}\_\text{start} \) call). Thus, we can use Lemma 6.2.5 (for the suffix of \( \hat{\rho} \) starting from \( \hat{s}'' \)) to obtain \( s' \in \text{Extend}_E(\hat{s}') \), as required.
Lemma 6.2.7 (Computations of \( P_E \)). Let \( \hat{\rho} \) be a computation in \( P_E \) from \( \hat{s} \) to \( \hat{s}' \neq \epsilon \) such that \( l(\hat{s}), l(\hat{s}') \) are not inside \texttt{try.start} and \( \sigma(\hat{s})(\text{start}) = \text{TRUE} \).

Then for every \( s \in \text{Extend}_E(\hat{s}) \) there exists \( s' \in \text{Extend}_E(\hat{s}') \) and a computation of \( t_E \) in \( P \) from \( s \) to \( s' \).

Proof. Let \( s \in \text{Extend}_E(\hat{s}) \). If \( \hat{\rho} \) has no original commands of \( t_E \), then we can choose \( s' = s \) (and a computation of length 0) and the results follows from Lemma 6.2.5. Otherwise, \( \hat{\rho} \) contains an original command of \( t_E \). Note that by Lemma 6.2.5, if \( \hat{s}'' = \hat{s}' \) is the first state in \( \hat{\rho} \) such that \( \text{cmd}(\hat{s}'') \) is an original command of \( t_E \), then \( s \in \text{Extend}_E(\hat{s}'') \). Therefore, proving the lemma with \( \hat{s}'' \) as the initial state would prove the lemma for \( \hat{s} \) as well. Thus, we can assume w.l.g. that \( \text{cmd}(\hat{s}) \) is an original command of \( t_E \).

Let \( \hat{s}_0, \ldots, \hat{s}_k \) be a partial series of the states in \( \hat{\rho} \), consisting of all states \( \hat{s}'' \) such that \( \text{cmd}(\hat{s}'') \) is an original command of \( t_E \) and including \( \hat{s}' \) as well (even if \( \text{cmd}(\hat{s}') \) is not an original command of \( t_E \)). By our assumption, \( \hat{s}_0 = \hat{s} \). Moreover, by Lemma 6.2.4, \( \sigma(\hat{s}_i)(\text{start}) = \text{TRUE} \) for \( 0 \leq i < k \). Further, according to our state selection, for every \( 0 \leq i < (k - 1) \), there exists a computation in \( P_E \) from \( \hat{s}_i \) to \( \hat{s}_{i+1} \), in which \( \text{cmd}(\hat{s}_i) \) is the only original command of \( t_E \). Additionally, \( l(\hat{s}_i) \) is not inside \texttt{try.start} for \( 0 \leq i \leq k \) (for \( \hat{s}_k = \hat{s}' \) by the lemma’s condition, and for the rest, \( \text{cmd}(\hat{s}_i) \) is an original command of \( t_E \), which cannot be inside \texttt{try.start} \)). Hence, by applying Lemma 6.2.6 iteratively \( k \) times (starting from \( s_0 = s \)), we obtain a series of \( k \) states in \( P \), \( s_1 \ldots s_k \), such that \( s_i \in \text{Extend}_E(\hat{s}_i) \) and \( s_i \in \text{next}(s_{i-1}, t_E) \) for \( 1 \leq i \leq k \). Since \( \hat{s}_k = \hat{s}' \), we can choose \( s' = s_k \) to satisfy the lemma. \( \square \)

The purpose of the lemmas above was to describe how a computation of \( P_E \), in which the \texttt{start} variables was activated, corresponds to a computation of \( t_E \) in \( P \). This is conceptually similar to lemma 4.2.6. A computation that does not use the abstracting commands, is de facto a computation of a single thread. In \( P_M \), not using the abstraction meant avoiding the \texttt{env.move} function. In \( P_E \), the activation of the \texttt{start} variable prevents the computation from havocking the variables written by \( t_M \).

Now, we also need a converse relation. More precisely, we need to show that a reachable computation \( \rho \) of \( t_E \) is indeed represented within \( P_E \). For the correctness of our proof, we are only interested in computations from \( \alpha \) to \( \beta \). This again resembles a previous claim we proved about \( P_M \), in Lemma 5.1.6. There, we proved that violating computations of \( P \) have a representative violating computation in \( P_M \), using commands of \( t_M \) in \( P_M \) and the \texttt{env.move} function to abstract \( t_E \). In the following lemma, we show that the initial computation in \( P \) which reaches the first state of \( \rho \) (and hence making \( \rho \) reachable) is represented by commands of \( t_E \) in \( P_E \) and \texttt{try.start} to abstract \( t_M \). The representation of \( \rho \) does not need to use the abstraction. The proof is very similar to Lemma 5.1.6, and therefore it is only sketched, with explanations of the differences from and similarities to Lemma 5.1.6.
Lemma 6.2.8. Let $\rho$ be a computation of $t_E$ from $s$ to $s'$ that is reachable in $P$ such that $s \models \alpha$, $s' \models \beta$ and $s, s'$ are a cut-point states (the same conditions as in the first property of Definition 3.2.1). Then there exists an initial violating computation $\hat{\rho}$ in $P_E$.

Proof Sketch. $\rho$ is reachable in $P$, then there exists a computation $\rho_0$ in $P$ from an initial state, $s_0$ of $P$, to $s$. $\rho_0$ contains both commands of $t_M$ and commands of $t_E$. To prove the lemma, we need to partition $\rho_0$ to segments (i.e., sub-computations of $\rho_0$), similar to Definition 5.1.4. However, in Definition 5.1.4, we split $\rho_0$ to computations of $t_M$ and computations of $t_E$, and further split computations of $t_M$ at cut point states. For this lemma, we need to split computations of $t_E$ at cut point states, and leave computations of $t_M$ as a whole segment (even if they contain inner cut point states).

Then, we can iteratively match each segment $r$ with a computation $\hat{r}$ in $P_E$, such that the first computation starts from an initial state, and each next one starts from the last state of the previous computation, thus creating an initial computation in $P_E$.

A segment $r$ of $t_E$ with no inner cut-point states would correspond to a computation $\hat{r}$ of $P_E$ containing only original commands of $t_E$. When $r$ reaches the cut-point state, $\hat{r}$ would reach a try_start call. This is similar to Lemma 5.1.6, where the corresponding computation in $P_M$ reaches an env_move call. if $r$ is a segment of $t_M$, the try_start function would be used to abstract $r$ (i.e., $\hat{r}$ would be an execution of try_start). $\hat{r}$ would skip the inner if-statement (i.e., not setting start to $TRUE$). Since the try_start havoc all the shared variables written by $t_M$, any modification performed by $r$ is possible in $\hat{r}$ as well (for all variables relevant to $P_E$). If we have two adjacent segments of $t_E$, the try_start can also abstract a zero length computation of $t_M$, since one possible result of a havoc command is keeping the existing value of a variable.

Eventually, the constructed computation would end with a try_start (whether it is a segment of $t_E$ reaching a cut-point state, or a segment of $t_M$ abstracted by that try_start). This last try_start will be chosen as the activation try_start, i.e., it sets start to $TRUE$ and assumes $\alpha$. Since $\rho_0$ ends with $s$ and $s \models \alpha$, the compatible state in $P_E$ would also satisfy $\alpha$, thus not pruning the computation.

To complete, we need to match each command in $\rho$ with the compatible command in $P_E$. Since start was already set to $TRUE$, when the computation in $P_E$ reaches another try_start call, it will skip it (except for updating pc_E). This process would end at some state $s'$ such that $s' \in \mathit{Extend}_E(s')$ and cmd($s'$) is assert ($!start \implies \neg\beta$). start was set to $TRUE$ at the activation try_start and is unchanged for the rest of the computation. Since $s' \models \beta$, we will have $s(\hat{s'}) \models start \land \beta$. Hence, the assertion will be violated and the constructed computation is indeed an initial violating computation. \hfill \Box

After discussing the relation between $P_E$ and $t_E$, we need two additional lemmas that characterize the violating computations in $P_E$ and the return value of our mechanism.

Lemma 6.2.9. Let $\hat{\rho} = \hat{s} \rightarrow \ldots \hat{s'} \rightarrow \epsilon$ be a violating initial computation in $P_E$. The activation try_start (i.e., an execution of try_start where start is changed from FALSE to $TRUE$) exists and is unique.
Proof. All the assertions in \( P_E \) are of the form \( \text{assert} \left( \text{!start} \mid \neg \beta \right) \). The variable \( \text{start} \) has an initial value of \( \text{FALSE} \). If \( \text{start} \) is left unchanged during \( \hat{\rho} \), then in particular \( \sigma(\hat{s}'')(\text{start}) = \text{FALSE} \), which is a contradiction to \( \hat{\rho} \) being a violating computation.

Since the only location in which \( \text{start} \) can be set to \( \text{TRUE} \) is within the \text{try\_start} function, there exists an activation \text{try\_start}, which sets \( \text{start} \) to \( \text{TRUE} \). After setting \( \text{start} \) to \( \text{TRUE} \), the \text{try\_start} assumes \( \alpha \) and then exits. Let \( \hat{s}'' \) be the first state in \( \hat{\rho} \) after the first activation \text{try\_start}. Then by Lemma 6.2.4 for the sub-computation of \( \hat{\rho} \) from \( \hat{s}'' \) to \( \hat{s}' \), \( \text{start} \) remains \( \text{TRUE} \) in all the next states. Hence, the activation \text{try\_start} is unique. \( \Box \)

With the proof above, the next Lemma is well defined:

**Lemma 6.2.10.** Let \( \hat{\rho} = \hat{s} \rightarrow \ldots \hat{s}' \rightarrow \epsilon \) be a violating initial computation in \( P_E \), and let \( \hat{s}'' \) be the first state in \( \hat{\rho} \) after the activation \text{try\_start}. Let \( \hat{\rho}_{end} \) be the sub-computation of \( \hat{\rho} \) from \( \hat{s}'' \) to \( \hat{s}' \), \( \hat{\pi}_{end} \) the path of \( \hat{\rho}_{end} \), and \( l \) be the program location parameter passed to the activation \text{try\_start}.

Then \( \sigma(\hat{s}'') \models \alpha \land (pc_E = l) \land wp(\hat{\pi}_{end}, \beta)[\text{start}/\text{TRUE}] \).

Proof. The only command in the activation \text{try\_start}, after setting \( \text{start} \) to \( \text{TRUE} \), is \text{assume} \( (\alpha) \), after which the function returns. Hence, \( \hat{s}'' \) is the next state in \( \hat{\rho} \) after \text{assume} \( (\alpha) \) is executed. Since passed \text{assume} commands do not change the valuation \( \sigma \), it follows that \( \sigma(\hat{s}'') \models \alpha \).

The assertions in \( P_E \) are of the form \( \text{assert} \left( \text{!start} \mid \neg \beta \right) \). Since \( \hat{\rho} \) is a violation, it follows that \( \hat{s}' \models \text{start} \land \beta \) and in particular, \( \hat{s}' \models \beta \). Since \( \hat{\rho}_{end} \) is a computation from \( \hat{s}'' \) to \( \hat{s}' \), by the definition of a weakest precondition, it follows that \( \sigma(\hat{s}'') \models wp(\hat{\pi}_{end}, \beta) \). Since, \( \hat{s}'' \) appears after the activation \text{try\_start}, then \( \sigma(\hat{s}'') (\text{start}) = \text{TRUE} \). Therefore, it must also hold that \( \sigma(\hat{s}'') \models wp(\hat{\pi}_{end}, \beta)[\text{start}/\text{TRUE}] \).

Finally, the activation \text{try\_start} sets the value of \( pc_E \) to \( l \). This value is only set at the beginning of \text{try\_start}, and is unchanged until the next \text{try\_start} call. Hence, \( \sigma(\hat{s}'')(\text{pc}_E) = l \), which means that \( \sigma(\hat{s}'') \models (pc_E = l) \)

With the last lemma, we can finally complete the correctness proof of our technique for answering an environment query:

**Theorem 6.1 (Environment Query Construction).** Let \( P_E \) be a sequential program constructed for an environment query \( \text{Reach}(t_E)(\alpha, \beta) \) and let \( \psi \) be the results of the query, obtained according to the description in Section 6.1. Then \( \psi \) satisfies the properties of Definition 3.2.1.

Proof. First, if \( \alpha \land \beta \neq \text{FALSE} \), then \( \beta \) is a valid answer for \( \text{Reach}(t_E)(\alpha, \beta) \), as mentioned in Observation 3.2.2. Otherwise, the answer is according to the construction of \( P_E \).

Assume first that the constructed \( P_E \) is safe. In this case, we return \( \psi = \text{FALSE} \). When \( \psi = \text{FALSE} \), the second and third property of Definition 3.2.1 hold vacuously. Additionally, by Lemma 6.2.8, when \( P_E \) is safe, there is no computation of \( t_E \) from \( s \models \alpha \) to \( s' \models \beta \) that is reachable in \( P \) such that \( s, s' \) are cut-point states. Hence the first property holds as well.
Assume now that $P_E$ is unsafe, and let $\hat{\rho} = \hat{s} \rightarrow \cdots \rightarrow \hat{s}' \rightarrow \epsilon$ be a violating computation in $P_E$. Let $\hat{s}''$ be the first state after the activation $\text{try} \_ \text{start}(l)$ ($l$ being the parameter passed), and $\hat{\pi}_{\text{end}}$ the path used by $\hat{\rho}$ starting from $\hat{s}''$ until $\hat{s}'$. The returned formula is $\psi = (\text{pc}_E = l) \land \text{wp}(\hat{\pi}_{\text{end}}, \beta)[\text{start}/\text{TRUE}]$. According to Lemma 6.2.10, $\sigma(\hat{s}'') \models \alpha \land \psi$. In particular, this means that $\alpha \land \psi \neq \text{FALSE}$. Hence, the first and third property of Definition 3.2.1 hold.

For the second property, let $s_\psi$ be a state of $P$, such that $s_\psi \models \psi$. We construct a state $\hat{s}_\psi$ of $P_E$ as follows:

- $l(\hat{s}_\psi) = l(\hat{s}'')$.
- $\sigma(\hat{s}_\psi)(\text{pc}_E) = l$.
- $\sigma(\hat{s}_\psi)(\text{start}) = \text{TRUE}$.
- For every $v \in V \cap \hat{V}_E$: $\sigma(\hat{s}_\psi)(v) = \sigma(s_\psi)(v)$.

We first wish to show that $s_\psi \in \text{Extend}_E(\hat{s}_\psi)$. The properties for variables in $V \cap \hat{V}_E$ hold immediately, by the definition of $\hat{s}_\psi$. It is left to show that $l_E(s_\psi) = \text{Lab}_E(l(\hat{s}_\psi))$. $\text{Lab}_E(l(\hat{s}''))$ is mapped to the label associated with the next original command of $t_E$ to be executed in $P_E$, starting from location $l(\hat{s}'')$. $\hat{s}''$ is the next state after a $\text{try} \_ \text{start}$, called with parameter $l$, which is also the label associated with the next original command of $t_E$ to be executed. Hence, $\text{Lab}_E(l(\hat{s}'')) = l$. Since $s_\psi \models \psi$, then in particular $s_\psi \models (\text{pc}_E = l)$, i.e., $l_E(s_\psi) = l$. Since $l(\hat{s}_\psi) = l(\hat{s}'')$, we get $l_E(s_\psi) = \text{Lab}_E(l(\hat{s}_\psi))$, and $s_\psi \in \text{Extend}_E(\hat{s}_\psi)$ as required.

Note also that $\hat{s}_\psi$ has an updated value of $\text{pc}_E$, i.e., $\sigma(\hat{s}_\psi)(\text{pc}_E) = l = \text{Lab}_E(l(\hat{s}_\psi))$. Hence, since $s_\psi \models \psi$ and $s_\psi \in \text{Extend}_E(\hat{s}_\psi)$, it follows that $\sigma(\hat{s}_\psi) \models \psi$ (see the last property of Observation 6.2.3). In particular, $\sigma(\hat{s}_\psi) \models \text{wp}(\hat{\pi}_{\text{end}}, \beta)[\text{start}/\text{TRUE}]$. Since $\sigma(\hat{s}_\psi)(\text{start}) = \text{TRUE}$, it also holds that $\sigma(\hat{s}_\psi) \models \text{wp}(\hat{\pi}_{\text{end}}, \beta)$. Hence, by the definition of a weakest precondition, there exists a computation $\hat{\rho}_\psi$ from $\hat{s}_\psi$ to some $\hat{s}'_\psi$, which uses the path $\hat{\pi}_{\text{end}}$ such that $\sigma(\hat{s}'_\psi) \models \beta$.

The last label of $\hat{\pi}_{\text{end}}$ is $l(\hat{s}')$, which must be a label associated with an assertion, as the next state in $\hat{\rho}$ after $\hat{s}'$ is $\epsilon$. Therefore, $\hat{s}'_\psi \neq \epsilon$ and $l(\hat{s}'_\psi)$ is not inside $\text{try} \_ \text{start}$ (which does not contain assertions). Then by Lemma 6.2.7, there exists $s'_\psi \in \text{Extend}(\hat{s}'_\psi)$ and a computation of $t_E$ from $s_\psi$ to $s'_\psi$ in $P$. The $\text{pc}_E$ variable has an updated value at $\hat{s}'_\psi$, as well, since assertion are placed immediately after $\text{try} \_ \text{start}$ calls (which update $\text{pc}_E$), and before additional original commands of $t_E$. Hence, again by Observation 6.2.3, $s'_\psi \models \beta$. \qed
Chapter 7

Extending the Algorithm

In this chapter we discuss two extensions of our algorithm. First, in Sections 7.1 and 7.2, we extend it to concurrent programs that consist of more than two threads. Next, in Section 7.3, we extend it to handle assertions that appear in any thread, and not only the one designated as the main thread.

7.1 Extending to Multiple Threads

In this section, we describe the extension of our method to programs with more than two threads. We first give an overview of the differences from (and similarities to) the case of two threads (and in particular, a single environment thread), and discuss the required adaptations to support this setting. We then continue more formally with the necessary new definitions. Finally, Section 7.2 describes how environment queries with multiple threads are answered, which is the key ingredient of this extension.

Consider a program $P$ with threads $t_1, \ldots, t_m$, and assume $t_1$ is chosen as $t_M$. The environment of $t_1$ now consists of all the other threads: $T = \{t_2, \ldots, t_m\}$. Recall that in the case of two threads, the verification problem was divided to verification of the main thread, performed by Algorithm 4.1, and answering environment queries, described in Chapter 6. Note that our correctness proof of Algorithm 4.1 in Section 5.1 only relies on our construction and refinement of $P_M$ according to the chosen main thread and the results of environment queries, and in particular the construction of the env_move function to abstract the environment. This means that the correctness of Algorithm 4.1, which solves the verification problem “from the point of view of $t_M$”, is independent of the number of environment threads, as long as information describing the environment is added correctly to $P_M$.

More precisely, this means that the algorithm for handling the main thread, Algorithm 4.1, remains the same in the presence of multiple threads. In order for the correctness proof of Section 5.1 to be applicable for the case of multiple environment threads, we need to make sure that (1) the initial env_move function of $P_M$ over-approximates the environment $T$ (Definition 4.2.5), (2) refinements of env_move preserve this over-approximation, and (3) newly added assertions still represent promises of error, with respect to the environment $T$. This is
sufficient, as these are the only modifications applied to \( P_M \) during Algorithm 4.1.

To achieve this, we consider the construction (and transformation) of \( P_M \) and the env\_move function in particular. Initially, the env\_move havoc all shared variables changed by the environment. For the case of a single thread \( t_E \), it means all variables of \( t_M \) that are also written by \( t_E \). For the case of multiple environment threads \( T \), it simply means all variables of \( t_M \) that are written by any of the threads in \( T \). Next, there are two types of modifications that can be applied to \( P_M \) during Algorithm 4.1. First, the env\_move is changed by the refinement step described in (4) of Section 4.5 iff an environment query \( \text{Reach}(t_E)(\alpha, \beta) \) returned \( \psi = \text{FALSE} \). Second, new assertions are added to \( P_M \) iff an environment query returned \( \psi \neq \text{FALSE} \). As we argue next, these modifications remain correct, as long as the query’s result is correct. In fact, their correctness relies completely on the correctness of the query’s result. The latter is addressed in Section 7.2.

To this end, we first need to establish a slightly different definition for environment queries, to capture the fact that the environment consists of multiple threads \( T = \{t_2,\ldots, t_m\} \). As each thread \( t_i \in T \) has its own pc variable, \( \overrightarrow{V_M} \) may now include \( \overrightarrow{pc_i} \) as an additional variable for every thread \( t_i \in T \), instead of a single \( pc_E \).

**Definition 7.1.1 (Multithreaded Environment Query).** Let \( P \) be a concurrent programs with threads \( t_1,\ldots, t_m \), and let \( T \subset\subset \{t_1,\ldots, t_m\} \). An environment query \( \text{Reach}(T)(\alpha, \beta) \) receives conditions \( \alpha \) and \( \beta \) over \( V \cup \bigcup_{i: t_i \in T} \{pc_i\} \), and returns a formula \( \psi \) over \( V \cup \bigcup_{i: t_i \in T} \{pc_i\} \) such that:

1. If there exists a computation of \( T \) in \( P \) that is (1) reachable in \( P \), (2) starts from a cut-point state \( s \) s.t. \( s \models \alpha \), (3) ends in a cut-point state \( s' \) s.t. \( s' \models \beta \), then \( \psi \land \alpha \not\equiv \text{FALSE} \).
2. For every state \( s \) s.t. \( s \models \psi \), there exists a computation (not necessarily reachable) of \( T \) in \( P \) from \( s \) to some \( s' \) s.t. \( s' \models \beta \).
3. \( \psi \neq \text{FALSE} \Rightarrow \psi \land \alpha \neq \text{FALSE} \).

Essentially, this definition is identical to Definition 3.2.1, where computations of \( t_E \) are replaced with computations of \( T \), and in which the conditions \( (\alpha, \beta, \psi) \) can address the pc variable of all environment threads. Next, we can rephrase the definition about the env\_move "over-approximating \( T \)“, and also the definition of a promise of error for the case of multiple threads. For the former, we also need to adapt the definition of Extend to the case of multiple threads:

**Definition 7.1.2 (Multithreaded Extend).** Let \( \hat{s} = (\hat{l}, \hat{\sigma}) \) be a state in \( P_M \), s.t. \( \hat{l} \) is not inside env\_move. We define the set \( \text{Extend}(\hat{s}) \) to be the set of all states \( s = (l, \sigma) \) of \( P \) such that:

- \( l_M(s) = \text{Lab}(l(\hat{s})) \).
- For every thread \( t_i \in T \): If \( pc_i \in \overrightarrow{V_M} \) then \( l_i(s)^1 = \sigma(\hat{s})(pc_i) \).

\(^1\) Where \( l_i(s) \) denotes the value of \( pc_i \) in \( s \).
• For every $v \in V$: $v \in \overline{V}_M \Rightarrow \sigma(s)(v) = \sigma(\hat{s})(v)$.

**Definition 7.1.3** (Multithreaded Over-approximation). We say that $\text{env}_\text{move}$ over-approximates the computations of $T$ in $P$ if for every reachable (and possibly of length zero) computation $\rho$ of $T$ in $P$ from a cut-point state $s$ to a cut-point state $s'$, and for every state $\hat{s}$ of $P_M$ s.t. $s \in \text{Extend}(\hat{s})$, and $\text{cmd}(\hat{s})$ is an $\text{env}_\text{move}$ call, there exists a computation $\hat{\rho} = \hat{s} \rightarrow \cdots \rightarrow s'$ of $P_M$ s.t.

- $l(\hat{s}')$ is the next label in $P_M$ after the $\text{env}_\text{move}$ called at $\hat{s}$, and for every inner state $\hat{s}''$ in $\hat{\rho}$, $l(\hat{s}'')$ is a label inside $\text{env}_\text{move}$ (i.e., $\hat{\rho}$ is a complete single execution of $\text{env}_\text{move}$)
- $s' \in \text{Extend}(\hat{s}')$

**Definition 7.1.4** (Multithreaded Promise of Error). Let $\psi, \psi'$ be formulas over $V \cup \bigcup_{i. t_i \in T} \{pc_i\}$ and let $l, l'$ be labels of $t_M$. We say that $(l, \psi)$ is a promise of $(l', \psi')$ if for every state $s$ of $P$ s.t. $l_M(s) = l$ and $s \models \psi$ there exists a computation in $P$ starting from $s$ to a state $s'$ s.t. $l_M(s') = l'$ and $s' \models \psi'$.

If $(l, \psi)$ is a promise of $(l', \neg b)$ and $\text{cmd}(l') = \text{assert}(b)$ for some condition $b$, then we say that $(l, \psi)$ is a promise of error.

We note that Definition 7.1.4 is almost identical to Definition 4.4.1, with $V \cup \{pc_E\}$ simply replaced by $V \cup \bigcup_{i. t_i \in T} \{pc_i\}$. Definition 7.1.2 also highly resembles Definition 4.2.1, where instead of addressing a single $pc_E$ variable in $\overline{V}_M$, the definition addresses multiple $pc_i$ variables (for each thread $t_i$). Finally, Definition 7.1.3 is also identical to Definition 4.2.5, with computations of $t_E$ replaced by computations of $T$.

In fact, the entire correctness proof from Section 5.1, can be left almost unchanged, while only using the same replacements mentioned above: Computations of $t_E$ become computations of $T$, and $pc_E$ should be replaced by multiple $pc_i$ variables for each thread $t_i$. Naturally, the proof assumes that the environment query returns a correct result according to our new definition. Hence, the core of the extension of the algorithm to multiple threads relies on our technique for answering environment queries, which is explained next.

### 7.2 Environment Queries with Multiple Threads

**Overview** In this section, we wish to describe an algorithm for answering environment queries, $\text{Reach}_{\{T\}}(\alpha_1, \beta_1)$ for some environment $T$ such that $|T| \geq 1$. To preserve modularity, we use an algorithm similar to Algorithm 4.1, which can answer the query by analyzing a single thread with some abstraction of its environment. We select a thread $t_M' \in T$ and view it as the main thread for the query. The environment in this context will consist of $T' = T \setminus \{t_M'\}$. During the algorithm, we would require some information concerning $T'$, which again will be answered by environment queries: $\text{Reach}_{\{T'\}}(\alpha_2, \beta_2)$. Note that the size of the environment for the inner queries is reduced. Thus, we can continue recursively until reaching an environment of size one, which was already handled in Chapter 6.
After selecting $t'_M$, we construct a sequential program $P'_M$ with try_start calls and assertions as described in Chapter 6. After each try_start call (and before the assert), we add a call to an env_move function. $P'_M$ will have an env_move’ function of its own, which is different from the env_move of $P_M$. The goal of env_move’ is to over-approximate $T'$.

Recall that we are interested in over-approximating reachable, but not necessarily initial, computations of $T$ in $P$. For each such computation $\rho$, there exists some initial computation $\rho_i$ in $P$, which is a prefix of a computation $\rho'$ such that $\rho' = \rho_i \cdot \rho$. Each step in $\rho_i$ can be performed by any of the threads in $P$, while each step in $\rho$ is restricted to $T$. Our construction of $P'_M$ models steps of $t'_M$ precisely, and it should abstract the steps of all other threads in both $\rho_i$ and $\rho$.

**Over-approximating $\rho_i$** The purpose of the try_start function is to abstract $\rho_i$, and to non-deterministically decide when $\rho_i$ ends, $\alpha_1$ is satisfied, and $\rho$ should start. To this end, as long as the start variable remains FALSE, the try_start function havoc all variables written by any thread of $P$ other than $t'_M$. Similar to Chapter 6, the try_start also non-deterministically sets start to TRUE and assumes $\alpha_1$. As before, we refer to this call as the activation try_start. After this point, the try_start function does not havoc any variables (as start $= TRUE$). As in Chapter 6, we pass the label associated with the next original command of $t'_M$ to try_start, in order to update pc$^M$, the pc variable of $t'_M$.

**Over-approximating $\rho$** When start is set to TRUE, we move from over-approximating $\rho_i$ to over-approximating $\rho$. As before, steps of $t'_M$ are modeled precisely, and we wish to over-approximate the steps of all other threads in $T$, i.e., the threads in $T'$. This over-approximation will be handled by the env_move’ function of $P'_M$. First, note that we are only interested in this over-approximation once start was set to TRUE. Hence, the code of the env_move’ function should start with an if (start){...} condition, and all other commands would be inside the true-branch of this if statement. The rest of env_move’ is constructed in the same manner as described in Chapter 4. The initial env_move’ function havoc variables written by threads in $T'$, similar to the description in Section 4.3. It is later refined by statements of the form if ($\alpha'_2$[W_copy/W]) assume ($\neg \beta'_2$), similar to (4) of Section 4.5, following inner environment queries $\text{Reach}_{(T')}(\alpha_2, \beta_2)$, which returned FALSE.

**Algorithm Outline** Algorithm 7.1 provides the complete procedure of how environment queries are answered. The algorithm follows a similar pattern as Algorithm 4.1. First, if at any stage there are no initial violating computations in $P'_M$, the over-approximation ensures that there is indeed no reachable computation of $T$ in $P$ from $\alpha_1$ to $\beta_1$. In this case, Algorithm 7.1 returns $\psi = FALSE$ (Line 25).

Next, assume that a violating computation $\hat{\rho}$ was found in $P'_M$, and that $\hat{\rho}$ contains at least one env_move’ call after the activation try_start (Line 13). The last env_move’ call

---

2Variables that neither belong to any of the threads in $T$, nor appear in $\alpha_1$ or $\beta_1$ can be omitted, as they clearly do not affect the result.
Algorithm 7.1 AnswerEnvQuery

1: procedure ANSWERENVQUERY($P, T, t'_M, \alpha_1, \beta_1$) 
2: \hspace{1em} $P_M' = \text{add try\_start calls in cut-point locations of } t'_M$ 
3: \hspace{1em} $P_M' = \text{add env\_move calls after try\_start in } P_M'$ and initialize env\_move' 
4: \hspace{1em} $P_M' = \text{add assert (} \neg \text{start} \lor \neg \beta_1 \text{) after every env\_move in } P_M'$ 
5: \hspace{1em} while a violating path exists in $P_M'$ do \hspace{1em} // using sequential MC 
6: \hspace{2em} $\hat{\pi} = \hat{l}_0, \ldots, \hat{l}_{n+1}$ be a violating path (with $\hat{l}_{n+1} = \hat{l}_k$) 
7: \hspace{2em} let $\text{assert (} \neg \text{start} \lor b \text{) be the violated assertion at label } \hat{l}_n$ 
8: \hspace{2em} let try\_start(l) be the activation try\_start at some label $\hat{l}_j$ 
9: \hspace{2em} let $\hat{\pi}' = \hat{l}_{j+1}, \ldots, \hat{l}_n$ be the sub-path of $\hat{\pi}$ starting from the activation try\_start. 
10: \hspace{1em} if there are no env\_move calls in $\hat{\pi}'$ then: 
11: \hspace{2em} return $\psi \triangleq (pc_{M'} = 1) \land wp(\hat{\pi}', \beta_1)[\text{start}/\text{TRUE}]$ 
12: \hspace{1em} end if 
13: \hspace{1em} let $\hat{l}_k$ be the label of last env\_move call in $\hat{\pi}$ (and in $\hat{\pi}'$) 
14: \hspace{1em} let $\hat{\pi}_{\text{start}} = \hat{l}_0, \ldots, \hat{l}_k$ and $\hat{\pi}_{\text{end}} = \hat{l}_{k+1}, \ldots, \hat{l}_n$ 
15: \hspace{1em} $\beta_2 = \text{wp(}\hat{\pi}_{\text{end}}, \neg b\text{)}$ \hspace{1em} // see (1) in Section 4.5 
16: \hspace{1em} $\alpha_2 = \text{post(}\hat{\pi}_{\text{start}}, \hat{\phi}_{\text{init}}\text{)}$ \hspace{1em} // see (2) in Section 4.5 
17: \hspace{1em} let $\psi_2 = \text{Reach}_{(T')}(\alpha_2, \beta_2)$ \hspace{1em} // environment query for $T'$ 
18: \hspace{1em} if $\psi_2$ is FALSE then 
19: \hspace{2em} Let $\alpha'_2, \beta'_2$ be as in (4) in Section 4.5. 
20: \hspace{2em} $P_M' = \text{RefineEnvMove(} P_M', \alpha'_2, \beta'_2\text{)}$ 
21: \hspace{1em} else 
22: \hspace{2em} Add new label $\hat{\nu}$ in $P_M'$ right before $\hat{l}_k$ with $cmd(\hat{\nu}) = \text{assert (} \neg \text{start} \lor \neg \psi_2\text{)}$ 
23: \hspace{1em} end if 
24: \hspace{1em} end while 
25: \hspace{1em} return $\psi \triangleq \text{FALSE}$ 
26: end procedure

will be examined in the same manner as in Algorithm 4.1, by another environment query $\text{Reach}_{(T')}(\alpha_2, \beta_2)$ (Line 17). If we learn that $T'$ cannot lead from $\alpha_2$ to $\beta_2$, env\_move' is refined to eliminate $\hat{\rho}$ (Line 20). Otherwise, a new assertion is added before the env\_move' call (Line 22), whose violation ensures the reachability of $\beta_2$ through computations of $T'$.

Finally, if the algorithm finds an initial violating computation $\hat{\rho}$ in $P_M'$ with no env\_move' calls after the activation try\_start (Line 11), it terminates and returns $\psi$ as appears in Line 11. The returned $\psi$ guarantees the reachability of the violated assertion through steps of $t'_M$ alone. The violated assertion itself guarantees the reachability of $\beta_1$. The computed $\psi$ must also intersect $\alpha_1$ which is assumed at the activation try\_start.

Correctness The formal correctness proof for Algorithm 7.1 combines and repeats the correctness proofs from Sections 5 and 6. The parts about the role of the try\_start function and the final result $\psi$ intersecting $\alpha_1$, are similar to the ones in Chapter 6. The parts about the over-approximation of the env\_move' function and assertion being “promises of $\beta_1$”, i.e., conditions ensuring the reachability of $\beta_1$ through computations of $T$, are similar to the ones used in Chapter 5. Next, we give a formal description of the correctness of Algorithm 7.1, followed by a proof sketch with references to the relevant previously proved lemmas.
Theorem 7.1 (Correctness of Algorithm 7.1). Let $\text{Reach}(T)(\alpha_1, \beta_1)$ be an environment query in $P$ for an environment $T$, and let $\psi$ be the results of the query, obtained by Algorithm 7.1. Then $\psi$ satisfies the properties of Definition 7.1.1.

Proof. (detailed sketch) The proof is by induction on the size of the environment $T$.

**Base Case:** $|T| = 1$: If $|T| = 1$, then $T$ consist of a single thread $t'_M$, with no environment. The $\text{env}_\text{move}'$ mentioned in Algorithm 7.1 would be empty, and can be omitted. There is no need to examine inner environment queries from some $\alpha_2$ to some $\beta_2$, as there are no environment computations. Note that without the $\text{env}_\text{move}'$ functions, the construction is identical to the one in Chapter 6, and the algorithm will terminate after a single model-checker call. Hence, the correctness proof for the base case was already presented in Chapter 6.

**Induction Step:** $|T| > 1$: Assume that $|T| > 1$. let $t'_M \in T$ and let $T' = T\setminus \{t'_M\}$. $|T'| < |T|$, hence by the induction hypothesis, Algorithm 7.1 provides correct results for environment queries about $T'$. The induction hypothesis is needed for the two possible refinements of $P'_M$: adding new assertions and refining $\text{env}_\text{move}'$. Next, we discuss both of these refinements.

$\text{env}_\text{move}'$ Over-approximates $T'$: For this part we need an adjusted version of Lemma 5.1.1. In simple words, we claim that for every reachable computation $\rho$ of $T'$ in $P$ from some $s$ to some $s'$, and for every state $\hat{s}$ of $P'_M$ that “matches” $s$ (with an adapted version of the Extend definition), there exists a complete single execution of the $\text{env}_\text{move}'$ function from $\hat{s}$ to some $\hat{s}'$ matching $s'$. The proof follows the same pattern as the proof of Lemma 5.1.1. At initialization, the lemma is correct as $\text{env}_\text{move}'$ havocs all variables written by threads in $T'$. Refinements only occur after an inner environment query $\text{Reach}(T')(\alpha'_2, \beta'_2)$ returned FALSE. By the induction hypothesis, this means that there are no reachable computations of $T'$ in $P$ from $\alpha'_2$ to $\beta'_2$. We refine $\text{env}_\text{move}'$ with commands of the form $\text{if} (\alpha'_2|W_{\text{copy}}/W]) \text{assume} (\neg \beta_1)$, which only block $\text{env}_\text{move}'$ from reaching $\beta'_2$ when it was called with a state satisfying $\alpha'_2$. Thus the refinement does not “lose” the representation of any reachable computation of $T'$ in $P$.

Assertions in $P'_M$ Represent Promises of $\beta_1$. A promises of $\beta_1$ is a condition $\psi'$ and a label $l$ of $t'_M$, such that for every state $s$ of $P$ with $l_M(s)^3 = l$ and $s \models \psi'$ there exists a computation of $T$ in $P$ from $s$ to some state $s'$ such that $s' \models \beta_1$. We wish to show that the violation of every assertion in $P'_M$ is a promise of $\beta_1$. The proof follows the same pattern as Lemma 5.1.3. At initialization, this holds since all assertions are of the form $\text{assert} (\neg \text{start} \lor \neg \beta_1)$. Other assertions are only added after inner environment queries returned $\psi_2 \triangleq \text{Reach}(T')(\alpha_2, \beta_2) \neq FALSE$. Consider the first newly added assertion. It can only be added after a violating computation $\hat{\rho}$ was found in $P'_M$, leading to an original assertion. Thus, $\beta_2$ is computed as a weakest precondition and ensures the reachability of $\beta_1$ using steps of $t'_M$ alone. Further, by

\footnote{\text{denoting the pC variable of $t'_M$}}
the induction hypothesis, the result $\psi_2$ ensures the reachability of $\beta_2$ using computations of $T'$. Combining computations of $t'_M$ and computations of $T'$ yields computations of $T$. Hence, $\beta_1$ is reachable from $\psi_2$ by computations of $T$. Since the newly added assertion is of the form $\text{assert}(\neg \text{start} \lor \neg \psi_2)$, its violation indeed represents a promise of $\beta_1$. For other assertions, we can continue inductively and use the transitivity of the definition of “promises of $\beta_1$”, similar to Lemma 5.1.3.

$\psi \neq \text{FALSE} \Rightarrow \psi \land \alpha_1 \neq \text{FALSE}$ We now wish to show that whenever Algorithm 7.1 returns $\psi \neq \text{FALSE}$, it means that $\psi \land \alpha_1 \neq \text{FALSE}$. This is in fact the last property of Definition 7.1.1. The proof is based mainly on the proof of Lemma 6.2.10. If Algorithm 7.1 returned $\psi \neq \text{FALSE}$, it means that an initial violating computation $\hat{\rho} = \hat{s} \rightarrow \ldots \hat{s}' \rightarrow \epsilon$ was found in $P'_M$ with no env.move' calls after the activation try.start. Let $\text{assert}(\neg \text{start} \lor b)$ be the violated assertion (Line 7). The state immediately after the activation try.start must satisfy $\psi \land \alpha_1$. It satisfies $\psi$, because $\psi$ was computed as a weakest precondition from $\neg b$ exactly until the activation try.start, using steps of $t'_M$, and the computation $\hat{\rho}$ used these exact steps to reach $\neg b$ and violate the assertion. It satisfies $\alpha_1$ as $\alpha_1$ was assumed at the end of the activation try.start, and otherwise $\hat{\rho}$ could not have continued. We conclude that $\psi \land \alpha_1 \neq \text{FALSE}$.

$\psi = \text{FALSE}$ Implies no Computation from $\alpha_1$ to $\beta_1$ We now wish to show that when Algorithm 7.1 returns $\psi = \text{FALSE}$, it means that there is no reachable computation of $T$ in $P$ from a cut point state $s$ satisfying $\alpha_1$, to a cut point state $s'$ satisfying $\beta_1$. This, together with the property above, proves the first property of Definition 7.1.1.

The proof is a combination of the proofs of Lemma 5.1.6 and Lemma 6.2.8. Assume that there exists such a reachable computation $\rho$. As discuss in the beginning of this section, since $\rho$ is reachable, there exists some initial computation $\rho_i$ in $P$, which is a prefix of a computation $\rho'$ such that $\rho' = \rho_i \cdot \rho$. The main idea of the proof is to construct an initial violating computation $\hat{\rho}'$ in $P'_M$. As long as such a computation exists, Algorithm 7.1 cannot reach Line 25 and would not return $\psi = \text{FALSE}$.

$\hat{\rho}'$ would also consist of two subcomputations $\hat{\rho}' = \hat{\rho}_i \cdot \hat{\rho}$, based on $\rho_i$ and $\rho$. During $\hat{\rho}_i$, the start variable will be FALSE. Thus, the env.move' function, whose body is contained within a if (start){...} statement, will be effectively disabled. Therefore, we can use a construction similar to Lemma 6.2.8 until the activation try.start. $\rho_i$ would be partitioned into segments, where each segment either corresponds to a computation of $t'_M$ with no inner cut-point states, or to a computation of the rest of the threads in $P$, other than $t'_M$. Segments of $t'_M$ would be represented in $\hat{\rho}_i$ by the exact same commands of $t'_M$, which appear in $P'_M$. Segments of the other threads would be represented by the try.start function. $\hat{\rho}_i$ would end after the activation try.start, where $\hat{\rho}$ would start.

The connecting state $\hat{s}$ between $\hat{\rho}_i$ and $\hat{\rho}$ would match $s$, which connects $\rho_i$ and $\rho$. Since $s \vdash \alpha_1$, the assume($\alpha_1$) command within the activation try.start would not trim the computation, and it can continue from $\hat{s}$. Next, during the activation try.start, the start variable is set to TRUE, which means that the havocs within the try.start function are
disabled. Thus, we can now use a construction similar to Lemma 5.1.6. As before, $\rho$ would be partitioned into segments. Recall that $\rho$ is a computation of $T$. Thus, each segment will either correspond to a computation of $t_M'$ with no inner cut-point states, or to a \textit{computation of $T'$ in $P$}. Segments of $t_M'$ would be represented by the exact same commands in $P_M'$, and segments $T'$ would be represented by the $\text{env}_{\text{move}}'$ function.

Finally, $\hat{\rho}$ would reach a state $s'$ matching $s'$. Since $s'$ is a cut-point state, there would be an $\text{assert(} \neg \text{start} \lor \neg \beta_1 \text{)}$ command at the compatible label at the end of $\hat{\rho}$, i.e., at $l(s')$. Since $s' \models \beta_1$, $s'$ would also violate that assertion. Note that it is also possible that $\hat{\rho}$ would violate a previous assertion and thus would not continue until $s'$. Nevertheless, we get a violating computation $\rho'$ in $P_M'$.

\textbf{$\psi$ Guarantees the Reachability of $\beta_1$} To conclude, we need to show that the second property of Definition 7.1.1 holds. That is, if $s \models \psi$, then there exists a computation of $T$ in $P$ from $s$ to some $s'$ satisfying $\beta_1$. The proof is a combination of Item 2 of Theorem 5.1, and the proof of the second property of Definition 3.2.1 in Theorem 6.1. The main idea is that the returned $\psi$ is computed as the weakest precondition in $P_M$, from the violation of some assertion $\text{assert(} \neg \text{start} \lor b \text{)}$ to the activation $\text{try}_\text{start}$. Therefore, if $s \models \psi$, there exists a computation of $t_M'$ in $P_M'$ from $s$ to some $s''$ satisfying $\neg b$. We already showed that the location of the violated assertion, together with $\neg b$ are a promise of $\beta_1$. Hence there is a computation of $T$ in $P$ from $s''$ to some $s'$ satisfying $\beta_1$. Concatenating these two computations, gives the desired result.

\begin{proof}

\end{proof}

\textbf{Example 7.2.1.} Figure 7.1 presents a program with three threads, inspired by the fib_bench examples of the SV-Comp concurrency benchmark. Assume that $t_1$ was selected as the main thread in $P$, and during the analysis of the corresponding sequential program $P_1$, an environment query $\text{Reach}_{\{(t_2, t_3)\}}(\alpha_1, \beta_1)$ was initiated, with $\alpha_1 \triangleq (a == 1 \land b == 1 \land c == 1)$ and $\beta_1 \triangleq (a + 2b > 15)$. Figure 7.2 presents the sequential program $P_2$ constructed to answer the query. The $\text{try}_\text{start}$ havoccs both $a$, which is written by $t_1$ and $c$, which is written by $t_3$ (and $\text{pc}_{t_3}$ as well), as it should abstract any prefix in $P$. The $\text{env}_{\text{move}}$ function, on the other hand, only abstracts the next threads in the hierarchy, i.e., $t_3$. Hence, it need not havoc $a$.

Note that there is no computation of $T = \{t_2, t_3\}$ from $\alpha_1$ to $\beta_1$. The variable $a$ is not written by the threads in $T$, hence it keeps the value of 1 in any computation of $T$ starting from $\alpha_1$. Therefore, $c$ is bounded by 3, and $b$ is bounded by 7, making $\beta_1 \triangleq (a + 2b > 15)$ unreachable.

Nevertheless, there exists a violating computation in $P_2$. The reason is of course the abstraction of $c$. Assume that the first $\text{try}_\text{start}$ in Line 28 is chosen as the activation $\text{try}_\text{start}$. The following $\text{env}_{\text{move}}$ havoccs $c$, and can set it to an arbitrary high value. This value can then be added to $b$ in Line 32, leading to the violation of the assertion in Line 35. This violating computation will now be examined by Algorithm 7.1, and eventually eliminated, after an appropriate refinement of $P_2$’s $\text{env}_{\text{move}}$ function.

66
7.3 Extending Assertions to all Threads of $P$

The next paragraph briefly explains how our technique can support assertions appearing everywhere in $P$, and not only in $t_M$. Since our sequential program $P_M$ for $t_M$ uses explicit variables to address the pc of other threads, we can use these variables to address assertions from other threads as well. Assume that an `assert(b)` appears at some label $l$ of a thread $t_i \neq t_M$. We can equip $P_M$ with a new assertion of the form `assert((pc_i = l) \Rightarrow b)`. Note that this assertion should hold regardless of the program location in $t_M$. That is, the environment of $t_M$ should never be able to reach $[(pc_i = l) \land \neg b]$, from every label of $t_M$. To express this, we can simply add this assertion at the end of the `env_move` function of $P_M$. Thus, this assertion will be handled as if it was an original assertion of $t_M$, appearing after every cut-point, and simply referring to $pc_i$ as well.

Another important observation is that environment queries issued by the algorithm, only
consider computations between cut-point states. If \( l \) is not a cut-point of \( t_i \), clearly there is no reachable computation of the environment ending at a cut-point state \( s \) s.t. \( s \Vdash (pc_i = l) \). Thus, we need to add the labels of all assertions to the sets of cut-points.
Chapter 8

Optimizations

The following chapter describes a list of optimizations used by our tool.

8.1 General Optimization

Reusing Counterexamples  This optimization is used when \( \text{Reach}_{(E)}(\alpha, \beta) \) returned \( \psi \neq \text{FALSE} \), and a new assertion \( \text{assert}(\neg \psi) \) was added right before an \text{env.move} call (case (5) of Section 4.5). In that case, instead of initiating another call to our model-checker, as described in Algorithm 4.1, we can reuse the counterexample of the previous iteration, including all of its commands up to (excluding) the last \text{env.move} call, and adding the new assertion \( \text{assert}(\neg \psi) \) at the end.

In this case, upon encountering a violating path, we try to validate all uses of \text{env.move} along that path. The main advantage of this optimization is that it avoids calling the model checker (which is the most expensive part of Algorithm 4.1). The main disadvantage is that the newly composed counterexample is not constructed by a model checker, as opposed to the non-optimized case, and hence may not be feasible, even in \( P_M \). The reason for that is that the computed postcondition \( \alpha \) only over-approximates the reachable states before the \text{env.move} call. Hence, though \( \psi \) is guaranteed to intersect \( \alpha \), it may not intersect any reachable state of \( P_M \).

Therefore, in order to make sure we only return sound answers, we only apply this optimization when there is at least one \text{env.move} in the composed counterexample. Since we only return "Unsafe" upon encountering a counterexample with no \text{env.move} calls, this ensures that the model checker was actually used to verify the feasibility of the counterexample. Note that despite being possibly unreachable in \( P_M \), the new assertions are still promises of error, hence any additional assertions added by analyzing the (possibly infeasible) counterexamples leading to them, are also promises of error.

Skipping Assertions in Weakest Precondition  Our technique includes weakest precondition computations both in \( P_M \) and in \( P_E \). In both cases, we compute a weakest precondition \( \beta = \text{wp}(\hat{\pi}, \neg b) \) for some path \( \hat{\pi} \), ending at an assertion \( \text{assert}(b) \). If the path \( \hat{\pi} \) contains other
assertions, that may either be original assertions of $P$, new assertions added by our algorithm, or assertions placed to answer an environment query, we can ignore them while computing the weakest precondition. Thus, the obtained formula represents more states.

$\beta = wp(\delta, \neg b)$ has the property that for every states $\hat{s}$ satisfying $\beta$, there exists a computation $\hat{\rho}$ from $\hat{s}$, whose path is $\hat{\pi}$, reaching $\neg b$, and thus violating the assertion. However, the important part of this property is the reachability of an error from $\hat{s}$. We only use $\hat{\pi}$, as this was the witness that assured us the reachability of that error. By ignoring other assertions in $\pi$, we obtain a more general formula $\beta'$. $\beta'$ has the property that for every state $\hat{s}$ satisfying $\beta'$, there exists a computation $\hat{\rho}$ whose path is either $\hat{\pi}$, in which case it reaches $\neg b$, or some prefix of $\hat{\pi}$, ending at another violated assertion.

**Example 8.1.1.** Consider the code below as an example, and assume we wish to compute the weakest precondition from the violation of the assertion in Line 5, (i.e., for the condition $a > 1$), for the path $\hat{\pi} = 3, 4$. This will later be sent to an environment query.

```plaintext
... 
1 env_move();
2 assert(b > 0);
3 a := 1;
4 assert(a <= 1);
```

Without the optimization, the computed condition is $\beta \equiv (a > 0 \land b > 0)$. This condition indeed ensures the violation of the second assertion. Thus, the following environment query will search for computations ending in both $a > 0$ and $b > 0$. However, if we ignore the `assert` command in Line 3, we get $\beta' \equiv (a > 0)$. This guarantees the violation of one of the assertions, but not necessarily the latter. We can then apply a more general environment query, which searches for computations reaching $a > 0$ only.

### 8.2 Optimizations for Generalizing Environment Information

The following three optimizations can be motivated by the following scenario. Assume the main thread learns new information about the environment, of the form $if \ (\alpha(W_\text{old})) assume(\neg \beta)$. The relevant formulas $\alpha$ and $\beta$ may refer to several variables that are used only by $t_M$. Since the environment query only considers computations of $t_E$, there exists some property of the environment alone, which prevents the transition from $\alpha$ to $\beta$. Our goal is to identify such properties.

Let $C$ be the set of variables of $t_M$ that are not written by $t_E$ and appear in $\alpha$ and $\beta$. Those variables are in fact constant in the context of $t_E$. Thus, by the absence of computations of $t_E$ from $\alpha$ to $\beta$ we can conclude that for every assignment to the (unchanged) variables in $C$, there is no computation from $\alpha$ to $\beta$.

This quantified property is of course a property of $t_E$ alone, since it is defined only over variables of $t_E$ (it may contain variables shared between $t_E$ and $t_M$, but none that are local to $t_M$). However, our representation does not allow us to express quantified predicates.
Thus, the following three optimizations all try to eliminate variables of $t_M$ that appear in $\alpha$ and $\beta$, without applying quantification. The first optimization is applied before the query is made (when we do not know yet whether indeed there are no computations from $\alpha$ and $\beta$), and the other two are applied after the query returned $\psi = \text{FALSE}$.

Using Query Invariants  A query invariant is a condition that appears in $\alpha$ or $\beta$ as a conjunct and consists only of variables not written by $t_E$. Clearly, any computation from $\alpha$ to $\beta$ (if exists) must satisfy the query invariant along all states in the computation, and in particular in the first and last states (satisfying $\alpha$ and $\beta$ resp.). Hence, query invariants may be used to simplify $\alpha$ and $\beta$.

If $\alpha$ or $\beta$ contains a query invariant $p$, our optimization first constructs a simplification $\alpha'$ and $\beta'$ of $\alpha$ and $\beta$ w.r.t. $p$. Then an optimized environment query $\text{Reach}(t_E)(\alpha',\beta')$ is sent to $P_E$ (see (3) in Section 4.5).

For example, if $\alpha = (i == 1 \land j > 0)$, and $i$ is not written by $t_E$, then $(i == 1)$ is a query invariant. If $\beta = (i + j > 5)$, then we can use the query invariant to obtain $\beta' = (j > 4)$. If $i$ is also not read by $t_E$, then the predicate $i == 1$ can now be omitted entirely from $\alpha$. With the optimized query, instead of learning that there is no computation from $\alpha = (i == 1 \land j > 0)$ to $\beta = (i + j > 5)$, we may learn that there is no computation from $\alpha' = (j > 0)$ to $\beta' = (j > 4)$. This property is (a) much stronger, and (b) only refers to variables of $t_E$.

Query invariants may be useful not only for the case of constant variables. For example, if $\beta$ has a conjunct $p$ as a query invariant, and $\alpha$ contains a sub-formula of the form $q_1 \lor q_2$, s.t. $p \Rightarrow \neg q_1$, then this sub-formula can be simplified to $q_2$.

The optimization described above is useful for the case where $\text{Reach}(t_E)(\alpha',\beta')$ returned $\psi = \text{FALSE}$, since the nonexistence of reachable computations of $t_E$ from $\alpha'$ to $\beta'$ implies that there are no such computations from $\alpha$ to $\beta$ as well. However, if $\psi' = \text{Reach}(t_E)(\alpha',\beta') \neq \text{FALSE}$, we cannot simply add $\text{assert}(\neg \psi')$ as before. The reason is that query invariants are only guaranteed to hold, given the context ($\alpha$ and $\beta$) of the query. More specifically, the computed $\psi'$ only guarantees that we can reach $\beta'$. But $\beta'$ does not necessarily lead to $\beta$. Reaching $\beta'$ guarantees that we reach $\beta$ only for the specific $\alpha$ (and the query invariants) used for this query. Since we might reach the env_move that initiated the query using different computations in which $\alpha$ does not hold, it is unsound to add $\text{assert}(\neg \psi')$ before the env_move call. To resolve this, if a computation from $\alpha'$ to $\beta'$ was found in $P_E$, the returned formula $\psi$ is computed as the weakest precondition from the original $\beta$, and not from $\beta'$.

For example, if we learn that there exists a computation from $j > 0$ to $j > 4$ (using the example above, where $\beta = (i + j > 5)$ is an error), and compute some weakest precondition $\psi'$ ensuring that we can reach $j > 4$, it will be unsound to add $\text{assert}(\neg \psi')$ before the env_move call. This is because $j > 4$ implies the error only in case $i == 1$. If we reach the same env_move call with $i == 0$, reaching $j > 4$ does not guarantee that we can reach the original $\beta$. 

71
Constants Replacement The next optimization can be used when $\alpha$ contains a conjunct of the form $v == p(U)$, where $v$ is not written by $t_E$, and $p$ is a term over some other variables $U$ (that might be written by $t_E$). The main observation here is that since $v$ is unchanged during computations of $t_E$, the value of $v$ at the end of the computation still equals to $p(U_{\text{old}})$, where $U_{\text{old}}$ represents the value of the variables in $U$ at the start of the computation.

Recall that the standard refinement of $\text{env\_move}$ uses a statement of the form $\text{if (} \alpha(W_{\text{old}}) \text{) assume(} \neg \beta \text{)}$. That is, we already have a method for arguing about variables at the beginning of the computation, but it was typically used just for $\alpha$. In this case, we can omit the predicate $v == p(U)$ from $\alpha$, and simply replace every other occurrence of $v$ (in $\alpha$ or $\beta$) with $p(U_{\text{old}})$.

Searching for Inductive Environment Properties A useful type of property is an inductive environment property. An inductive environment property is a condition $\gamma$, defined only over variables written by $t_E$, such that there is no reachable computation of $t_E$ from $\gamma$ to $\neg \gamma$. Recall that when a query $\text{Reach}(t_E)(\alpha, \beta)$ returns FALSE, we learn that there is no reachable computation of $t_E$ from $\alpha$ to $\beta$. We try to utilize such results to proactively search for additional relevant information, in the form of inductive environment properties. However, checking whether some $\gamma$ is an inductive environment property (by computing $\text{Reach}(t_E)(\gamma, \neg \gamma)$) is computationally expensive, as it involves calling the model checker. Hence, this check is only applied in specific cases, as described by the two heuristics below.

The first heuristic is applied if after the two optimizations above, the resulted $\alpha'$ and $\beta'$ still contain variables not written by $t_E$. In a typical case, $\alpha$ is a conjunction of several conditions. It might be the case that $\alpha$ itself contains variables not written by $t_E$, but has some conjuncts which are only over variables that are written by $t_E$. For each such conjunct $\gamma$, we check if $\gamma$ is an inductive environment property. We also save $\gamma$, so that we will not check it again if it reappears as a conjunct in some future $\alpha$.

The second heuristic is applied when $\alpha = \text{TRUE}$, and $\beta$ is a conjunction. When $\alpha = \text{TRUE}$, it means that $\neg \beta$ is a global invariant. Global invariants are typically strong properties which incorporate some fundamental property of the program. When combined with generalization, as described in Section 4.6, none of the disjuncts forming $\neg \beta$ can also be global invariant on their own (otherwise $\beta$ would have been generalized). Nevertheless, they might be inductive environment properties. Hence, we check whether any of the disjuncts of $\neg \beta$ defined only over variables written by $t_E$, are indeed inductive environment properties.

In both cases, if an inductive environment property $\gamma$ was found, we use it to further refine the $\text{env\_move}$ function, by adding $\text{if (} \gamma(W_{\text{old}}) \text{) assume(} \gamma \text{)}, additionally to the } \text{if (} \alpha(W_{\text{old}}) \text{) assume(} \neg \beta \text{)} \text{ statement we always add when environment queries return FALSE.}$

8.3 Multiple Threads Optimizations

The optimizations appearing in this section are relevant for the case where $P$ has more than two threads, and environment queries are answered by Algorithm 7.1.
Preserving Environment Information. Typically, during the run of Algorithm 4.1, multiple environment queries \( \text{Reach}_{(T)}(\alpha_1, \beta_1) \) are made. When \( P \) consists of two threads, each such query is answered using a single model checker call, on a sequential program \( P_E \) (see Chapter 6). However, when the environment \( T \) of the main thread \( t_M \) consists of more than one thread, environment queries are answered by Algorithm 7.1. During this algorithm, a sequential program \( P_M' \) is constructed and refined according to additional inner environment queries \( \text{Reach}_{(T')}(\alpha_2, \beta_2) \) for a smaller environment \( T' \). The information learned about \( T' \) can also be relevant for future queries. Hence, \( P_M' \) does not have to be reconstructed from the beginning for every new environment query, and it can reuse information learned about \( T' \).

We now discuss which information can remain in \( P_M' \) and serve future queries (made by \( P_M \)). Assume that \( P_M' \) was constructed once for answering an environment query \( \text{Reach}_{(T)}(\alpha_1, \beta_1) \) from \( P_M \), and later reconstructed to answer another query \( \text{Reach}_{(T)}(\alpha_1', \beta_1') \). Any new assertion added to \( P_M' \) during the process of answering the first query, is in fact a “promise of \( \beta_1' \)”. That is, if the assertion is violated, it means that \( \text{start} \land \beta_1 \) can be reached. In the general case, \( \beta_1' \neq \beta_1 \) and hence this information does not guarantee the reachability of \( \beta_1' \), needed by the query \( \text{Reach}_{(T)}(\alpha_1', \beta_1') \). Thus, it should be removed from \( P_M' \).

However, the information which appears inside the \text{env.move}' function, represents the absence of computations of \( T' \) from some \( \alpha_2 \) to some \( \beta_2 \). Since this is a property of \( T' \), it remains correct regardless of the conditions \( (\alpha_1, \beta_1) \) given by the query from \( P_M \). Hence, information gathered within the \text{env.move}' of \( P_M' \) can be accumulated and serve future queries.

Quick Generalization. The generalization described in Section 4.6 is by itself a useful optimization. The difference in runtime between solving with and without generalizations can sometimes be in orders of magnitude. A common interesting pattern observed during our experiments, is that calling our tool with generalization, results in more calls to the model-checker than calling it without generalization, for the same program \( P \). However, the overall verification time for the generalization case is still shorter. The reason for this pattern is that many of the model-checker calls are performed as part of the generalization process, thus increasing the number of calls. Nevertheless, the sequential programs obtained after generalization tend to be simpler than those obtained without. This results in a much shorter runtime per each model checker call.

Notwithstanding the above, consider the case of more than two threads, as described in Chapter 7. Let \( P_M \) be the sequential program of the main thread, \( t_M \), \( P_M' \) the sequential program of the main thread of the environment, \( t_M' \), and \( T' \) the rest of the threads of \( P \). Assume also that \( \text{Reach}_{(T)}(\alpha_1, \beta_1) = \text{FALSE} \) for some environment query, originated from the analysis of \( P_M \). Essentially, generalization is performed by guessing conditions \( \alpha_1', \beta_1' \) s.t. \( \alpha_1 \Rightarrow \alpha_1' \) and \( \beta_1 \Rightarrow \beta_1' \), and checking whether still \( \text{Reach}_{(T)}(\alpha_1', \beta_1') = \text{FALSE} \). If this query is answered by Algorithm 7.1, it may initiate additional inner environment queries \( \text{Reach}_{(T')}(\alpha_2, \beta_2) \). These queries, in turn, might trigger another generalization, and so forth. When the number of threads increases, this overhead becomes significant and compromises the benefits of generalization.

We therefore suggest an intermediate approach. Recall that if \( \text{Reach}_{(T)}(\alpha_1, \beta_1) \) returned
FALSE, it means that \( P'_M \) already has enough information to refute the transition from \( \alpha_1 \) to \( \beta_1 \). We can therefore check if \( P'_M \), by itself, can also refute the transition from \( \alpha'_1 \) to \( \beta'_1 \). If so, we can generalize \((\alpha_1, \beta_1)\) to \((\alpha'_1, \beta'_1)\). Otherwise, this generalization call fails and we retreat to \((\alpha_1, \beta_1)\) (and might try another generalization). Note that in the latter case, it still might be the case that \( \text{Reach}_{(T)}(\alpha'_1, \beta'_1) \) would have returned FALSE, but this would have required additional refinements of \( P'_M \). In practice, we reduce the number of model-checker calls allowed per generalization attempt \( \text{Reach}_{(T)}(\alpha'_1, \beta'_1) = \text{FALSE} \) to one. This approach often gives a substantial share of the benefits of generalization, while significantly reducing the overhead.
Chapter 9

Experimental Results

Setup  We implemented our algorithm in a prototype tool called CoMuS. The implementation is written in Python 3.5, uses pycparser [3] for parsing and transforming C programs, uses SeaHorn [22] for sequential model checking, and uses Z3 [10] to check implications of formulas for some of the optimizations described in Chapter 8. CoMuS currently supports only a subset of the syntax of C (as appears in the preliminaries). It does not perform alias analysis and hence does not handle pointers. It also does not support dynamic thread creations, although we support any fixed number of threads.

We compare CoMuS with Threader [36], VVT [19] and UL-CSeq [29], the last two being the top scoring model checkers on the concurrency benchmark among sound unbounded tools in SVCOMP’16 and SVCOMP’17 (resp.). On the concurrency benchmark, VVT was 4th overall in SVCOMP’16, and UL-CSeq was 8th overall in SVCOMP’17. Threader performs modular verification, abstracts each thread separately and uses an interference abstraction for each pair of threads. UL-CSeq performs a reduction to a single non-deterministic sequential program. The program is then passed to a sequential model-checker. We used UL-CSeq in its default setting, with CPA-Checker [4] as a backend. VVT combines bounded model checking for bug finding with an IC3 [6] based method for full verification.

We ran the experiments on a x86-64 Linux machine, running Ubuntu 16.04 (Xenial) using Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz with 8GB of RAM.

Experiments  We evaluated the tools using three experiments. One compares the four tools on concurrent programs with a clear hierarchy. The second compares syntactically similar programs with and without hierarchal structure to evaluate the effect of the structure on the verification time. The last one looked at general concurrent programs.

Hierarchically Structured Programs  For the first experiment, we used three concurrent dynamic-programming algorithms: Sum-Matrix, Pascal-Triangle and Longest-Increasing-Subsequence.

\footnote{The same benchmark was used for unbounded sound tools and tools which perform unsound bounded reductions. Bounded tools are typically ranked higher. Our method is unbounded and is able to provide proofs, hence we find the selected tools more suitable for comparison.}
The Sum-Matrix programs receive a matrix $A$ as input. For every pair of indexes $(i, j)$, it computes the sum of all elements $A[k, l]$, where $k \geq i$ and $l \geq j$. In their concurrent version, each thread is responsible for the computation of a single row. The Pascal-Triangle programs compute all the binomial coefficients up to a given bound. Each thread computes one row of the triangle, where each element in the row depends on two results of the previous row. The Longest-Increasing-Subsequence programs receive an array, and compute for each index $i$, the length of the longest increasing subsequence that ends at index $i$. Each thread is responsible for computing the result for a given index of the array, depending on the result of all prefixes.

These algorithms have a natural definition for any finite number of threads. Typically, the verification becomes harder as the number of threads increases. For evaluation, we used programs with an increasing number of threads, and checked the influence of the number on the different tools. For each instance, we use both a safe and an unsafe version. Both versions differ from each other either only by a change of specification, or by a slight modification that introduces a bug.

The chosen programs have two meaningful characteristics: (i) They exhibit non-trivial concurrency. This means that each thread performs a series of computations, and it can advance when the data for each computation is ready, without waiting for the threads it depends on to complete. Consider the Sum-Matrix problem as an example. Assume thread $t_i$ needs to compute the result at some location $(i, j)$, and that each row is computed backwards (from the last cell to the first). The computation exploits the results of thread $t_{i+1}$. Thread $t_i$ needs to wait for thread $t_{i+1}$ to compute the result for location $(i + 1, j)$. However, $t_i$ does not wait for $t_{i+1}$ to terminate, as it can compute the cell $(i, j)$, while $t_{i+1}$ continues to compute $(i + 1, j - 1)$. (ii) Their data flow graph has a clear chain structure. That is, the threads can be ordered in a chain hierarchy, and each thread only requires information computed by its immediate successor.

Figure 9.1 summarizes the results for these programs. The timeout was set to 3600 seconds. The code of the programs is available at https://tinyurl.com/tacascomus. We include in the table also our running example, the Peterson algorithm.

The results demonstrate a clear advantage for CoMuS for verification (i.e., for safe programs) as the number of threads increases. For falsification, CoMuS is outperformed by VVT’s bounded method. However, it still performs significantly better than the two other tools when the number of threads grows.

**Hierarchical vs. Non-hierarchical Programs** The programs used for this evaluation are variants of the “fib_bench” examples of the SV-COMP concurrency benchmark. We compare programs in which the data flow graph has a ring topology, vs. programs in which it has a chain topology. Figure 7.1 presents such a program with three threads and a ring topology. For the ring case, consider a program with threads $t_0, \ldots, t_{n-1}$ and variables $v_0, \ldots, v_{n-1}$. Each thread $t_i$ runs in a loop, and iteratively performs $v_i = v_{(i+1) \mod n(i)}$. The checked property is that $v_0$ does not surpass an upper bound. The chain case is identical except that for the last thread, $t_{n-1}$, we break the chain and perform $v_{n-1} = v_0$ instead of $v_{n-1} = v_0$. Figure 9.2 presents the results of this comparison. All the programs in the table are safe and with two loop iterations. The
timeout was set to 1200 seconds.

For the ring case, all tools fail to verify programs with \( \geq 4 \) threads. Threader presents similar results for both ring and chain topologies. VVT benefits from the less dependent chain topology, but still timeouts on more than three threads. CoMuS, on the other hand, is designed to exploit hierarchy, and benefits significantly from the chain topology, where it verifies all instances. UL-CSeq is excluded from the table as it performs sub-optimally for “fib_bench” examples (both in our experiments and in the SV-COMP results).

**General Concurrent Programs**  We also evaluated the tools on a partial subset of the SV-COMP concurrency benchmark, whose code is supported by CoMuS. Typically, on these runs CoMuS was outperformed by the other tools.

We conclude that even though our method can be applied to programs without a clear hierarchical structure, it is particularly beneficial for programs in which the hierarchy is inherent.
Figure 9.2: Run times [secs] for fib_bench programs with ring topology vs. chain topology.

<table>
<thead>
<tr>
<th>threads</th>
<th>Ring</th>
<th></th>
<th></th>
<th></th>
<th>Chain</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>CoMus</td>
<td>VVT</td>
<td>Time</td>
<td>Seahorn Calls</td>
<td>CoMus</td>
<td>VVT</td>
<td>Time</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>4.03</td>
<td>53.89</td>
<td>6.23</td>
<td>27</td>
<td>2.84</td>
<td>9.67</td>
<td>1.98</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>681.57</td>
<td>T\O</td>
<td>111.32</td>
<td>272</td>
<td>615.87</td>
<td>387.11</td>
<td>7.31</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>T\O</td>
<td>T\O</td>
<td>T\O</td>
<td>-</td>
<td>T\O</td>
<td>T\O</td>
<td>20.21</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>T\O</td>
<td>T\O</td>
<td>T\O</td>
<td>-</td>
<td>T\O</td>
<td>T\O</td>
<td>53.71</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>T\O</td>
<td>T\O</td>
<td>T\O</td>
<td>-</td>
<td>T\O</td>
<td>T\O</td>
<td>123.53</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>T\O</td>
<td>T\O</td>
<td>T\O</td>
<td>-</td>
<td>T\O</td>
<td>T\O</td>
<td>307.3</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>T\O</td>
<td>T\O</td>
<td>T\O</td>
<td>-</td>
<td>T\O</td>
<td>T\O</td>
<td>680.99</td>
</tr>
</tbody>
</table>
Chapter 10

Conclusion and Future Work

10.1 Conclusion

In this work we develop an automatic, hierarchical and modular method for proving or disproving safety of concurrent programs by exploiting model checking for sequential programs. The technique chooses one “main thread” and constructs a sequential program based on the main thread and an abstraction of all other threads. It then uses a model checker on this sequential program, and tries to determine the safety of the original program based on the model checker’s result.

If the abstraction is not sufficient for a conclusive answer, we generate “environment queries” to check whether the other threads can perform computations that would “help” the main thread to reach a violation. The queries are answered by a recursive application of the same approach, i.e., by creating a sequential program based on one of the environment threads, with an abstraction of the rest. The number of abstracted threads decreases by one with every step of the recursion, until we reach a single environment thread. In that case, environment queries are answered by a single call to a model checker.

The method can use any off-the-shelf model checker, thus benefiting from possible developments in the field of sequential verification. The minimal requirement from the model checker is to determine the safety of sequential programs, and to provide counterexamples in the form of a violating path when the program is unsafe.

The method can handle infinite-state programs, assuming such programs are supported by the sequential model checker. It is sound and unbounded. We proved that our method is sound, ensured to make progress, and terminates in the case of finite state programs. We implemented our approach in a prototype tool called CoMuS, which compares favorably with top scoring model checkers on programs which have a hierarchical structure.

10.2 Future Work

Getting Inside the Model Checker Although a key aspect of the designed reduction is to be independent of the chosen model-checker, there can be benefits for implementing the method
“inside” the model-checker. For example, when a model checker determines that a program is
safe, it typically learns certain invariants at different locations of the program. If the model-
checker is called to answer an environment query, such invariants can be used to learn properties
of the environment, that are more general than the specific question asked by the original query.
Thus, it can replace or diminish the need of our own generalization technique (which requires
additional model-checker calls). Further, since we generate a series of sequential programs that
are syntactically similar, it is to be expected that incremental techniques could yield a significant
speedup. In the future, we intend to exploit internal information gathered by the sequential
model-checker (e.g., SeaHorn) to further speedup our results.

**Extension to Liveness**  An interesting view of our method is the question of whether it can
be applied to liveness properties as well. The main difference between liveness and safety
properties, is that liveness counterexamples are infinite. Given such a counterexample, possible
research directions are to try and compute a promise of error at some point along the infinite
counterexample (i.e., a property that ensures that the program will reach the infinite computation
violating the liveness property), or to reverse the counterexample analysis and try to compute a
strongest postcondition and an infinite precondition. [5] showed how liveness checking problems
can be transformed to safety checking problems for finite systems. However, it is remains an
open question if these ideas (or the other directions mentioned above) can be used to extend our
method.
Bibliography


The context switch is a common problem in multi-threaded systems. In the CoMuS framework, this is addressed by developing a new model for context switching that takes into account the dynamic behavior of the system. This model is then implemented in a tool called CoMuS-Test, which allows for the automatic verification of context switching behavior.

In our experiments, we found that CoMuS-Test was able to accurately model context switching behavior in a variety of scenarios. This includes situations where context switching is triggered by external events, as well as situations where it occurs as part of the normal execution flow.

Overall, we believe that our approach to modeling context switching behavior provides a valuable tool for the verification of multi-threaded systems.
לא ניתןقرأ את התוכן המוצג בתמונה.
קצירró מחרת

אמות חכויות מקבליות היא בעינו קשה بوית. בנסף לאנתרופים המובנים או אימוז חכויות רימי גרובי
נוסח הצירים הלא חסרים מסמר בה. מבנה המילים ורמי הלחם של
בסטוריים ממסתיים. עבר חוכיות אצלי, מבטבekte לוחז ובעת המבנה המדויק, שחקן
יאומי. ולא שולב. חוכית של התכנית כלול אל תוך החלק הקבוצת חוכית כшли אוח
מקוית עברה עד החוכיות. אך, החוכית הבחנה עברה את החוכיות ודוותי יעד פסיפס
לבר.

האינטראקציה שלל_renderer.
המחק בוצע בהנחייתם של פרופסורים ארנה גרימברג וד"ר שחרת שוהם, בפקולטה למדעי המחשב.

תודה

בראש ובראשונה, אני רוצה להודות למנהלה של הטכניון, פרופסור ארנה גרימברג, על הנחיותיו ובמקומיו המ�מים והומריים שהעלו את היכולת שלי להשתתף במערך התוכניות והפעילות החברתית של חוג מחקר. כך שהיינו כל שבועיים עם פרוור רעיונות חדשים וחשבון חידושים, והם הביאוuchi את השפעה על התוכנית וה옿ה העזה וrideיה פנימה.

ערת ל thereof, ואת חותין הלימודים הולכת תמיכה. זה היא בד קמד גוזי עליון, לעובד النظر והיוצרしてください.

אני מודה להנחיית הנוספת שלי, ד"ר שחרת שוהם, על חלקה העצום בעבודה זו. תודה על המעודדות שלך, ועל היכולת להפוך כל פגישה לפוריה עם רעיונות חדשים וחשבון חידושים. זה תמיד היה ניתן עבורי, גם בת"א וגם בטכניון. את מנחה נפלאה ואדם נפלא, ואני אסיר תודה על עזרתך והיוותהך.

אני רוצה להודות לparents של מה הוספתי, ד"ר שחרת שוהם, על אהבתם ותמיכתם האינסופית. תודה שלימדתם אותי לעון גבוה, ועל כך שהייתי שם בשבילי בכל שלב חיי.

לבסוף, אני רוצה להודות לחברה שלי ורוניקה. ליווה אותי במהלך התקופה הזו גם בימים הטובים יותר בהם הצלחתי להתקדם, וגם בימים טובים פחות מתסכלים. תמיד תמוכת עליי ועזרת לי להמשיך, ואני אסיר תודה לך על זה.

אני מודה לטכניון על התמיכה הכספית והdìバラבשת ומכוונים.

אני מודה לטכניון על התמיכה הכספית והdìバラבשת וברק שוחם,بقוקית למדעי המחשב.
אימוט מודולרי של תכנון מכביילית
באמצעות בדיקת מודל סדרתית

תירבע על מחקר

לשמ מליי חלקי של הדרישות לקבלי התואר
מגייסרפּתו למדעי במדעי המחשב

דנ רסין

הוגש לסנט הטכניון - מכון טכנולוגי לישראל
כסלו התשע"ח
חיפה
נובמבר 2017
אימות מודולים של תכנויות מקבליות
באמצעית בדיקה מודל סדרתית

דן רסין