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 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

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 `env_move`, which is called by t_M whenever a context switch should be considered. Initially, `env_move` havocs 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 `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 `env_move` function is refined to exclude it. Eventually, `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 **Co**ncurrent to **Mu**ltiple **S**equential (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, $t_1, t_2, \dots, t_k$, 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 $t_2, \dots, t_k$; 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 $l^{init} \in L$. Each node l is associated with a command $c \in cmds$, denoted $cmd(l)$, which can be an assignment or an if command, as well as `havoc`, `assume` and `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 $\texttt{pc} \notin V$, ranging over L, indicates the program location. A *state* s of the sequential program P, is a pair (l, σ) where $l \in L$ is the value of `pc` and σ 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 σ over V and a set of variables U, we denote by $\sigma|_U$ the restriction of σ 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 $\mathit{Vars}(c)$ the set of variables appearing in c. We denote $\sigma|_c = \sigma|_{\mathit{Vars}(c)}$.

A program may include an *initialization condition* in the form of a valuation σ_{init} of a set of variables $U \subseteq V$. We denote by $\sigma_{init}(v)$ the *initial value* of a variable $v \in U$. The set of *initial states* consists of all states (l^{init}, σ) where $\sigma|_U = \sigma_{init}$. We denote by ϕ_{init} the *initialization formula* $\phi_{init} \triangleq \bigwedge_{v \in U} [v = \sigma_{init}(v)]$. It is possible to express more complex initial conditions, e.g. $(v_1 > 0 \land v_2 + v3 = 7)$, via explicit assume commands in the code, as described below.

For a state $s = (l, \sigma)$, let $cmd(s)$=$cmd(l)$. We denote $next(s) = \{s' \mid s'$ can be obtained from s using $cmd(s)\}$[1]. This set is defined according to the command $cmd(s)$. In particular, $s' \in next(s)$ implies that $(l(s), l(s'))$ is an edge in P. The definition of $next(s)$ for assignments and if commands is standard. A 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 condition

[1]`havoc` commands are non-deterministic, hence s' is not unique.

b does not hold. It does nothing, otherwise. Formally, if $s = (l, \sigma)$ and $cmd(s)$=`assume(b)`, then $\sigma \models b \Rightarrow next(s) = \{(l', \sigma)\}$ where l' is the label of the next command to be executed after the `assume` command (i.e., l' is the single label such that $(l, l') \in E$), and $\sigma \not\models b \Rightarrow next(s) = \emptyset$. An `assert(b)` command moves to the error state if b is violated, and does nothing otherwise. Formally, if $s = (l, \sigma)$ and $cmd(s)$=`assert(b)` for some condition b, then $\sigma \models b \Rightarrow next(s) = \{(l', \sigma)\}$ where l' is the label of the next command to be executed, and $\sigma \not\models b \Rightarrow next(s) = \{\epsilon\}$.

A *computation* ρ 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}: $s_{i+1} \in next(s_i)$. ρ is an *initial computation* in P if it starts from an initial state. ρ is a *reachhable computation* in P if there exists an initial computation $\rho^{'}$ for which ρ 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 $pre(e, q)$, is a condition p such that for every state s, if $\sigma(s) \models p$ and $l(s) = l$ then there exists $s' \in next(s)$ s.t. $\sigma(s') \models q$ and $l(s') = l'$ [2]. 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., π) 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 $wp(e, q)$ (resp., $wp(\pi, q)$).

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

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

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 $\texttt{pc}_i$. 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 σ is a valuation of V and $\bar{l} = (l_1, \ldots, l_m)$ with l_i being

[2]Note that our definition of a precondition does not require all the successors to satisfy q.

the value of $\mathtt{pc}_i$ for every thread t_i. We denote $l(s, t_i) = l_i$. We also assume one common error state ϵ. An initialization condition σ_{init} and initialization formula ϕ_{init} are defined as in the sequential case. The set of *initial states* consists of all states $(\bar{l}^{init}, \sigma)$ where $\sigma \models \phi_{init}$ and $\bar{l}^{init} = (l_1^{init}, \ldots, l_m^{init})$ 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 non-deterministically. We consider a sequentially consistent semantics in which the effect of a single command on the memory is immediate. For $s = (\bar{l}, \sigma)$, let $cmd(s, t_i)$ denote the command of thread t_i at label l_i. We denote $next(s, t_i) = \{s' \mid s'$ can be obtained from s after t_i performs $cmd(s, t_i)\}$. A *computation* ρ of the concurrent program P is a sequence $s_0 \xrightarrow{t^{(1)}} s_1 \xrightarrow{t^{(2)}} \ldots \xrightarrow{t^{(n)}} s_n$ s.t. for every two adjacent states s_i, s_{i+1}: $t^{(i+1)} \in \{t_1, \ldots, t_m\}$ and $s_{i+1} \in next(s_i, t^{(i+1)})$. We say that ρ is a computation of thread t in P if $t^{(j)} = t$ for every $1 \leq j \leq n$. Given a set of threads $T \subseteq \{t_1, \ldots, t_m\}$, we sat that ρ is a computation of T in P if $t^{(j)} \in T$ 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, `Lock(lock)` is modeled by atomic execution of `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 $v \notin V_j$ 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 `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 $s_{i-1} \xrightarrow{t^{(i)}} s_i \xrightarrow{t^{(i+1)}} s_{i+1}$ 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., $l_j^{init} \in CL_j$ 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 $\texttt{pc}_M$ and $\texttt{pc}_E$, respectively. V_M and V_E might intersect. Let $V = V_M \cup V_E$. Given a state $s = (\bar{l}, \sigma)$, we denote by $l_M(s)$ and $l_E(s)$ the values of $\texttt{pc}_M$ and $\texttt{pc}_E$ in a state s, respectively. We also denote by l_M^{init} and l_E^{init} the initial values of $\texttt{pc}_M$ and $\texttt{pc}_E$. 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 $\widehat{V_M} \supseteq V_M$ and $\widehat{V_E} \supseteq V_E$. 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, `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, α and β, and checks whether there exists a reachable computation of t_E in P from α to β. 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 ψ, which ensures that all states satisfying it can reach β using t_E only. We also require that α and ψ overlap. In order to ensure the reachability of β, the formula ψ might need to address local variables of t_E, as well as $\mathtt{pc}_E$. 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 $\psi = \mathit{FALSE}$. The formal definition follows.

Definition 3.2.1 (Environment Query). An *environment query* $\mathit{Reach}_{(t_E)}(\alpha, \beta)$ receives conditions α and β over $V \cup \{\mathtt{pc}_E\}$, and returns a formula ψ over $V \cup \{\mathtt{pc}_E\}$ 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. $s \vDash \alpha$, and (3) ends in a cut point state s' s.t. $s' \vDash \beta$, then $\psi \wedge \alpha \not\equiv \mathit{FALSE}$.

2. For every state s s.t. $s \vDash \psi$, there exists a computation (not necessarily reachable) of t_E in P from s to some s' s.t. $s' \vDash \beta$.

3. $\psi \neq \mathit{FALSE} \Rightarrow \psi \wedge \alpha \not\equiv \mathit{FALSE}$.

Observation 3.2.2.

1. We note that ψ is not required to be precise, i.e., nothing is required of states s s.t. $s \nvDash \psi$.

2. If $\psi = \mathit{FALSE}$, the first property implies that there is no reachable computation of t_E in P between cut point states from α to β.

3. If $\alpha \wedge \beta \not\equiv \mathit{FALSE}$, β is always a valid result for $\mathit{Reach}_{(t_E)}(\alpha, \beta)$, 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 `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 $\widehat{V_M} \supseteq V_M$, 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, $\widehat{V_M}$ includes all variables of t_M, i.e., $\widehat{V_M} \supseteq V_M$. $\widehat{V_M}$ may also include variables in $V_E \setminus V_M$, as well as include $\mathtt{pc}_E$ as an explicit variable. All other variables in $\widehat{V_M}$ 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 $V \cup \{\mathtt{pc}_E\}$. 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 $V \cup \{\mathtt{pc}_E\}$.

Algorithm 4.1 Algorithm MainThreadCheck

```
1:  procedure MAINTHREADCHECK($t_M$, $t_E$)
2:      $P_M$ = add env_move calls in $t_M$ and initialize env_move()
3:      while a violating path exists in $P_M$ do        // using sequential MC
4:          Let $\hat{\pi} = \hat{l}_0, \ldots, \hat{l}_{n+1}$ be a violating path.
5:          if there are no env_moves in $\hat{\pi}$ then:
6:              return "Real Violation"
7:          end if
8:          let $\hat{l}_k$ be the label of last env_move call in $\hat{\pi}$
9:          let $\hat{\pi}_{start} = \hat{l}_0, \ldots, \hat{l}_k$ and $\hat{\pi}_{end} = \hat{l}_{k+1}, \ldots, \hat{l}_n$
10:         $\beta = wp(\hat{\pi}_{end}, \neg b)$                // see (1) in Section 4.5
11:         $\alpha = post(\hat{\pi}_{start}, \hat{\phi}_{init})$          // see (2) in Section 4.5
12:         Let $\psi = Reach_{(t_E)}(\alpha, \beta)$       // environment query for $t_E$ (see Chapter 6)
13:         if $\psi$ is FALSE then
14:             Let $\alpha'$, $\beta'$ be as in (4) in Section 4.5.
15:             $P_M$ = RefineEnvMove($P_M, \alpha', \beta'$)
16:         else        // see (5) in Section 4.5
17:             Add assert($\neg\psi$) in $P_M$ at new label $\hat{l}'$, placed right before $\hat{l}_k$
18:         end if
19:     end while
20:     return "Program is Safe".
21: end procedure
```

Initialization Condition of P_M All variables in $\widehat{V_M} \cap V$ appear in P_M with the same declaration as in P, and the same (possible) initialization as in P. In addition, if $\mathtt{pc}_E \in \widehat{V_M}$, it is initialized with l_E^{init}. Formally, if P uses the initialization formula $\phi_{init} \triangleq \bigwedge_{v \in U} [v = \sigma_{init}(v)]$, then $\hat{\phi}_{init}$, the initialization formula of P_M, is $\hat{\phi}_{init} \triangleq \bigwedge_{v \in U \cap \widehat{V_M}} [v = \sigma_{init}(v)]$ if $\mathtt{pc}_E \notin \widehat{V_M}$, and $\hat{\phi}_{init} \triangleq (\bigwedge_{v \in U \cap \widehat{V_M}} [v = \sigma_{init}(v)]) \wedge (\mathtt{pc}_E = l_E^{init})$ if $\mathtt{pc}_E \in \widehat{V_M}$.

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 $cmd(l)$ =`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 $Lab(\hat{l}) = l$.

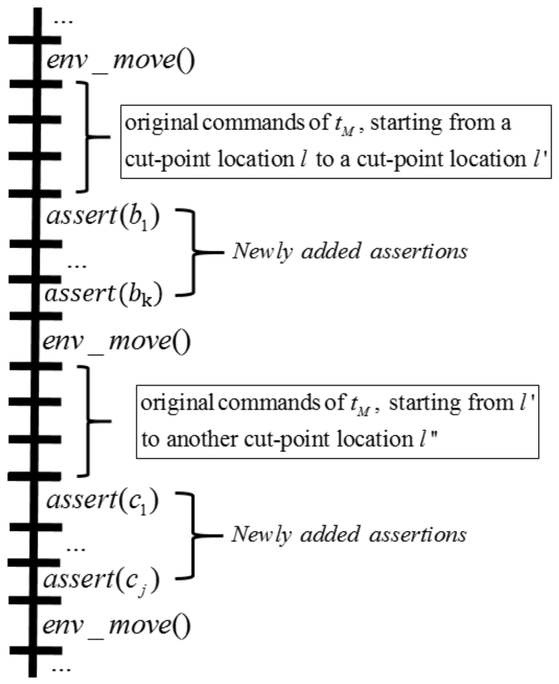

...
$env\_move()$
original commands of t_M, starting from a cut-point location l to a cut-point location l'
$assert(b_1)$
...
$assert(b_k)$
Newly added assertions
$env\_move()$
original commands of t_M, starting from l' to another cut-point location l''
$assert(c_1)$
...
$assert(c_j)$
Newly added assertions
$env\_move()$
...

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, $Lab(\hat{l}) = Lab(\hat{l}')$, where $\hat{l}'$ is the next label in P_M after the `env_move` or the assert command (resp.).

Lab is well defined, as the next label after `assert`s 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 $P0$ in Figure 4.3, which is based on $t0$ from Figure 4.2. In this example, line 10 of $P0$ with the command `turn = 1`, as well as the preceding line 9 with an `env_move` call, are both mapped to line 8 of $t0$ with the original `turn = 1` command. Thus, $Lab(\hat{9}) = Lab(\hat{10}) = 8$.

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 `env_move`. We define the set $Extend(\hat{s})$ to be the set of all states $s = (\bar{l}, \sigma)$ of P such that:

- $l_M(s) = Lab(\hat{l})$
- If $\texttt{pc}_E \in \widehat{V_M}$ then $l_E(s) = \hat{\sigma}(\texttt{pc}_E)$
- For every $v \in V$: $v \in \widehat{V_M} \Rightarrow \sigma(v) = \hat{\sigma}(v)$

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

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

Observation 4.2.3. If $s \in Extend(\hat{s})$, then for every formula γ over $\widehat{V_M} \cap (V \cup \{\texttt{pc}_E\})$, it holds that $s \vDash \gamma \iff \sigma(\hat{s}) \vDash \gamma$ [1]. This is true, since Definition 4.2.1 implies that s and $\hat{s}$ have identical values on all the variables in γ.

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 $s \in Extend(\hat{s})$.*

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

- $l_M(s) = l_M^{init}$.
- $l_E(s) = l_E^{init}$.
- For every $v \in U$: $\sigma(s)(v) = \sigma_{init}(v)$.
- For $v \in \widehat{V_M} \setminus U$: $\sigma(s)(v) = \sigma(\hat{s})(v)$.
- If $v \notin \widehat{V_M}$ and $v \notin U$: $\sigma(s)(v)$ can be chosen arbitrarily.

[1] The difference in notation (i.e., "s" vs. "$\sigma(\hat{s})$") results from the special status of the variable $\texttt{pc}_E$. 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 $s \in Extend(\hat{s})$.

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

Since $\hat{s}$ is initial, $\sigma(\hat{s}) \vDash \hat{\phi}_{init}$. Thus, if $\texttt{pc}_E \in \widehat{V_M}$, then $\sigma(\hat{s})(\texttt{pc}_E) = l_E^{init} = l_E(s)$. Similarly, if $v \in \widehat{V_M} \cap U$, $\sigma(\hat{s})(v) = \sigma_{init}(v) = \sigma(s)(v)$. For $v \in \widehat{V_M} \setminus U$, $\sigma(\hat{s})(v) = \sigma(s)(v)$ by the construction of s. Thus, $s \in Extend(\hat{s})$. □

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 `env_move` function over-approximates computations of t_E. Intuitively, the `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 `env_move` *over-approximates the computations of* t_E *in* P if for every reachable (and possibly empty) computation $\rho = s \xrightarrow{t_E} \ldots \xrightarrow{t_E} s'$ 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. $s \in Extend(\hat{s})$, and $cmd(\hat{s})$ is an `env_move` call, there exists a computation $\hat{\rho} = \hat{s} \rightarrow \cdots \rightarrow \hat{s}'$ of P_M s.t.

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

The definition implies that if t_E can move (in any finite number of steps) from a state s satisfying some condition α to a state s' satisfying some condition β, `env_move` should also allow reaching a state satisfying β, when it is called from a state satisfying α. In order to prove that `env_move` always over-approximates the computations of t_E in P, we need to consider the possible refinements and changes applied to `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 `env_move`, to representative computations of t_M in P. Lemma 4.2.7 maps computations ρ 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 ρ' of ρ, followed by an error state. This may happen, in case that Algorithm 4.1 already learned that the last state of ρ' 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 (P_M representation I)**.** *Let $\hat{\rho} = \hat{s} \rightarrow \cdots \rightarrow \hat{s}'$ 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 $s \in Extend(\hat{s})$ there exists a computation ρ of t_M in P from s to some state s' s.t. $s' \in Extend(\hat{s}')$.*

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 $s \in Extend(\hat{s})$ implies $s' \in Extend(\hat{s}')$.

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

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

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

Let c=$cmd(\hat{s}'')$. c is the command used by $\hat{\rho}$ to move from $\hat{s}''$ 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 $\mathtt{pc}_E$ in P_M, we get $\sigma(\hat{s}'')(\mathtt{pc}_E) = \sigma(\hat{s}')(\mathtt{pc}_E)$. Since $s'' \in \mathit{Extend}(\hat{s}'')$, $\mathtt{pc}_E \in \widehat{V_M}$ implies $l_E(s'') = \sigma(\hat{s}'')(\mathtt{pc}_E)$. Combining this with the definition of s', we get $l_E(s') = l_E(s'')$, and all together $l_E(s') = \sigma(\hat{s}')(\mathtt{pc}_E)$.

   It is now sufficient to show that $s' \in \mathit{next}(s'', t_M)$ (a computation of length one from s'' to s').

   Let $V_{act} = \mathit{Vars}(c)$. The command c can only change the variables in V_{act} and the possible modifications are independent of variables not in V_{act}. Since c is in P_M, $V_{act} \subseteq \widehat{V_M}$ and hence by the definition of s', $\sigma'|_{V_{act}} = \sigma(\hat{s}')|_{V_{act}}$. Since $s'' \in \mathit{Extend}(\hat{s}'')$, we also get $\sigma(s'')|_{V_{act}} = \sigma(\hat{s}'')|_{V_{act}}$. Since $\hat{s}' \in \mathit{next}(\hat{s}'')$, it means that $\sigma(\hat{s}')|_{V_{act}}$ can be obtained from $\sigma(\hat{s}'')|_{V_{act}}$ by performing c, and hence $\sigma'|_{V_{act}}$ can be obtained from $\sigma(s'')|_{V_{act}}$ by performing c. For a variable $v \notin V_{act}$, $\sigma'(v) = \sigma(s'')(v)$ (by the definition of s' if $v \notin \widehat{V_M}$, and because $\sigma(\hat{s}')(v) = \sigma(\hat{s}'')(v)$ for $v \in \widehat{V_M} \setminus V_{act}$, as c cannot change variables outside V_{act}). Hence, σ' 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 $\sigma(s'')|_{V_{act}}$. 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_{act}). Since $s'' \in \mathit{Extend}(\hat{s}'')$, it holds that $l_M(s'') = \mathit{Lab}(l(\hat{s}''))$ and $\sigma(s'')|_{V_{act}} = \sigma(\hat{s}'')|_{V_{act}}$. Hence, the label reached in t_M after performing c from s'', is also given by $\mathit{Lab}(l(\hat{s}'))$, where $\hat{s}'$ 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, $\hat{s}' \neq \epsilon$, the assertion is not violated and hence, $\sigma(\hat{s}'') = \sigma(\hat{s}')$. In particular, $\mathtt{pc}_E \in \widehat{V_M}$ implies $\sigma(\hat{s}'')(\mathtt{pc}_E) = \sigma(\hat{s}')(\mathtt{pc}_E)$ as before, and hence again we have $l_E(s') = l_E(s'') = \sigma(\hat{s}'')(\mathtt{pc}_E) = \sigma(\hat{s}')(\mathtt{pc}_E)$.

   It is now sufficient to show that $s' = s''$ (a computation of length zero). By Definition 4.2.1, $l_M(s'') = \mathit{Lab}(l(\hat{s}''))$, which equals to $\mathit{Lab}(l(\hat{s}'))$ 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 $\sigma(\hat{s}'') = \sigma(\hat{s}')$ and the assumption $s'' \in \mathit{Extend}(\hat{s}'')$. □

Lemma 4.2.7 (P_M representation II). *Let ρ 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 ρ (but s and s' may be cut-point states). Let $\hat{s}$ be a state of P_M such that $s \in \mathit{Extend}(\hat{s})$, and $\mathit{cmd}(\hat{s})$ 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:*

- $s' \in \mathit{Extend}(\hat{s}')$

- *If s' is a cut-point state then $cmd(\hat{s}')$ is an* `env_move` *call.*
- *If s' is not a cut-point state then $cmd(\hat{s}')$ is an original command of t_M (and by the first requirement $cmd(\hat{s}') = cmd(s')$).*

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 $s \in Extend(\hat{s})$. We can construct $\hat{\rho}$ by adding each command from ρ to $\hat{\rho}$. Since $V_M \subseteq \widehat{V_M}$, and $s \in Extend(\hat{s})$, 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 ρ. Therefore, after performing a command from ρ in P_M, the next command to be executed is also an original command of t_M, and cannot be an `env_move` call or a newly added assertion. Thus, we can continue with the construction of $\hat{\rho}$ for every command in ρ.
- The last state $\hat{s}'$ may be ϵ. If s' is a cut-point state, then the last step of $\hat{\rho}$ should lead to an `env_move` call. However, some newly added assertions, not appearing in t_M, may appear before that `env_move` call in P_M. If s' contradicts one of these assertions, the computation in P_M will move to ϵ (before reaching the `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 `env_move` call, we can simply add all these assertions to $\hat{\rho}$, making $cmd(\hat{s}')$ an `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 `env_move` call (nor the new assertions added before such calls), making $cmd(\hat{s}')$ an original command of t_M.

□

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 `env_move` in every cut point label of t_M. In addition, the algorithm constructs the initial `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 $\neg cs0 \vee \neg cs1$ 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 P_0, based on the code of $t0$: we add `env_move`s 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. [2]

The initial `env_move` only havocs 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 P_M (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 P_M, 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 P_M. 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

[2] In 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 ;
bool cs1 = false , cs0 = false ;
int turn ;

void t0 () {
  while (true) {
    claim0 = true ;
    turn = 1;
    while (claim1 && turn != 0) {
    }
    cs0 = true ;
    // CRITICAL_SECTION
    assert(!cs1);
    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 $t1$.

```
bool claim0 = false , claim1 = false ;
bool cs1 = false , cs0 = false ;
int turn ;

void P0 () {
  env_move();
  while (true) {
    claim0 = true ;
    env_move();
    turn = 1;
    env_move();
    while (claim1 && turn != 0) {
      env_move();
    }
    env_move();
    cs0 = true ;
    assert(!cs1);
    env_move();
    cs0 = false ;
    claim0 = false ;
    env_move();
  }
}
```

Figure 4.3: The sequential program P_0 after adding environment move calls.

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

Definition 4.4.1. Let ψ, ψ' be formulas over $V \cup \{\mathtt{pc}_E\}$ and let l, l' be labels of t_M. We say that (l, ψ) is a *promise* of (l', ψ') 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, ψ) is a *promise* of $(l', \neg b)$ and $cmd(l')$ =`assert(b)`, then we say that (l, ψ) is a *promise of error*.

A promise (and a promise of error) (l, ψ) is defined "from the point of view" of t_M. That is, if ψ holds when t_M is at a specific label l, then (l', ψ') (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, ψ) is a *promise* of (l', ψ') and (l', ψ') is a promise of error, then (l, ψ) is also a promise of error. The proof follows.

Lemma 4.4.2 (Transitivity of promises of error). *Let (l, ψ) be a promise of (l', ψ'), and let (l', ψ') be a promise of error. Than (l, ψ) is also a promise of error.*

Proof. Since (l', ψ') is a promise of error, there exists a label $l'' \in L$ s.t. $cmd(l'') = assert(b)$ and (l', ψ') is a promise of $(l'', \neg b)$. It is sufficient to show that (l, ψ) 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 ρ 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 ρ' 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, ψ) is indeed a promise of $(l'', \neg b)$. $\square$

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 `env_move` along the path is proved to be infeasible. The counterexample is eliminated by refining the `env_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 `env_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 `env_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 `env_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 $\hat{l}_{n+1} = \hat{l}_\epsilon$ and that $cmd(\hat{l}_n) =$`assert(`b`)`, for some condition b. Let $\hat{l}_k$, for some $0 \leq k \leq (n-1)$, be the label of the last `env_move` call in $\hat{\pi}$. For convenience, we assume that labels within `env_move` are omitted from $\hat{\pi}$. That is, $\hat{l}_{k+1}$ is the label of the next command in P_M after the `env_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 $\hat{\pi}_{end} = \hat{l}_{k+1}, \ldots, \hat{l}_n$ to obtain a formula $\beta = wp(\hat{\pi}_{end}, \neg b)$ (Line 10 of Algorithm 4.1). Recall that β 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 $\hat{\pi}_{end}$, that reaches a state $\hat{s}'$ s.t $\sigma(\hat{s}') \models \neg b$. Also, note that there are no `env_move` calls in $\hat{\pi}_{end}$.

(2) Computing Condition Before the Environment Step: We compute (forward) a postcondition $\alpha = post(\hat{\pi}_{start}, \hat{\phi}_{init})$ starting from $\hat{\phi}_{init}$ for the path $\hat{\pi}_{start} = \hat{l}_0, \ldots, \hat{l}_k$. Note that α is not necessarily a strongest postcondition. Therefore, in order to ensure progress, we need to make sure that if $\hat{\pi}_{start}$ ends with a suffix of asserts, `assert(`c_1`)`, ..., `assert(`c_m`)`, they are all taken into account in our postcondition computation (e.g., by conjoining each c_i with

$post(\hat{\pi}_{start}, \hat{\phi}_{init})$). 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 α has the property that for every computation $\hat{\rho}$ from a state $\hat{s}$ of P_M to state $\hat{s}'$ whose path is $\hat{\pi}_{start}$: if $\sigma(\hat{s}) \models \hat{\phi}_{init}$ then $\sigma(\hat{s}') \models \alpha$. 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`)`, ..., `assert(`c_m`)`. If $\sigma(\hat{s}') \not\models c_i$ 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 α as an arbitrary postcondition. For progress, the minimal necessary requirement is the one mentioned above.

(3) Environment Query: We compute $\psi = Reach_{(t_E)}(\alpha, \beta)$ (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., $\phi_{init} \triangleq$ [`claim0 = 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 || (claim1 && turn != 0))` at Line 9 (i.e., $b = [\neg$`cs1` $\vee$ (`claim1` $\wedge$ `turn` $\neq 0)]$).

To check whether the last `env_move` call at Line 7 represents a real computation of $t1$, we compute the weakest precondition of the condition $\neg b \triangleq$ `cs1` $\wedge$ ($\neg$`claim1` $\vee$ `turn` $= 0$), taken from the violated assertion at Line 9, w.r.t. the path $\hat{\pi}_{end} = 8, 9$. The result is $\beta = wp(\hat{\pi}_{end}, \neg b) =$ (`cs1` $\wedge \neg$`claim1`). The computation of $\alpha = post(\hat{\pi}_{start}, \hat{\phi}_{init})$ for the path $\hat{\pi}_{start} = 2, 3, 4, 5, 6$ yields $\alpha = [\neg$`cs0` $\wedge$ `claim0` $\wedge$ ($\neg$`cs1` $\vee$ `claim1`)]. Note the conjunction with ($\neg$`cs1` $\vee$ `claim1`), which is the condition inside the last assert of $\hat{\pi}_{start}$. We then generate an environment query $Reach_{(t_E)}(\alpha, \beta)$.

(4) Refining the `env_move` Function: If ψ = *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 α', β' such that $\alpha \Rightarrow \alpha', \beta \Rightarrow \beta'$ and still $Reach_{(t_E)}(\alpha', \beta')$ = *FALSE* (Line 14). Finally, we refine `env_move` to eliminate the environment transition from α' to β' (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 (`if (`$\alpha'(W$`_old)) assume(`$\neg\beta'$`))`, where W_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 α' on the values of the variables before `env_move` is called). The commands ensure that if the condition α' is met when entering the `env_move` function, then condition β is satisfied when it exits. Since reachable computations from α' to β'

were proven by the environment query to be infeasible in t_E, we are ensured that `env_move` remains an over-approximation of the reachable computations of t_E.

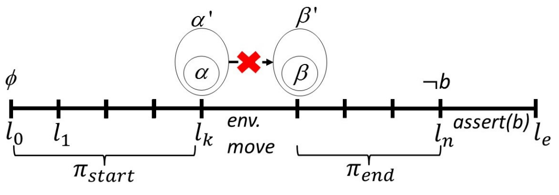
Example 4.5.2. Figure 4.6 presents the `env_move` function before and after the refinement step resulting from the reachability query described in Example 4.5.1, with $\alpha = [\neg \texttt{cs0} \wedge \texttt{claim0} \wedge (\neg \texttt{cs1} \vee \texttt{claim1})]$ and $\beta = (\texttt{cs1} \wedge \neg \texttt{claim1})$. The refinement step adds the two highlighted lines to the initial `env_move` function. The call to $Reach_{(t_E)}(\alpha, \beta)$ in this example results in $\psi = \mathit{FALSE}$. After generalization is applied (see (4) in Section 4.5, and Section 4.6), we obtain two formulas $\alpha' = TRUE, \beta' = \beta$ which indeed satisfy $Reach_{(t_E)}(\alpha', \beta') = \mathit{FALSE}$, $\alpha \Rightarrow \alpha'$ and $\beta \Rightarrow \beta'$. Since $Reach_{(t_E)}(\alpha', \beta') = \mathit{FALSE}$, there are no reachable computations of t_E reaching a state satisfying $\beta' = cs1 \wedge \neg claim1$. We therefore augment `env_move` with a new constraint (`if` $(\alpha'[W/W_0])$ `assume`$(\neg\beta')$), derived from this observation. Since $\alpha' = True$, this constraint is simplified to (`if (true) assume(!cs1 || claim1)`).

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

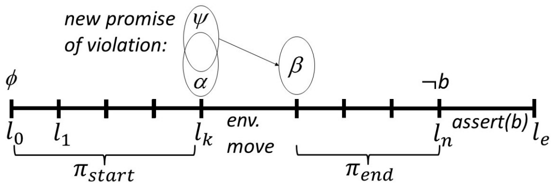
Remark. In both steps (4) and (5), the algorithm learns new information about the environment and is ensured to make progress. The formal proof appears in Section 5.2.

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 $\hat{\pi}_{start}$ would reach a state satisfying α. Then, the refined `env_move` forces the computation to reach $\neg\beta$, which means that the computation cannot reach and violate `assert`(b) using $\hat{\pi}_{end}$. 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 `assert`(b) is part of a sequence of newly added assertions before some `env_move` call, $\hat{\pi}'$ can be identical to $\hat{\pi}$, except for the last assertion (i.e., $\hat{\pi}'$ will not violate `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 `env_move` call). Nevertheless, the algorithm may also have to re-examine the path $\hat{\pi}$ (extended with the new



(a)



(b)

Figure 4.4: (a) If $Reach_{(t_E)}(\alpha, \beta) = \mathit{FALSE}$, we search fore more general α' and β' which restrict the environment transition; (b) If $Reach_{(t_E)}(\alpha, \beta) = \psi \neq \mathit{FALSE}$, then we know that ψ leads to β and that $\psi \wedge \alpha \neq \mathit{FALSE}$.

```
void P0() {
  assert((!cs1) || claim1);
  env_move();
  while (true) {
    claim0 = 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.

```
void env_move() {
  bool claim1_copy = claim1;
  int turn_copy = turn;
  bool cs1_copy = cs1;
  claim1 = havoc_bool();
  turn = havoc_int();
  cs1 = havoc_bool();
  if (true) {
    assume(!cs1 || claim1);
  }
}
```

Figure 4.6: The `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 α, β s.t. $Reach_{(t_E)}(\alpha, \beta) = \mathit{FALSE}$, we wish to generalize them into α', β' s.t. $Reach_{(t_E)}(\alpha', \beta') = False$ and $\alpha \Rightarrow \alpha'$ and $\beta \Rightarrow \beta'$. We start by representing α and β in negation normal form (NNF) ([40]). Note that, if a formula γ is in NNF, then for every subformula (except atomic negations) δ of γ and for every formula δ' s.t $\delta \Rightarrow \delta'$, it holds that $\gamma \Rightarrow \gamma[\delta/\delta']$ (monotonicity). Then generalization is performed by iteratively choosing subformulas δ of α (or β), replacing δ by a generalization $\delta' \not\equiv \delta$ s.t. $\delta \Rightarrow \delta'$, and computing $Reach_{(t_E)}(\alpha, \beta)$. For example, we can attempt to generalize $\delta = \delta_1 \wedge \delta_2$ to δ_1 or to δ_2. If the result of $Reach_{(t_E)}(\alpha, \beta)$ after generalization is still *FALSE*, the process continues. Otherwise, α (or β) 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 `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 `env_move`)**.** *During the run of Algorithm 4.1,* `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 `env_move` function. We start by proving that the initial `env_move` satisfies the lemma. We then consider the two possible modifications of P_M: refining `env_move`, and adding a new `assert(`$\neg\psi$`)` command, outside of `env_move`. Though the latter, by itself, cannot affect any computation within `env_move`, it may introduce new variables (from t_E) to P_M, if such a variable v, not already in $\widehat{V_M}$, 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 (`$\alpha'[W\_\text{copy}/W]$`) assume(`$\neg\beta'$`)` 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 $\widehat{V_M}$ 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, $\hat{\rho} = \hat{s}_0 \to \hat{s}_1 \to \hat{s}_2 \cdots \to \hat{s}_n$ starting from $\hat{s} = \hat{s}_0$ and ending at $\hat{s}' = \hat{s}_n$ such that $\hat{s}_i$ is obtained from $\hat{s}_{i-1}$ by performing v_i`=havoc()`, for every $1 \le i \le n$.

Since a havoc command can assign *any* non-deterministic value, we can choose a computation in which v_i`=havoc()` assigns $\sigma(\hat{s}_i)(v_i)$ with $\sigma(s')(v_i)$ (or with $l_E(s')$ if $v_i = \mathtt{pc}_E$). Since v_i`=havoc()` does not change variables other than v_i, we get $\sigma(\hat{s}_i)(v) = \sigma(\hat{s}_{i-1})(v)$ for every $v \neq v_i$. This yields a computation from $\hat{s}$ to $\hat{s}'$, which is clearly a complete single execution of `env_move`. It is left to show that $s' \in Extend(\hat{s}')$.

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

Finally, let $v \in \widehat{V_M}$. 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 $\hat{s}$ and $\hat{s}'$ is identical. Since $s \in Extend(\hat{s})$, we get the desired requirement about v for s' and $\hat{s}'$ 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 $\sigma(s')(v)$ (or to $l_E(s')$ if $v = \mathtt{pc}_E$) by the `v=havoc()` command, and it is unchanged by all other `havoc`s. Hence, $\sigma(\hat{s}')(v) = \sigma(s')(v)$ if $v \in V$ and $\sigma(\hat{s}')(v) = l_E(s')$ if $v = \mathtt{pc}_E$, which concludes the proof that $s' \in Extend(\hat{s}')$.

Induction Step: For the next two parts, let P_M denote the sequential program before refinement, and P^r_M the sequential program after refinement. Let `env_move`$_r$ denote the `env_move` function of P^r_M. Let $Extend_r$ denote the mapping $Extend$ for states of P^r_M.

We need to show that given a state $\hat{s}_r$ of P^r_M, s.t. $cmd(\hat{s}_r)$ is an `env_move`$_r$ call and $s \in Extend_r(\hat{s}_r)$, there exists a computation $\hat{\rho}_r = \hat{s}_r \to \cdots \to \hat{s}'_r$ in P^r_M, which is a complete single execution of `env_move`$_r$ s.t. $s' \in Extend_r(\hat{s}'_r)$. By the induction hypothesis, We know that for every state $\hat{s}$ of P_M, s.t. $cmd(\hat{s})$ is an `env_move` call and $s \in Extend(\hat{s})$, there exists a computation $\hat{\rho} = \hat{s} \to \cdots \to \hat{s}'$ in P_M, which is a complete single execution of `env_move` s.t. $s' \in Extend(\hat{s}')$.

Adding a New Variable to P_M Assume first that P^r_M was obtained from P_M after introducing a new variable v. Given a state $\hat{s}_r$ of P^r_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, $s \in Extend_r(\hat{s}_r)$, then $s \in Extend(\hat{s})$, 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 $cmd(\hat{s})$ is an `env_move` call. Thus, there exists a computation $\hat{\rho} = \hat{s}_0 \to \hat{s}_1 \to \cdots \to \hat{s}_n$ in P_M from $\hat{s} = \hat{s}_0$ to some $\hat{s}' = \hat{s}_n$, as guaranteed by the induction hypothesis above. There are two cases to consider:

- If v is not written by t_E, then `env_move`$_r$=`env_move`. Since ρ is a computation of t_E, it holds that $\sigma(s')(v) = \sigma(s)(v)$ (or $l_E(s') = l_E(s)$ if $v = \texttt{pc}_E$). Let $\hat{s}^r_0, \ldots, \hat{s}^r_n$ be a sequence of states of P^r_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 $\sigma(\hat{s}^r_i)(v) = \sigma(\hat{s}_r)(v)$, for $0 \le i \le n$. Note that in particular, it means that $\hat{s}^r_0 = \hat{s}_r$.

  Since all the commands in $\hat{\rho}$ do not use v (which does not appear at all in P_M), and since $\sigma(\hat{s}^r_{i-1})(v) = \sigma(\hat{s}^r_i)(v)$ for every $1 \le i \le n$, we can use the exact same commands as in $\hat{\rho}$ to form a computation $\hat{\rho}_r = \hat{s}^r_0 \to \cdots \to \hat{s}^r_n$. Since `env_move`$_r$=`env_move`, this is also a complete single execution of `env_move`$_r$ in P^r_M. It is left to show that $s' \in Extend_r(\hat{s}^r_n)$.

  Since $s' \in Extend(\hat{s}_n)$, and $\hat{s}'$ only differs from $\hat{s}^r_n$ by the appearance of v to $\sigma(\hat{s}^r_n)$, 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 $s \in Extend(\hat{s}_r)$, we get that $\sigma(\hat{s}^r_n)(v) = \sigma(\hat{s}^r)(v) = \sigma(s)(v) = \sigma(s')(v)$ (or $= l_E(s) = l_E(s')$, if $v = \texttt{pc}_E$), as required.

- If v is written by t_E, then a new v =`havoc()` command is added at the beginning of `env_move`$_r$. Let $\hat{s}^r_0, \ldots, \hat{s}^r_n$ be a sequence of states of P^r_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 $\sigma(\hat{s}^r_i)(v) = \sigma(s')(v)$ (or to $l_E(s')$, if $v = \texttt{pc}_E$), for $0 \le i \le n$. We also change $l(\hat{s}^r_0)$ to point to the second command in `env_move`$_r$, immediately after the new v =`havoc()`. Note that $\sigma(\hat{s}_r)$ and $\sigma(\hat{s}^r_0)$ agree on all variables, except for, possibly, v. Further, since $cmd(\hat{s}_r)$ is an `env_move`$_r$ call and $l(\hat{s}^r_0)$ is immediately after the first command in `env_move`$_r$, there exists a transition in P^r_M which moves from $\hat{s}_r$ to $\hat{s}^r_0$, by using the v =`havoc()` command at the beginning of `env_move`$_r$, which sets the value of v to $\sigma(\hat{s}^r_0)(v) = \sigma(s')(v)$ (or to $l_E(s')$ if $v = \texttt{pc}_E$). Let $\hat{\rho}_r$ be a computation in P_M, defined as follows: the computation starts by moving from $\hat{s}_r$ to $\hat{s}^r_0$, as described above. The computation continues from $\hat{s}^r_0$, using the transitions $\hat{s}^r_0 \to \cdots \to \hat{s}^r_n$, 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 `env_move`, and `env_move` only differs from `env_move`$_r$ by the addition of the initial v `=havoc()` command (which was used to reach from $\hat{s}_r$ to $\hat{s}_0^r$) at the beginning of `env_move`$_r$, then $\hat{\rho}_r$ is a complete single execution of `env_move`$_r$ from $\hat{s}_r$ to $\hat{s}_n^r$.

Again, it is left to show that $s' \in \mathit{Extend}_r(\hat{s}_n^r)$. 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 $v = \mathtt{pc}_E$), by construction.

Refining `env_move` `env_move` is only refined after an environment query $\mathit{Reach}_{(t_E)}(\alpha', \beta')$ returned *FALSE*, as described in the beginning of this section. The refinement step adds the commands `if` $(\alpha'[W\_\mathtt{copy}/W])$ `assume`$(\neg\beta')$ at the end of `env_move`$_r$. Given a state $\hat{s}_r$ of P_M^r as described above, let $\hat{s}$ be the state of P_M identical to $\hat{s}_r$. Clearly, since, $s \in \mathit{Extend}_r(\hat{s}_r)$, then $s \in \mathit{Extend}(\hat{s})$. Since $l(\hat{s})$ is also copied from $\hat{s}_r$, then $\mathit{cmd}(\hat{s})$ is an `env_move` call. Thus, there exists a computation $\hat{\rho} = \hat{s}_0 \to \hat{s}_1 \to \cdots \to \hat{s}_n$ 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` $(\alpha'[W\_\mathtt{copy}/W])$ 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 `env_move` and `env_move`$_r$ share the same prefix until $l(\hat{s}_n^r)$, the computation $\hat{\rho}_r' = \hat{s}_0^r \to \cdots \to \hat{s}_n^r$ is a valid computation in P_M^r, from $\hat{s}_r$, at the beginning of `env_move`$_r$ to $l(\hat{s}_n^r)$.

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

The condition $\alpha'[W\_\mathrm{copy}/W]$ uses auxiliary variables that are local to `env_move`$_r$ and reflects the values of $\hat{s}_r$'s variables, when entering `env_move`$_r$. 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 `env_move`$_r$.

If $\sigma(\hat{s}_r) \vDash \alpha'$, then $\sigma(\hat{s}_n^r) \vDash \neg\beta'$, 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 $\sigma(\hat{s}_r) \nvDash \alpha'$, 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 `env_move`$_r$ without changing $\sigma(\hat{s}_n^r)$. Hence, we achieve a computation from $\hat{s}_r$ to some new state $\hat{s}_r'$ at the end of `env_move`$_r$.

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

When ρ 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 $s \in Extend(\hat{s})$ and $cmd(\hat{s})$ is an* `env_move` *call, there exists a computation consisting of a complete single execution of* `env_move` *to a state $\hat{s}'$ such that $s \in Extend(\hat{s}')$.*

Lemma 5.1.3 (Assertions are promises of error)**.** *Let $\hat{l}_a$ be a label of P_M with $cmd(\hat{l}_a) =$* `assert(b)` *for some condition b. Then $(Lab(\hat{l}_a), \neg b)$ 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 `assert(b)` is an original assertion of t_M in P_M. According to Definition 4.1.3, $Lab(\hat{l}_a) = l_a$, where l_a is the label of t_M, for which $cmd(l_a)$ is the original `assert(b)`. Let s be some state of P such that $l_M(s) = l_a$ and $s \vDash \neg b$. Let ρ 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 the $cmd(l_a)$=`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 $Reach_{(t_E)}(\alpha, \beta)$ returned $\psi \not\equiv$ *FALSE*. The added assertion is of the form `assert(`$\neg\psi$`)` and it is added at a new label $\hat{l}_\psi$, right before the `env_move` call for which the environment query was applied. Let $l_\psi = Lab(\hat{l}_\psi)$ We would like to show that (l_ψ, ψ) is a promise of error.

Let s be a state of P such that $l_M(s) = l_\psi$ and $s \vDash \psi$. By Definition 3.2.1, there exists a computation ρ of t_E in P from s to some state s' such that $s' \vDash \beta$. Since this computation is a computation of t_E, $\texttt{pc}_M$ is left unchanged at every stage in ρ, hence $l_M(s') = l_M(s) = l_\psi$. Therefore, (l_ψ, ψ) is a promise of (l_ψ, β).

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

We now wish to show that (l_ψ, β) is a promise of $(Lab(\hat{l}_n), \neg b)$. Let $\hat{l}_k$ be the label in P_M, of the last `env_move` call in the found path, and $\hat{l}_{k+1}$ the label of the next command in P_M after the `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 `env_move` call, by Definition 4.1.3 $l_\psi = Lab(\hat{l}_\psi) = Lab(\hat{l}_k)$. Using the same argument for the `env_move` call at $\hat{l}_k$, we get that $Lab(\hat{l}_k) = Lab(\hat{l}_{k+1})$, and hence $l_\psi = Lab(\hat{l}_{k+1})$.

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

- $l(\hat{s}_\beta) = \hat{l}_{k+1}$
- $\forall v \in (V \cap \widehat{V_M})$: $\sigma(\hat{s}_\beta)(v) = \sigma(s_\beta)(v)$
- If $\texttt{pc}_E \in \widehat{V_M}$ then $\sigma(\hat{s}_\beta)(\texttt{pc}_E) = l_E(s_\beta)$

Clearly $s_\beta \in Extend(\hat{s}_\beta)$, and by Observation 4.2.3, $\hat{s}_\beta \vDash \beta$. 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. $\sigma(\hat{s}) \vDash \beta$, and specifically for $\hat{s}_\beta$, there exists a computation in P_M that passes through $\hat{\pi}_{end}$ (and therefore ends at $\widehat{l_n}$) and reaches a state $\hat{s}_{\neg b}$ such that $\sigma(\hat{s}_{\neg b}) \vDash \neg b$ (and $l(\hat{s}_{\neg b}) = \hat{l}_n$).

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

We conclude that indeed (l_ψ, β) is a promise of $(Lab(\hat{l}_n), \neg b)$. Since $(Lab(\hat{l}_n), \neg b)$ is a promise of error, by Lemma 4.4.2, (l_ψ, β) is also a promise of error. Since (l_ψ, ψ) is a promise of (l_ψ, β), we can use Lemma 4.4.2 again, to conclude that (l_ψ, ψ) is a promise of error. □

Essentially, lemma 5.1.3 describes the manner is 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 `env_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 ρ be a computation in P from a state s to a state s'. The *partition* of ρ 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 ρ.
- 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 ρ, then the *partitioning states* of ρ, $s_0, s_1, \ldots, s_k$ are the boundary states of the partition. That is, s_0 is the first state of ρ and for $1 \leq i \leq k$, s_i is the last state of r_i.

Figure 5.1 illustrates the definition above.

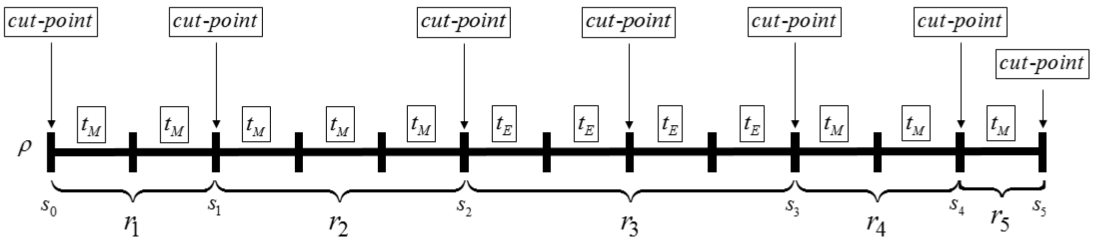


Figure 5.1: The partition and partitioning states of a computation ρ. Note that r_3 is allowed to have inner cut-point states, as it is a computation of t_E.

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 $\rho_e = s \to \cdots \to s' \to \epsilon$ be an initial violating computation in P, and let ρ be the prefix of ρ_e from s to s' (i.e., $cmd(s')$ =*`assert (b)` *and $\sigma(s') \not\models b$). Let $r_1, \ldots, r_k$ be the partition of ρ, and $s_0, s_1, \ldots, s_k$ the partitioning states of ρ. 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 $0 \leq i \leq j$: $s_i \in Extend(\hat{s}_i)$*
5. *If $i < j$, and r_{i+1} is a segment of t_M, then $cmd(\hat{s}_i)$ is an original command of t_M*
6. *If $i < j$, and r_{i+1} is a segment of t_E, then $cmd(\hat{s}_i)$ is an* `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 ρ 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 `env_move` function (Lemma 5.1.1). It might be the case that ρ contains two adjacent segments of t_M, in which case their representative computations in P_M will have an `env_move` call between them. We will use Corollary 5.1.2 to "skip" the `env_move` while preserving the valuation of the connecting state.

However, the representative computation of ρ 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 ρ. Our goal is to show that when ρ_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:

- $l(\hat{s}) = \hat{l}^{init}$ (initial label of P_M)
- For every $v \in V \cap \widehat{V_M}$: $\sigma(\hat{s})(v) = \sigma(s)(v)$
- If $\mathtt{pc}_E \in \widehat{V_M}$, then $\sigma(\hat{s})(\mathtt{pc}_E) = l_E(s)$

Let $\phi_{init} = \bigwedge_{v \in U} [v = \sigma_{init}(v)]$ be the initialization formula of P, for some $U \subseteq V$. Since s is initial in P, it holds that $\sigma(s)|_U = \sigma_{init}$, and $l_E(s) = l_E^{init}$. By the definition of $\hat{s}$, $\sigma(\hat{s})|_{V \cap \widehat{V_M}} = \sigma(s)|_{V \cap \widehat{V_M}}$. Since $U \subseteq V$, then clearly $\sigma(\hat{s})|_{U \cap \widehat{V_M}} = \sigma(s)|_{U \cap \widehat{V_M}} = \sigma_{init}|_{U \cap \widehat{V_M}}$. Further, if $\mathtt{pc}_E \in \widehat{V_M}$, then $\sigma(\hat{s})(\mathtt{pc}_E) = l_E(s) = l_E^{init}$. Hence, $\hat{s} \vDash \hat{\phi}_{init}$, and is an initial state of P_M.

By Definition 4.1.3, $Lab(\hat{l}^{init})$ 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., $Lab(l(\hat{s})) = l_M^{init} = l_M(s)$. By Definition 4.2.1, $s \in Extend(\hat{s})$.

Since all initial labels are considered as cut-points, the code of P_M starts with an `env_move` call, before the first original command of t_M, $cmd(s)$, and possibly some new assertions (not originally from t_M) prior to that `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 $cmd(\hat{s}')$ is the `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 $Lab(l(\hat{s})) = Lab(l(\hat{s}'))$. Hence, since $s \in Extend(\hat{s})$, it also holds that $s \in Extend(\hat{s}')$. 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 `env_move` function, from $\hat{s}'$ to another state $\hat{s}''$, satisfying $s \in Extend(\hat{s}'')$. $cmd(\hat{s}'')$, the next command after the `env_move`, is the first command of t_M, i.e. $cmd(s)$. We consider three possible cases:

1. If $k = 0$, ρ is an empty computation, and since ρ_e is a violation, then $cmd(\hat{s}'') = cmd(s)$=`assert(b)` for some condition b s.t. $\sigma(s) \not\models b$. Since $s \in Extend(\hat{s}'')$, then $\sigma(\hat{s}'') \not\models b$ as well. Hence we can choose $j = 0$, and $\hat{s}_0 = s''$. s'' is reachable from $\hat{s}$ (through $(\hat{s}')$ and the `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 $\hat{s}_0 = \hat{s}'$, as $\hat{s}'$ is reachable from $\hat{s}$, $s \in Extend(\hat{s}')$ and $cmd(\hat{s}')$ is an `env_move` call (Item 6).

3. If $k > 0$ and r_1 is a segment of t_M, we can choose $\hat{s}_0 = \hat{s}''$, as $\hat{s}''$ is reachable from $\hat{s}$, $s \in Extend(\hat{s}'')$ and $cmd(\hat{s}'')$ 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, $cmd(\hat{s}_{i-1})$ 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 stare $\hat{s}'$, that is either ϵ, or satisfies that $s_i \in Extend(\hat{s}')$ and $cmd(\hat{s}')$ is an `env_move` call. If $\hat{s}' = \epsilon$, we can now choose $j = (i-1)$ and complete the construction. Otherwise, if r_{i+1} is a segment of t_E, we choose $\hat{s}_i = \hat{s}'$, 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 $\hat{s}'$, consisting of a single complete execution of `env_move`, to a state $\hat{s}''$, also satisfying $s_i \in Extend(\hat{s}'')$. Since $\hat{\rho}$ passes through the entire `env_move` function, $cmd(\hat{s}'')$ is the next command after the `env_move`, i.e., an original command of t_M (see Observation 4.1.1). We then choose $\hat{s}_i = \hat{s}''$, 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, $cmd(\hat{s}_{i-1})$ is an `env_move` call. Then by Lemma 5.1.1, there exists a computation in P_M, consisting of a complete single execution of `env_move` from $\hat{s}_{i-1}$ to some state $\hat{s}'$ satisfying $s_i \in Extend(\hat{s}')$. Therefore, $cmd(\hat{s}')$ is the next command after `env_move`, i.e., an original command of t_M, and we can choose $\hat{s}_i = \hat{s}'$ and complete the construction of $\hat{s}_i$.

Last Step: i = k We now show that if $\hat{s}_k$ is constructed, ϵ 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, $cmd(\hat{s}_{k-1})$ 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 ϵ or to a state $\hat{s}'$ such that $s_k \in Extend(\hat{s}')$ and $cmd(\hat{s}')$ is either an `env_move` call or an original command of t_M. If $\hat{\rho}$ reaches ϵ, we choose $j = (k-1)$ and complete the construction. Otherwise, if $\mathit{cmd}(\hat{s}')$ is an `env_move` call, again by Corollary 5.1.2, there exists a computation from $\hat{s}'$ to some state $\hat{s}''$ also satisfying $s_k \in \mathit{Extend}(\hat{s}'')$ s.t. $\mathit{cmd}(\hat{s}'')$ is an original command of t_M, appearing immediately after the `env_move`. Since $s_k \in \mathit{Extend}(\hat{s}'')$, $\mathit{Lab}(l(\hat{s}'')) = l_M(s_k)$, and therefore $\mathit{cmd}(\hat{s}'') = \mathit{cmd}(s_k) =$`assert(`$b$`)`. Since $\sigma(s_k) \not\models b$, then $\sigma(\hat{s}'') \not\models b$ (see Observation 4.2.3), thus we can choose $\hat{s}_k = \hat{s}''$, as the next step from $\hat{s}''$ will reach ϵ. If $\mathit{cmd}(\hat{s}')$ is an original command of t_M, we can repeat the same arguments for $\hat{s}'$ instead of $\hat{s}''$. I.e., $\mathit{Lab}(l(\hat{s}')) = l_M(s_k)$ and $\mathit{cmd}(\hat{s}') = \mathit{cmd}(s_k) =$`assert(`$b$`)`. Thus, choosing $\hat{s}_k = \hat{s}''$ completes the construction.

Finally, if r_k is a segment of t_E, by Lemma 5.1.1, there exists a computation in P_M from $\hat{s}_{k-1}$ to some state $\hat{s}'$ satisfying $s_k \in \mathit{Extend}(\hat{s}')$ and whose command is the next command after the `env_move` function, i.e., an original command of t_M. As before, we learn that $\mathit{cmd}(\hat{s}') = \mathit{cmd}(s_k) =$`assert(`$b$`)`, and can choose $\hat{s}_k = \hat{s}'$.

□

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 ϵ 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 $\hat{\rho} = \hat{s} \to \cdots \to \hat{s}' \to \epsilon$ that does not use any `env_move`s. Since the last step of the computation leads to ϵ, $\mathit{cmd}(\hat{s}') =$`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. $s \in \mathit{Extend}(\hat{s})$. $\hat{s}' \neq \epsilon$, hence by Lemma 4.2.6, there exists a computation ρ in P from s to some state $s' \in \mathit{Extend}(\hat{s}')$. s is initial in P, and therefore, ρ is an initial computation in P.

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

□

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 P_M, 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 P_M 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 P_M. This set will strictly increase with every addition of a new assertion to P_M. The MOV set consists of pairs of states, that over-approximate the start and end state of all reachable computations of t_E. 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 $\hat{s}$ of P_M, satisfying (i) $s \in Extend(\hat{s})$ (ii) $cmd(\hat{s})$ is a newly added assertion `assert(b)` in P_M, and (iii) $s \models \neg b$
- MOV is the set of all pairs (s, s'), where s, s' are states of P such that there exist states $\hat{s}$, $\hat{s}'$ of P_M such that $s \in Extend(\hat{s})$, $s' \in Extend(\hat{s}')$, and there exists a complete single execution of `env_move` from $\hat{s}$ to $\hat{s}'$ (see Definition 4.2.5).

Lemma 5.2.2 (Progress when $Reach_{(t_E)}(\alpha, \beta)$ = *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 $Reach_{(t_E)}(\alpha, \beta)$ (in line 12) returned FALSE, and let ERR_r and MOV_r be the corresponding sets at the end of the iteration. Then $ERR_r = ERR$ and $MOV_r \subsetneq MOV$*

Proof. If $Reach_{(t_E)}(\alpha, \beta)$ returns *FALSE*, then Algorithm 4.1 refines `env_move`, and concludes the iteration. Since this does not change the code of P_M outside the `env_move` function (and specifically does not add, remove or change any exiting assertion), it is clear that $ERR_r = ERR$[1].

As for MOV, the refinement step augments `env_move` with `if` $(\alpha'[W\_copy/W])$ `assume`$(\neg\beta')$ commands at the end of the function. Let P_M^r be the program obtained from P_M following this refinement step, and let `env_move`$_r$ be the `env_move` function of P_M^r.

We first show that $MOV_r \subseteq MOV$. Let $(s, s') \in MOV_r$ 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 `env_move`$_r$ (who only has a single exit point, at its end), hence it must have a prefix $\hat{\rho}'_r$ consisting of the entire `env_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 `env_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, $s' \in Extend(\hat{s}')$, and we know that $s \in Extend(\hat{s})$ (because $\hat{s} = \hat{s}_r$), and that $\hat{\rho}$ is a complete single execution of `env_move` from $\hat{s}$ to $\hat{s}'$. By Definition 5.2.1, this proves $(s, s') \in MOV$.

Finally, we wish to show that $MOV_r \neq MOV$. The iteration can only reach line 12 (where $Reach_{(t_E)}(\alpha, \beta)$ is called), if an initial violating computation, $\hat{\rho}$, was found in P_M, containing at least one `env_move` call. Let $\hat{s}_\alpha$ be the state of that computation before the last `env_move` call, and let $\hat{s}_\beta$ be the state of the same computation immediately after the `env_move`. By the definition of a postcondition and a weakest precondition, and of α and β 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 `env_move`. According to Observation 4.2.2, there exist some $s \in Extend(s_\alpha)$ and $s' \in Extend(s_\beta)$, which means that $(s, s') \in MOV$. We would like to show that $(s, s') \notin MOV_r$

Let α', β' be the formulas used for refinement. The chosen formulas must satisfy $\alpha \Rightarrow \alpha'$, $\beta \Rightarrow \beta'$. Since $s \in Extend(\hat{s}_\alpha)$ 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 $(s, s') \in MOV_r$. That is, there exist some $\hat{s}_r$, $\hat{s}'_r$ of P_M^r, such that $s \in Extend(\hat{s}_r)$, $s' \in Extend(\hat{s}'_r)$ and a computation $\hat{\rho}_r$ from $\hat{s}_r$ to $\hat{s}'_r$, consisting of a complete single execution of `env_move`$_r$. In particular, $\hat{\rho}_r$ reaches (and passes) the newly added `if` $(\alpha'[W\_copy/W])$ `assume`$(\neg\beta')$ at the end of the `env_move` function. Since $s \in Extend(\hat{s}_r)$ and $s' \in Extend(\hat{s}'_r)$, 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` $(\alpha'[W\_copy/W])$ refers to the

[1]Note that the definition of ERR uses general (and not necessarily reachable) states of P_M. Though the set of reachable states in ERR may decrease due to the pruning of computations using `env_move`, the set of general states in ERR does not change

values of the variables of the first state that entered the `env_move`$_r$ function, i.e., $\hat{s}_r$. Since $\hat{s}_r \vDash \alpha'$, the computation enters the true branch of the if statement and reaches the `assume`$(\neg\beta')$ at some state $\hat{s}''_r$. Since the computation reaches the end of the function, it must hold that $\sigma(\hat{s}''_r) \vDash \neg\beta'$. The next (and final) state in $\hat{\rho}_r$ is $\hat{s}'_r$. However, passed `assume` commands do not change variables, hence $\sigma(\hat{s}''_r) = \sigma(\hat{s}'_r)$, which contradicts $\sigma(\hat{s}'_r) \vDash \beta'$.

□

When $Reach_{(t_E)}(\alpha, \beta)$ returns $\psi \neq$ *FALSE*, the new formula might contain new variables from V, which were not previously in $\widehat{V_M}$. These variables should then be added to $\widehat{V_M}$, and if they are written by t_E, they should also be havocked inside `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 $Reach_{(t_E)}(\alpha, \beta) \neq$ *FALSE*.

Lemma 5.2.3 (No regress when adding new variable)**.** *Let ERR and 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 $\widehat{V_M}$, creating a new program P^r_M with variables $\widehat{V_M}^r$. Let ERR_r and MOV_r be the corresponding sets in P^r_M. Then $ERR_r = ERR$ and $MOV_r = MOV$*

Proof. Adding v does not change any of the existing assertions, and does not add new assertions, hence it is clear that $ERR_r = ERR$. Let `env_move`$_r$ denote the `env_move` function of P^r_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 $\widehat{V_M}$, and maybe initializing v. In particular `env_move=env_move`$_r$. Hence, $MOV_r = MOV$ in this case as well.

If v is written by t_E, a new `havoc` statement is added for this variable inside `env_move`$_r$, and a new local auxiliary variable is added to `env_move`$_r$ to hold v's initial value.

We first show that $MOV \subseteq MOV_r$. Let $(s, s') \in MOV$, 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^r_M identical to $\hat{s}$ w.r.t all mutual variable ($\widehat{V_M}$) and with location $l(\hat{s}_r) = l(\hat{s})$, i.e., pointing to the same `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 $v = \texttt{pc}_E$). Clearly, since $s \in Extend(\hat{s})$, we also get $s \in Extend(\hat{s}_r)$.

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_COPY. The second step will be a v `=havoc()` command, setting the value of v to $\sigma(s')(v)$ (or $l_E(s')$ if $v = \texttt{pc}_E$). Let $\hat{s}''_r$ be the end state of $\hat{\rho}'_r$. As only v and v_COPY were changed by $\hat{\rho}'_r$, $\sigma(\hat{s}''_r)|_{\widehat{V_M}} = \sigma(\hat{s}_r)|_{\widehat{V_M}} = \sigma(\hat{s})$.

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

$\sigma(\hat{s}'_r)(v) = \sigma(\hat{s}''_r)(v) = \sigma(s')(v)$ (or $l_E(s')$ if $v = \mathtt{pc}_E$). Therefore, since $s' \in Extend(\hat{s}')$ then it also holds that $s' \in Extend(\hat{s}'_r)$. 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 env_move$_r$, which means that $(s, s') \in MOV_r$.

The proof of the converse, $MOV \supseteq MOV_r$ is similar, and is only briefly sketched. For this direction, we start with a computation $\hat{\rho}_r = \hat{s}_r \to \cdots \to \hat{s}'_r$ of env_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 env_move. We then 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 $\widehat{V_M}$, its value does not influence the definition of $Extend$ w.r.t. P_M, and hence we will have $s \in Extend(\hat{s})$ and $s' \in Extend(\hat{s}')$. □

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

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

Adding assert($\neg\psi$) does not change the env_move function, does not change any env_move call, and does not change original commands of t_M. If $(s, s') \in MOV$, then the corresponding states $(\hat{s}, \hat{s}')$ from Definition 5.2.1 are such that *cmd*$(\hat{s})$ is an env_move call, and *cmd*$(\hat{s}')$ is a command appearing immediately after an env_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, $MOV_r = MOV$ [2].

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

Finally, we need to show that $ERR_r \neq ERR$. Let P^r_M be the program obtained from P_M after adding the new assertion. The command assert($\neg\psi$) is added directly before an env_move call. Let assert(c_1), ..., assert(c_m) be the (possibly empty) sequence of assertions preceding this env_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, $\alpha \wedge \psi \neq$ *FALSE*.

Hence, there exists a valuation $\hat{\sigma}_r$ of $\widehat{V_M}$ such that $\hat{\sigma}_r \models (\psi \wedge \alpha)$, and since $\alpha \Rightarrow \gamma$, it holds that $\hat{\sigma}_r \models (\psi \wedge \gamma)$. Let $\hat{s}_r = (\hat{l}'_r, \hat{\sigma}_r)$, where $\hat{l}'_r$ is the new label associated with the newly added assert($\neg\psi$) in P^r_M. According to Observation 4.2.2, there exists some $s \in Extend(\hat{s}_r)$, and by Observation 4.2.3, $s \models \psi \wedge \gamma$. By Definition 5.2.1, we get that $s \in ERR_r$.

[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 $s \notin ERR$. Assume to the contrary that $s \in ERR$, and let $\hat{s}$ be a state of P_M such that $s \in Extend(\hat{s})$, $cmd(\hat{s})$ is some newly added `assert(`b`)` in P_M, and $s \vDash \neg b$.

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

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

□

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 $f(P_M) = (N - |ERR|) + |MOV|)$. 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 $Reach_{(t_E)}(\alpha, \beta)$ returns *FALSE* at some iteration, then by lemma 5.2.2, $|ERR|$ is left unchanged, while $|MOV|$ decreases, causing $f(P_M)$ to decrease. If $Reach_{(t_E)}(\alpha, \beta)$ returns $\psi \neq$ *FALSE*, then by lemma 5.2.4, $|MOV|$ is left unchanged, and $|ERR|$ 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 $Reach_{(t_E)}(\alpha, \beta)$ 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 `env_move` calls - in which case the algorithm terminates, proving the program unsafe. Therefore, termination is always guaranteed. □

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 $Reach_{(t_E)}(\alpha, \beta)$ is to check whether there exists a reachable computation ρ 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 α to β. Although, $\psi = \mathit{FALSE}$ guarantees that there are no such computations, there might be no such computation when $\psi \neq \mathit{FALSE}$ as well, if all the states satisfying $\psi \wedge \alpha$ are unreachable.

6.1 Sequential Program for Answering Environment Queries

If $\alpha \wedge \beta \not\equiv \mathit{FALSE}$, we simply return β, 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 ρ may not be initial. That is, ρ 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 ρ of t_E in P.

We observe that a reachable computation ρ is the continuation of a computation ρ' which is initial in P, but is not restricted to t_E alone. Before starting ρ, 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 ρ. Therefore, we add in P_E calls to a new function, `try_start`, which models the runs of t_M until the start of ρ, by assigning non-deterministic values to the variables written by t_M. The calls to `try_start` are added in all cut-points of t_E.

The `try_start` Function The `try_start` function is responsible for non-deterministically[1] setting the start point of ρ, 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` havocs all the variables in P_E, written by t_M. When `start` is set to $TRUE$, we add an `assume(`α`)` command after the havoc commands, as this is the state chosen to start the computation ρ. Whenever `start` is already $TRUE$, `try_start` exits without performing any `havoc` commands, ensuring that ρ indeed only uses transitions of t_E.

Recall that α and β may also refer to $\texttt{pc}_E$. To address this, we explicitly add $\texttt{pc}_E$ to $\widehat{V_E}$ as an auxiliary variable (which is different than the implicit `pc` variable of P_E). We need to make sure that whenever α or β are evaluated, the value of $\texttt{pc}_E$ 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 $\texttt{pc}_E$ variable.

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

Returning Result If a violating path is not found, we conclude that a computation as above does not exist and hence return $Reach_{(t_E)}(\alpha, \beta) = \textit{FALSE}$. If a violating path $\hat{\pi} = \hat{l}_0, \ldots, \hat{l}_{n+1}$ is found (with $\hat{l}_{n+1} = l_\epsilon$), 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 $\hat{\pi}_{end}$ be the sub-path of $\hat{\pi}$ from $\hat{l}_k$ to $\hat{l}_n$. We compute the weakest precondition of β w.r.t. the path $\hat{\pi}_{end}$. When $start = TRUE$, the `try_start` function does nothing except updating the explicit $\texttt{pc}_E$ variable of P_E. Hence, a computation passing through $\hat{\pi}_{end}$ is essentially a computation of t_E in P.

However, $\hat{\pi}_{end}$ 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 ψ over $V \cup \{\texttt{pc}_E\}$ 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 $\psi = (\texttt{pc}_E = l) \wedge wp(\hat{\pi}_{end}, \beta)[start/TRUE]$.

The computed ψ satisfies the desired requirement: For every state s of P with $l_E(s) = l$ s.t. $s \vDash wp(\hat{\pi}_{end}, \beta)$, there exists a computation ρ of t_E starting from s which follows a path corresponding to $\hat{\pi}_{end}$ in P and will reach a state s' satisfying β. 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 (!`$start$`)` 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 ρ 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 $Reach_{(t_E)}(\alpha, \beta)$ for $\alpha = \neg\texttt{cs1} \wedge \texttt{claim0} \wedge (\neg\texttt{claim1} \vee \texttt{turn} = 0)$ and $\beta = \texttt{cs1}$. The `try_start` function is called at every computed cut-point, followed by `assert(¬start ∨¬β)`, i.e. `assert(!start || !cs1)`. However, since `cs1` is only written by `t1`, 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 `assert(!start || !cs1)` immediately holds and can be dropped.

The parameters passed to `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 `start` is false, every call to `try_start` will havoc `claim0` and `turn`, as these are the variables written by `t0`. When the $TRUE$ branch of the `if(*)` is chosen, we set `start` to true and add assumptions for our precondition $\neg\texttt{cs1} \wedge \texttt{claim0} \wedge (\neg\texttt{claim1} \vee \texttt{turn} = 0)$.

If the activation `try_start` is in lines 2, 5, 16 or 18, `t1` will be stuck in the busy wait loop, since we assume `claim0`, whose value does not change, and `turn` is set to 0 before the loop in line 6. If the activation `try_start` is in lines 7 or 9, where `claim1` (who is only written by `t1`) is always true, the assumptions `assume(!claim1 || turn==0)` and `assume(claim0)` imply $\texttt{claim0} \wedge \texttt{turn} \neq 1$, hence there are no computations leaving the loop. If the activation `try_start` is in line 13, all the computations are disregarded, since `assume(!cs1)` conflicts with `cs1`'s actual value at this location, which is always true.

Finally, the only activation `try_start` that can lead to a violation is the one in line 11, which is followed by the command `cs1=true`. The path $\hat{\pi}_{end}$, starting after the activation `try_start` and reaching the violation, consists of the single command in line 12 (omitting the "non-activating" `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 `try_start`. Hence, we compute $wp(\hat{\pi}_{end}, \texttt{cs1})[start/TRUE] = TRUE$, and thus $Reach_{(t_E)}(\alpha, \beta)$ returns $\psi \triangleq (\texttt{pc}_E = L_{24})$.

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 ψ, 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.

```
bool claim0 = false, claim1 = false;
bool cs1 = false;
int turn;

bool start = false;

void try_start(int pc_aux) {
  pc_t1 = pc_aux;
  if (!start) {
    claim0 = havoc_bool();
    turn = havoc_int();
    if ( * ) {
      start = true;
      assume(!cs1);
      assume(claim0);
      assume(!claim1 || turn == 0);
    }
  }
}
```

```
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(L26);
    assert(!start || !cs1);
    cs1 = false;
    try_start(L27);
    claim1 = false;
    try_start(L28);
  }
}
```

Figure 6.1: The sequential program P_1 for computing $\mathit{Reach}_{(t_E)}(\alpha, \beta)$ for $\alpha = \neg\texttt{cs1} \wedge \texttt{claim0} \wedge (\neg\texttt{claim1} \vee \texttt{turn} = 0)$ and $\beta = \texttt{cs1}$. For the convenience of presentation, we split the `assume(`α`)` to three `assume` commands.

General Structure P_E is constructed for the purpose of answering an environment query $\mathit{Reach}_{(t_E)}(\alpha, \beta)$. 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 (Lab_E)**.** Let $\hat{l}$ be a label of P_E, not inside `try_start`, with $\mathit{cmd}(\hat{l}) = c$. We define $\mathit{Lab}_E(\hat{l})$ recursively as follows:

- If c is an original command of t_E, appearing at label l in t_E, then $\mathit{Lab}_E(\hat{l}) = l$.
- Otherwise, c is either an `assert` or a `try_start` call. We define $\mathit{Lab}_E(\hat{l}) = \mathit{Lab}_E(\hat{l}')$, 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 (Extend_E)**.** Let $\hat{s} = (\hat{l}, \hat{\sigma})$ be a state of P_E, s.t. $\hat{l}$ is not inside `try_start`. We define the set $\mathit{Extend}_E(\hat{s})$ to be the set of all states $s = (\bar{l}, \sigma)$ of P such that:

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

- For every $v \in V$: $v \in \widehat{V_E} \Rightarrow \sigma(s)(v) = \sigma(\hat{s})(v)$.

Observation 6.2.3.
1. Unlike Definition 4.2.1, the definition above does not make any requirement for the value $\hat{\sigma}(\texttt{pc}_E)$. 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 $\texttt{pc}_E$. The purpose of this variable is to allow the query parameters α and β to argue about the label of t_E. Hence, it is important that $\texttt{pc}_E$ will have an updated value at every place where we use one of the input formulas (α or β) in which it may appear. This is indeed the case, as α appears only inside the `try_start` function, whose first command is used to update $\texttt{pc}_E$, and β appears in assertions, which are placed immediately after `try_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, $Extend_E(\hat{s})$ is also never empty.

3. We would have also liked to claim, similar to Observation 4.2.3, that if $s \in Extend_E(\hat{s})$, then for every formula γ over $\widehat{V_E} \cap (V \cup \{\texttt{pc}_E\})$, it holds that $s \vDash \gamma \iff \sigma(\hat{s}) \vDash \gamma$. However, as mentioned above, the value of $\texttt{pc}_E$ is only updated (and corresponds to $l_E(s)$) at specific locations of P_E. Hence, if γ contains $\texttt{pc}_E$, this general claim may be incorrect. However, the claim is correct in all cases where $\texttt{pc}_E$ has an updated value. In particular, within `try_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 `start` variable is $TRUE$, 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 $\sigma(\hat{s})(start) = TRUE$. Then $\sigma(\hat{s}'')(start) = TRUE$ 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 $\sigma(\hat{s}'')(start) \neq TRUE$. $\hat{s}'' \neq \hat{s}$, the first state in $\hat{\rho}$, since $\sigma(\hat{s})(start) = TRUE$. Let $\hat{s}_p$ be the state preceding $\hat{s}''$ in $\hat{\rho}$. Since $\hat{s}''$ is the first state such that $\sigma(\hat{s}'')(start) \neq TRUE$, $cmd(\hat{s}_p)$ must change the value of $start$ (from $TRUE$ to $\sigma(\hat{s}'')(start) = FALSE$).

However, according to the construction described in Section 6.1, the only commands in P_E that change the value of $start$ (after initialization) are $start = TRUE$ inside the `try_start` function, and violated assertions (which transition to ϵ, in which the value of $start$ is undefined). None of the commands in $\hat{\rho}$ can be a violated assertion, as this would lead the computation to ϵ, which is a final state. This is impossible, since the last state of $\hat{\rho}$ satisfies $\hat{s}' \neq \epsilon$. Hence, $cmd(\hat{s}_p) =$ "$start = TRUE$", and since $\hat{s}''$ is obtained from $\hat{s}_p$ by performing $cmd(\hat{s}_p) =$ "$start = TRUE$", it must hold that $\sigma(\hat{s}'')(start) = TRUE$, 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 $\sigma(\hat{s})(start) = TRUE$. Assume that $l(\hat{s})$ and $l(\hat{s}')$ are not inside `try_start`, and that $\hat{\rho}$ has no original commands of t_E. Then for every state s of P: $s \in Extend_E(\hat{s}) \iff s \in Extend_E(\hat{s}')$.*

Proof. The only other commands in P_E which are not original commands of t_E are `try_start` calls and executions (i.e., commands within `try_start`), and assertions. Since, 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 $Lab_E(l(\hat{s})) = Lab_E(l(\hat{s}'))$. The definition of $Extend_E$ only depends on the mapping Lab_E, and the values of variables in $V_E \cap V$. Hence, it is sufficient to show that $\sigma(\hat{s})(v) = \sigma(\hat{s}')(v)$ 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 σ. Further, since $\sigma(\hat{s})(start) = TRUE$, by Lemma 6.2.4, $\sigma(\hat{s}'')(start) = TRUE$ for every $\hat{s}''$ in $\hat{\rho}$. When $start = TRUE$, the `try_start` function, only updates the value of $\mathtt{pc}_E$ (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, $\sigma(\hat{s})(v) = \sigma(\hat{s}')(v)$ 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 $\sigma(\hat{s})(start) = TRUE$ and $l(\hat{s})$, $l(\hat{s}')$ are not inside `try_start`. Assume also that $cmd(\hat{s})$ is an original command of t_E and the rest of the commands in $\hat{\rho}$ are not. Then for every state $s \in Extend_E(\hat{s})$ there exists a state $s' \in Extend_E(\hat{s}')$ such that $s' \in next(s,t_E)$.*

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

Additionally, $Vars(c) \subseteq V_E \subseteq \widehat{V_E}$, and since $s \in Extend_E(\hat{s})$, 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 `try_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 $c = cmd(s,t_E)$ to s (i.e., $s' \in next(s,t_E)$), which satisfies $s' \in Extend_E(\hat{s}'')$.

By Lemma 6.2.4, $\sigma(\hat{s}'')(start) = TRUE$. Since c is an original command of t_E, $l(\hat{s}'')$ cannot be inside `try_start` (but it can be a `try_start` call). Thus, we can use Lemma 6.2.5 (for the suffix of $\hat{\rho}$ starting from $\hat{s}''$) to obtain $s' \in Extend_E(\hat{s}')$, 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 `try_start` and $\sigma(\hat{s})(start) = TRUE$.*

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

Proof. Let $s \in Extend_E(\hat{s})$. 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 $\hat{s}''$ is the first state in $\hat{\rho}$ such that $cmd(\hat{s}'')$ is an original command of t_E, then $s \in Extend_E(\hat{s}'')$. 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 $cmd(\hat{s})$ 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 $cmd(\hat{s}'')$ is an original command of t_E and including $\hat{s}'$ as well (even if $cmd(\hat{s}')$ is not an original command of t_E). By our assumption, $\hat{s}_0 = \hat{s}$. Moreover, by Lemma 6.2.4, $\sigma(\hat{s}_i)(start) = TRUE$ for $0 \leq i \leq k$. Further, according to our state selection, for every $0 \leq i \leq (k-1)$, there exists a computation in P_E from $\hat{s}_i$ to $\hat{s}_{i+1}$, in which $cmd(\hat{s}_i)$ is the only original command of t_E. Additionally, $l(\hat{s}_i)$ is not inside `try_start` for $0 \leq i \leq k$ (for $\hat{s}_k = \hat{s}'$ by the lemma's condition, and for the rest, $cmd(\hat{s}_i)$ is an original command of t_E, which cannot be inside `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 $s_i \in Extend_E(\hat{s}_i)$ and $s_i \in next(s_{i-1}, t_E)$ for $1 \leq i \leq k$. Since $\hat{s}_k = \hat{s}'$, we can choose $s' = s_k$ to satisfy the lemma. □

The purpose of the lemmas above was to describe how a computation of P_E, in which the $start$ 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 `env_move` function. In P_E, the activation of the $start$ 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 ρ of t_E is indeed represented within P_E. For the correctness of our proof, we are only interested in computations from α to β. 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 `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 ρ (and hence making ρ reachable) is represented by commands of t_E in P_E and `try_start` to abstract t_M. The representation of ρ 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 ρ be a computation of t_E from s to s' that is reachable in P such that $s \vDash \alpha$, $s' \vDash \beta$ 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. ρ is reachable in P, then there exists a computation ρ_0 in P from an initial state, s_0 of P, to s. ρ_0 contains both commands of t_M and commands of t_E. To prove the lemma, we need to partition ρ_0 to segments (i.e., sub-computations of ρ_0), similar to Definition 5.1.4. However, in Definition 5.1.4, we split ρ_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` havocs 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 α. Since ρ_0 ends with s and $s \vDash \alpha$, the compatible state in P_E would also satisfy α, thus not pruning the computation.

To complete, we need to match each command in ρ 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 $\mathtt{pc}_E$). This process would end at some state $\hat{s}'$ such that $s' \in Extend_E(\hat{s}')$ and $cmd(\hat{s}')$ is `assert(!`$start$ `||` $\neg\beta$`)`. $start$ was set to $TRUE$ at the activation `try_start` and is unchanged for the rest of the computation. Since $s' \vDash \beta$, we will have $\sigma(\hat{s}') \vDash start \wedge \beta$. Hence, the assertion will be violated and the constructed computation is indeed an initial violating computation. □

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 $\hat{\rho} = \hat{s} \to \ldots \hat{s}' \to \epsilon$ 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 `assert(!`$start$` || `$\neg\beta$`)`. The variable $start$ has an initial value of *FALSE*. If $start$ is left unchanged during $\hat{\rho}$, then in particular $\sigma(\hat{s}')(start) =$ *FALSE*, which is a contradiction to $\hat{\rho}$ being a violating computation.

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

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

Lemma 6.2.10. *Let $\hat{\rho} = \hat{s} \to \ldots \hat{s}' \to \epsilon$ be a violating initial computation in P_E, and let $\hat{s}''$ be the first state in $\hat{\rho}$ after the activation `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 `try_start`.*

Then $\sigma(\hat{s}'') \vDash \alpha \wedge (\mathtt{pc}_E = l) \wedge wp(\hat{\pi}_{end}, \beta)[start/TRUE]$.

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

The assertions in P_E are of the form `assert(!`$start$` || `$\neg\beta$`)`. Since $\hat{\rho}$ is a violation, it follows that $\hat{s}' \vDash start \wedge \beta$ and in particular, $\hat{s}' \vDash \beta$. Since $\hat{\rho}_{end}$ is a computation from $\hat{s}''$ to $\hat{s}'$, by the definition of a weakest precondition, it follows that $\sigma(\hat{s}'') \vDash wp(\hat{\pi}_{end}, \beta)$. Since, $\hat{s}''$ appears after the activation `try_start`, then $\sigma(\hat{s}'')(start) = TRUE$. Therefore, it must also hold that $\sigma(\hat{s}'') \vDash wp(\pi_{end}, \beta)[start/TRUE]$.

Finally, the activation `try_start` sets the value of $\mathtt{pc}_E$ to l. This value is only set at the beginning of `try_start`, and is unchanged until the next `try_start` call. Hence, $\sigma(\hat{s}'')(\mathtt{pc}_E) = l$, which means that $\sigma(\hat{s}'') \vDash (\mathtt{pc}_E = l)$

□

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 $Reach_{(t_E)}(\alpha, \beta)$ and let ψ be the results of the query, obtained according to the description in Section 6.1. Then ψ satisfies the properties of Definition 3.2.1.*

Proof. First, if $\alpha \wedge \beta \not\equiv$ *FALSE*, then β is a valid answer for $Reach_{(t_E)}(\alpha, \beta)$, 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 $\psi =$ *FALSE*. When $\psi =$ *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 $s \vDash \alpha$ to $s' \vDash \beta$ 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 $\hat{\rho} = \hat{s} \to \cdots \to \hat{s}' \to \epsilon$ be a violating computation in P_E. Let $\hat{s}''$ be the first state after the activation $\texttt{try\_start}(l)$ (l being the parameter passed), and $\hat{\pi}_{end}$ the path used by $\hat{\rho}$ starting from $\hat{s}''$ until $\hat{s}'$. The returned formula is $\psi = (\texttt{pc}_E = l) \wedge wp(\hat{\pi}_{end}, \beta)[start/TRUE]$. According to Lemma 6.2.10, $\sigma(\hat{s}'') \vDash \alpha \wedge \psi$. In particular, this means that $\alpha \wedge \psi \not\equiv \mathit{FALSE}$. Hence, the first and third property of Definition 3.2.1 hold.

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

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

We first wish to show that $s_\psi \in Extend_E(\hat{s}_\psi)$. The properties for variables in $V \cap \widehat{V_E}$ hold immediately, by the definition of $\hat{s}_\psi$. It is left to show that $l_E(s_\psi) = Lab_E(l(\hat{s}_\psi))$. $Lab_E(l(\hat{s}''))$ 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 $\texttt{try\_start}$, called with parameter l, which is also the label associated with the next original command of t_E to be executed. Hence, $Lab_E(l(\hat{s}'')) = l$. Since $s_\psi \vDash \psi$, then in particular $s_\psi \vDash (\texttt{pc}_E = l)$, i.e., $l_E(s_\psi) = l$. Since $l(\hat{s}_\psi) = l(\hat{s}'')$, we get $l_E(s_\psi) = Lab_E(l(\hat{s}_\psi))$, and $s_\psi \in Extend_E(\hat{s}_\psi)$ as required.

Note also that $\hat{s}_\psi$ has an updated value of $\texttt{pc}_E$, i.e., $\sigma(\hat{s}_\psi)(\texttt{pc}_E) = l = Lab_E(l(\hat{s}_\psi))$. Hence, since $s_\psi \vDash \psi$ and $s_\psi \in Extend_E(\hat{s}_\psi)$, it follows that $\sigma(\hat{s}_\psi) \vDash \psi$ (see the last property of Observation 6.2.3). In particular, $\sigma(\hat{s}_\psi) \vDash wp(\hat{\pi}_{end}, \beta)[start/TRUE]$. Since $\sigma(\hat{s}_\psi)(start) = TRUE$, it also holds that $\sigma(\hat{s}_\psi) \vDash wp(\hat{\pi}_{end}, \beta)$. 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 $\hat{\pi}_{end}$ such that $\sigma(\hat{s}'_\psi) \vDash \beta$.

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

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` havocs 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 $Reach_{(t_E)}(\alpha, \beta)$ returned $\psi = \mathit{FALSE}$. Second, new assertions are added to P_M iff an environment query returned $\psi \neq \mathit{FALSE}$. 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, $\widehat{V_M}$ may now include $\texttt{pc}_i$ as an additional variable for every thread $t_i \in T$, instead of a single $\texttt{pc}_E$.

Definition 7.1.1 (Multithreaded Environment Query)**.** Let P be a concurrent programs with threads $t_1, \ldots, t_m$, and let $T \subsetneq \{t_1, \ldots, t_m\}$. An *environment query* $\mathit{Reach}_{(T)}(\alpha, \beta)$ receives conditions α and β over $V \cup \bigcup_{i:\, t_i \in T} \{\texttt{pc}_i\}$, and returns a formula ψ over $V \cup \bigcup_{i:\, t_i \in T} \{\texttt{pc}_i\}$ 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. $s \vDash \alpha$, (3) ends in a cut-point state s' s.t. $s' \vDash \beta$, then $\psi \wedge \alpha \not\equiv \mathit{FALSE}$.

2. For every state s s.t. $s \vDash \psi$, there exists a computation (not necessarily reachable) of T in P from s to some s' s.t. $s' \vDash \beta$.

3. $\psi \neq \mathit{FALSE} \Rightarrow \psi \wedge \alpha \not\equiv \mathit{FALSE}$.

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 (α, β, ψ) 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 $Extend(\hat{s})$ to be the set of all states $s = (\bar{l}, \sigma)$ of P such that:

- $l_M(s) = Lab(l(\hat{s}))$.
- For every thread $t_i \in T$: If $\texttt{pc}_i \in \widehat{V_M}$ then $l_i(s)$[1]$= \sigma(\hat{s})(\texttt{pc}_i)$.

[1] Where $l_i(s)$ denotes the value of $\texttt{pc}_i$ in s.

- For every $v \in V$: $v \in \widehat{V_M} \Rightarrow \sigma(s)(v) = \sigma(\hat{s})(v)$.

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

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

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

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

We note that Definition 7.1.4 is almost identical to Definition 4.4.1, with $V \cup \{\mathtt{pc}_E\}$ simply replaced by $V \cup \bigcup_{i:\ t_i \in T} \{\mathtt{pc}_i\}$. Definition 7.1.2 also highly resembles Definition 4.2.1, where instead of addressing a single $\mathtt{pc}_E$ variable in $\widehat{V_M}$, the definition addresses multiple $\mathtt{pc}_i$ 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 $\mathtt{pc}_E$ should be replaced by multiple $\mathtt{pc}_i$ 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, $\mathit{Reach}_{(T)}(\alpha_1, \beta_1)$ 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: $\mathit{Reach}_{(T')}(\alpha_2, \beta_2)$. 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 ρ, there exists some initial computation ρ_i in P, which is a prefix of a computation ρ' such that $\rho' = \rho_i \cdot \rho$. Each step in ρ_i can be performed by any of the threads in P, while each step in ρ 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 ρ_i and ρ.

Over-approximating ρ_i The purpose of the `try_start` function is to abstract ρ_i, and to non-deterministically decide when ρ_i ends, α_1 is satisfied, and ρ should start. To this end, as long as the `start` variable remains *FALSE*, the `try_start` function havocs all variables written by any thread of P other than t'_M[2]. Similar to Chapter 6, the `try_start` also non-deterministically sets `start` to $TRUE$ and assumes α_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 ρ When `start` is set to $TRUE$, we move from over-approximating ρ_i to over-approximating ρ. 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 havocs variables written by threads in T', similar to the description in Section 4.3. It is later refined by statements of the form `if` $(\alpha'_2[W\_\text{copy}/W])$ `assume`$(\neg\beta'_2)$, similar to (4) of Section 4.5, following inner environment queries $Reach_{(T')}(\alpha_2, \beta_2)$, 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 α_1 to β_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

[2]Variables that neither belong to any of the threads in T, nor appear in α_1 or β_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:     $P_M$' = add try_start calls in cut-point locations of $t'_M$
3:     $P_M$' = add env_move$'$ calls after try_start in $P_M$' and initialize env_move$'$
4:     $P_M$' = add assert($\neg start \vee \neg\beta_1$) after every env_move$'$ in $P_M$'
5:     while a violating path exists in $P_M$' do        // using sequential MC
6:         Let $\hat{\pi} = \hat{l}_0, \ldots, \hat{l}_{n+1}$ be a violating path (with $\hat{l}_{n+1} = \hat{l}_\epsilon$)
7:         Let assert($\neg start \vee b$) be the violated assertion at label $\hat{l}_n$
8:         Let try_start($l$) be the activation try_start at some label $\hat{l}_j$
9:         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:        if there are no env_move$'$ calls in $\hat{\pi}'$ then:
11:            return $\psi \triangleq (\mathtt{pc}_{M'} = l) \wedge wp(\hat{\pi}', \beta_1)[start/TRUE]$
12:        end if
13:        let $\hat{l}_k$ be the label of last env_move call in $\hat{\pi}$ (and in $\hat{\pi}'$)
14:        let $\hat{\pi}_{start} = \hat{l}_0, \ldots, \hat{l}_k$ and $\hat{\pi}_{end} = \hat{l}_{k+1}, \ldots, \hat{l}_n$
15:        $\beta_2 = wp(\hat{\pi}_{end}, \neg b)$              // see (1) in Section 4.5
16:        $\alpha_2 = post(\hat{\pi}_{start}, \hat{\phi}_{init})$          // see (2) in Section 4.5
17:        Let $\psi_2 = Reach_{(T')}(\alpha_2, \beta_2)$        // environment query for $T'$
18:        if $\psi_2$ is FALSE then
19:            Let $\alpha'_2$, $\beta'_2$ be as in (4) in Section 4.5.
20:            $P_M$' = RefineEnvMove($P'_M, \alpha'_2, \beta'_2$)
21:        else        // see (5) in Section 4.5
22:            Add new label $\hat{l}'$ in $P'_M$ right before $\hat{l}_k$ with $cmd(\hat{l}')$ =assert($\neg start \vee \neg\psi_2$)
23:        end if
24:    end while
25:    return $\psi \triangleq$ FALSE
26: end procedure
```

will be examined in the same manner as in Algorithm 4.1, by another environment query $Reach_{(T')}(\alpha_2, \beta_2)$ (Line 17). If we learn that T' cannot lead from α_2 to β_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 β_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 ψ as appears in Line 11. The returned ψ guarantees the reachability of the violated assertion through steps of t'_M alone. The violated assertion itself guarantees the reachability of β_1. The computed ψ must also intersect α_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 ψ intersecting α_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 β_1", i.e., conditions ensuring the reachability of β_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 $Reach_{(T)}(\alpha_1, \beta_1)$ be an environment query in P for an environment T, and let ψ be the results of the query, obtained by Algorithm 7.1. Then ψ 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 `env_move`$'$ mentioned in Algorithm 7.1 would be empty, and can be omitted. There is no need to examine inner environment queries from some α_2 to some β_2, as there are no environment computations. Note that without the `env_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 $T' = T \backslash \{t'_M\}$. $|T'| < |T|$, 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 `env_move`$'$. Next, we discuss both of these refinements.

`env_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 ρ 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 `env_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 `env_move`$'$ havocs all variables written by threads in T'. Refinements only occur after an inner environment query $Reach_{(T')}(\alpha'_2, \beta'_2)$ returned *FALSE*. By the induction hypothesis, this means that there are no reachable computations of T' in P from α'_2 to β'_2. We refine `env_move`$'$ with commands of the form `if` $(\alpha'_2[W\_\texttt{copy}/W])$ `assume`$(\neg\beta'_2)$, which only block `env_move`$'$ from reaching β'_2 when it was called with a state satisfying α'_2. Thus the refinement does not "lose" the representation of any reachable computation of T' in P.

Assertions in P'_M Represent Promises of β_1 . A *promises of* β_1 is a condition ψ' 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 β_1. The proof follows the same pattern as Lemma 5.1.3. At initialization, this holds since all assertions are of the form `assert`$(\neg start \vee \neg\beta_1)$. Other assertions are only added after inner environment queries returned $\psi_2 \triangleq Reach_{(T')}(\alpha_2, \beta_2) \neq$ *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, β_2 is computed as a weakest precondition and ensures the reachability of β_1 using steps of t'_M alone. Further, by

[3]denoting the `pc` variable of t'_M

the induction hypothesis, the result ψ_2 ensures the reachability of β_2 using computations of T'. Combining computations of t'_M and computations of T' yields computations of T. Hence, β_1 is reachable from ψ_2 by computations of T. Since the newly added assertion is of the form `assert(`$\neg start \vee \neg\psi_2$`)`, its violation indeed represents a promise of β_1. For other assertions, we can continue inductively and use the transitivity of the definition of "promises of β_1", similar to Lemma 5.1.3.

$\psi \neq$ *FALSE* $\Rightarrow \psi \wedge \alpha_1 \neq$ *FALSE* We now wish to show that whenever Algorithm 7.1 returns $\psi \neq$ *FALSE*, it means that $\psi \wedge \alpha_1 \neq$ *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 $\psi \neq$ *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 `assert(`$\neg start \vee b$`)` be the violated assertion (Line 7). The state immediately after the activation `try_start` must satisfy $\psi \wedge \alpha_1$. It satisfies ψ, because ψ 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 α_1 as α_1 was assumed at the end of the activation `try_start`, and otherwise $\hat{\rho}$ could not have continued. We conclude that $\psi \wedge \alpha_1 \neq$ *FALSE*.

$\psi =$ *FALSE* Implies no Computation from α_1 to β_1 We now wish to show that when Algorithm 7.1 returns $\psi =$ *FALSE*, it means that there is no reachable computation of T in P from a cut point stare s satisfying α_1, to a cut point state s' satisfying β_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 ρ. As discuss in the beginning of this section, since ρ is reachable, there exists some initial computation ρ_i in P, which is a prefix of a computation ρ' 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 $\psi =$ *FALSE*.

$\hat{\rho}'$ would also consist of two subcomputations $\hat{\rho}' = \hat{\rho}_i \cdot \hat{\rho}$, based on ρ_i and ρ. During $\hat{\rho}_i$, the `start` variable will be *FALSE*. Thus, the `env_move`$'$ function, whose body is contained within a `if (start)`$\{\ldots\}$ statement, will be effectively disabled. Therefore, we can use a construction similar to Lemma 6.2.8 until the activation `try_start`. $\hat{\rho}_i$ 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 ρ_i and ρ. Since $s \models \alpha_1$, the `assume(`α_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, ρ would be partitioned into segments. Recall that ρ is a computation of T. Thus, each segment will either corresponds to a computation of t'_M with no inner cut-point states, or to a *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 `env_move'` function.

Finally, $\hat{\rho}$ would reach a state $\hat{s}'$ matching s'. Since s' is a cut-point state, there would be an `assert(`$\neg start \vee \neg\beta_1$`)` command at the compatible label at the end of $\hat{\rho}$, i.e., at $l(\hat{s}')$. Since $s' \vDash \beta_1$, $\hat{s}'$ 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 $\hat{s}'$. Nevertheless, we get a violating computation $\hat{\rho}'$ in P'_M.

ψ Guarantees the Reachability of β_1 To conclude, we need to show that the second property of Definition 7.1.1 holds. That is, if $s \vDash \psi$, then there exists a computation of T in P from s to some s' satisfying β_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 ψ is computed as the weakest precondition in P_M, from the violation of some assertion `assert(`$\neg start \vee b$`)` to the activation `try_start`. Therefore, if $s \vDash \psi$, there exists a computation of t'_M in P_M' from s to some s'' satisfying $\neg b$. We already showed that the location ot the violated assertion, together with $\neg b$ are a promise of β_1. Hence there is a computation of T in P from s'' to some s' satisfying β_1. Concatenating these two computations, gives the desired result.

□

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 $Reach_{(\{t_2,t_3\})}(\alpha_1, \beta_1)$ was initiated, with $\alpha_1 \triangleq (a == 1 \wedge b == 1 \wedge c == 1)$ and $\beta_1 \triangleq (a + 2b > 15)$. Figure 7.2 presents the sequential program P_2 constructed to answer the query. The `try_start` havocs both a, which is written by t_1 and c, which is written by t_3 (and pc_{t_3} as well), as it should abstract any prefix in P. The `env_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 α_1 to β_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 α_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 `try_start` in Line 28 is chosen as the activation `try_start`. The following `env_move` havocs 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 `env_move` function.

```
 1  int a=1, b=1, c=1;
 2
 3  void t1() {
 4    int k1 = 0;
 5    while (k1 < 2) {
 6      a += b;
 7      k1++;
 8    }
 9    assert(a <= 15);
10  }
11  void t2() {
12    int k2 = 0;
13    while (k2 < 2) {
14      b += c;
15      k2++;
16    }
17  }
18  void t3() {
19    int k3 = 0;
20    while (k3 < 2) {
21      c += a;
22      k3++;
23    }
24  }
```

Figure 7.1: A variation of the fib_bench examples from the SV-Comp concurrency benchmark with three threads.

```
 1  int a=1, b=1, c=1;
 2  bool start = false;
 3  void environment_move() {
 4    if (start) {
 5      int c_COPY = c, pc_t3_COPY = pc_t3;
 6      c = havoc_int();
 7      pc_t3 = havoc_int();
 8    }
 9  }
10  void try_start(int pc_aux) {
11    int a_COPY = a, c_COPY = c, pc_t3_COPY = pc_t3;
12    pc_t2 = pc_aux;
13    if (!start) {
14      c = havoc_int();
15      a = havoc_int();
16      pc_t3 = havoc_int();
17      if (*) {
18        start = 1;
19        assume(a == 1);
20        assume(b == 1);
21        assume(c == 1);
22      }
23    }
24  }
25
26  int main() {
27    int k2 = 0;
28    try_start(L13);
29    environment_move();
30    assert((!start_) || (a + 2*b <= 15));
31    while (k2 < 2) {
32      b += c;
33      try_start(L15);
34      environment_move();
35      assert((!start_) || (a + 2*b <= 15));
36      k2++;
37    }
38  }
```

Figure 7.2: The sequential program P_2 with both `try_start` and `env_move`' calls, for answering $\mathit{Reach}_{(\{t_2,t_3\})}(\alpha_1, \beta_1)$ with $\alpha_1 \triangleq (a == 1 \wedge b == 1 \wedge c == 1)$ and $\beta_1 \triangleq (a + 2b > 15)$.

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`$((\mathtt{pc}_i = l) \Rightarrow b)$. 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 $[(\mathtt{pc}_i = l) \wedge \neg b]$, 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 $\mathtt{pc}_i$ 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. $s \models (\texttt{pc}_i = l)$. 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 $Reach_{(t_E)}(\alpha, \beta)$ returned $\psi \neq$ *FALSE*, and a new assertion `assert(`$\neg\psi$`)` was added right before an `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 `env_move` call, and adding the new assertion `assert(`$\neg\psi$`)` at the end.

In this case, upon encountering a violating path, we try to validate all uses of `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 α only over-approximates the reachable states before the `env_move` call. Hence, though ψ is guaranteed to intersect α, 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 `env_move` in the composed counterexample. Since we only return ”Unsafe” upon encountering a counterexample with no `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 $\beta = wp(\hat{\pi}, \neg b)$ for some path $\hat{\pi}$, ending at an assertion `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.

$\beta = wp(\hat{\pi}, \neg b)$ has the property that for every states $\hat{s}$ satisfying β, 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 π, we obtain a more general formula β'. β' has the property that for every state $\hat{s}$ satisfying β', 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.

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

Without the optimization, the computed condition is $\beta \triangleq (a > 0 \wedge b > 0)$. 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 $\beta' \triangleq (a > 0)$. 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 (`$\alpha(W$`_old)) assume(`$\neg\beta$`)`. The relevant formulas α and β 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 α to β. 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 α and β. Those variables are in fact constant in the context of t_E. Thus, by the absence of computations of t_E from α to β we can conclude that *for every* assignment to the (unchanged) variables in C, there is no computation from α to β.

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 α and β, 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 α and β), and the other two are applied after the query returned $\psi = \mathit{FALSE}$.

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

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

For example, if $\alpha = (i == 1 \wedge j > 0)$, 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 α. With the optimized query, instead of learning that there is no computation from $\alpha = (i == 1 \wedge j > 0)$ 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 β has a conjunct p as a query invariant, and α contains a sub-formula of the form $q_1 \vee q_2$, 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 $\mathit{Reach}_{(t_E)}(\alpha', \beta')$ returned $\psi = \mathit{FALSE}$, since the nonexistence of reachable computations of t_E from α' to β' implies that there are no such computations from α to β as well. However, if $\psi' = \mathit{Reach}_{(t_E)}(\alpha', \beta') \neq \mathit{FALSE}$, we cannot simply add `assert(`$\neg\psi'$`)` as before. The reason is that query invariants are only guaranteed to hold, given the context (α and β) of the query. More specifically, the computed ψ' only guarantees that we can reach β'. But β' does not necessarily lead to β. Reaching β' guarantees that we reach β only for the specific α (and the query invariants) used for this query. Since we might reach the `env_move` that initiated the query using different computations in which α does not hold, it is unsound to add `assert(`$\neg\psi'$`)` before the `env_move` call. To resolve this, if a computation from α' to β' was found in P_E, the returned formula ψ is computed as the weakest precondition from the original β, and not from β'.

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 ψ' ensuring that we can reach $j > 4$, it will be unsound to add `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 β.

β Constants Replacement The next optimization can be used when α 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_old represents the value of the variables in U at the start of the computation.

Recall that the standard refinement of `env_move` uses a statement of the form `if (`$\alpha(W\_$`old))` `assume(`$\neg\beta$`)`. That is, we already have a method for arguing about variables at the beginning of the computation, but it was typically used just for α. In this case, we can omit the predicate $v == p(U)$ from α, and simply replace every other occurrence of v (in α or β) 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 γ, defined only over variables written by t_E, such that there is no reachable computation of t_E from γ to $\neg\gamma$. Recall that when a query $Reach_{(t_E)}(\alpha, \beta)$ returns *FALSE*, we learn that there is no reachable computation of t_E from α to β. We try to utilize such results to proactively search for additional relevant information, in the form of inductive environment properties. However, checking whether some γ is an inductive environment property (by computing $Reach_{(t_E)}(\gamma, \neg\gamma)$) 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 α' and β' still contain variables not written by t_E. In a typical case, α is a conjunction of several conditions. It might be the case that α 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 γ, we check if γ is an inductive environment property. We also save γ, so that we will not check it again if it reappears as a conjunct in some future α.

The second heuristic is applied when $\alpha = TRUE$, and β is a conjunction. When $\alpha = TRUE$, 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 β 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 γ was found, we use it to further refine the `env_move` function, by adding `if (`$\gamma(W\_$`old))` `assume(`γ`)`, additionally to the `if (`$\alpha(W\_$`old))` `assume(`$\neg\beta$`)` 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 $Reach_{(T)}(\alpha_1, \beta_1)$ 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 $Reach_{(T')}(\alpha_2, \beta_2)$ 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 $Reach_{(T)}(\alpha_1, \beta_1)$ from P_M, and later reconstructed to answer another query $Reach_{(T)}(\alpha'_1, \beta'_1)$. Any new assertion added to P'_M during the process of answering the first query, is in fact a "promise of β_1". That is, if the assertion is violated, it means that $start \wedge \beta_1$ can be reached. In the general case, $\beta'_1 \neq \beta_1$ and hence this information does not guarantee the reachability of β'_1, needed by the query $Reach_{(T)}(\alpha'_1, \beta'_1)$. Thus, it should be removed from P'_M.

However, the information which appears inside the `env_move`$'$ function, represents the absence of computations of T' from some α_2 to some β_2. Since this is a property of T', it remains correct regardless of the conditions (α_1, β_1) given by the query from P_M. Hence, information gathered within the `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 $Reach_{(T)}(\alpha_1, \beta_1) =$ *FALSE* for some environment query, originated from the analysis of P_M. Essentially, generalization is performed by guessing conditions α'_1, β'_1 s.t. $\alpha_1 \Rightarrow \alpha'_1$ and $\beta_1 \Rightarrow \beta'_1$, and checking whether still $Reach_{(T)}(\alpha'_1, \beta'_1) =$ *FALSE*. If this query is answered by Algorithm 7.1, it may initiate additional inner environment queries $Reach_{(T')}(\alpha_2, \beta_2)$. 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 $Reach_{(T)}(\alpha_1, \beta_1)$ returned

FALSE, it means that P'_M already has enough information to refute the transition from α_1 to β_1. We can therefore check if P'_M, by itself, can also refute the transition from α'_1 to β'_1. If so, we can generalize (α_1, β_1) to (α'_1, β'_1). Otherwise, this generalization call fails and we retreat to (α_1, β_1) (and might try another generalization). Note that in the latter case, it still might be the case that $Reach_{(T)}(\alpha'_1, \beta'_1)$ 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 $Reach_{(T)}(\alpha'_1, \beta'_1) =$ *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 [1]. 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.

[1]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.

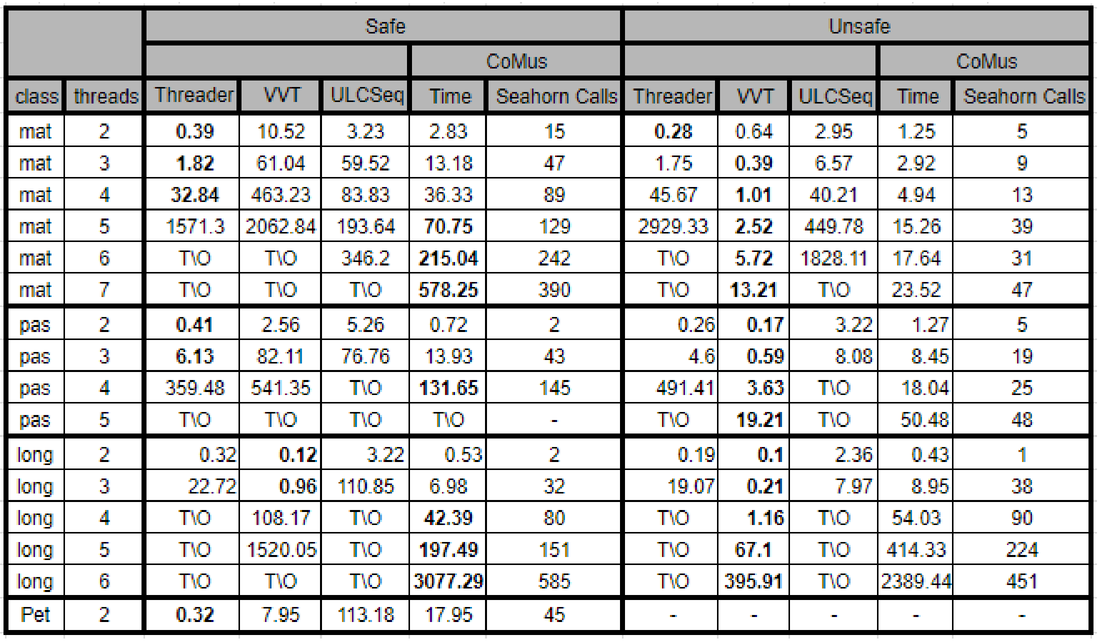
These algorithms have a natural definition for any finite number of threads. Typically, the verification becomes harder as the number of threads increases. For evaluation, we used programs with an increasing number of threads, and checked the influence of the number on the different tools. For each instance, 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 $t_0, \dots, t_{n-1}$ and variables $v_0, \dots, v_{n-1}$. Each thread t_i runs in a loop, and iteratively performs v_i+=$v_{(i+1(mod\ n))}$. 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}+=1 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



		Safe					Unsafe				
					CoMus					CoMus	
class	threads	Threader	VVT	ULCSeq	Time	Seahorn Calls	Threader	VVT	ULCSeq	Time	Seahorn Calls
mat	2	**0.39**	10.52	3.23	2.83	15	**0.28**	0.64	2.95	1.25	5
mat	3	**1.82**	61.04	59.52	13.18	47	1.75	**0.39**	6.57	2.92	9
mat	4	**32.84**	463.23	83.83	36.33	89	45.67	**1.01**	40.21	4.94	13
mat	5	1571.3	2062.84	193.64	**70.75**	129	2929.33	**2.52**	449.78	15.26	39
mat	6	T\O	T\O	346.2	**215.04**	242	T\O	**5.72**	1828.11	17.64	31
mat	7	T\O	T\O	T\O	**578.25**	390	T\O	**13.21**	T\O	23.52	47
pas	2	**0.41**	2.56	5.26	0.72	2	0.26	**0.17**	3.22	1.27	5
pas	3	**6.13**	82.11	76.76	13.93	43	4.6	**0.59**	8.08	8.45	19
pas	4	359.48	541.35	T\O	**131.65**	145	491.41	**3.63**	T\O	18.04	25
pas	5	T\O	T\O	T\O	T\O	-	T\O	**19.21**	T\O	50.48	48
long	2	0.32	**0.12**	3.22	0.53	2	0.19	**0.1**	2.36	0.43	1
long	3	22.72	**0.96**	110.85	6.98	32	19.07	**0.21**	7.97	8.95	38
long	4	T\O	108.17	T\O	**42.39**	80	T\O	**1.16**	T\O	54.03	90
long	5	T\O	1520.05	T\O	**197.49**	151	T\O	**67.1**	T\O	414.33	224
long	6	T\O	T\O	T\O	**3077.29**	585	T\O	**395.91**	T\O	2389.44	451
Pet	2	**0.32**	7.95	113.18	17.95	45	-	-	-	-	-

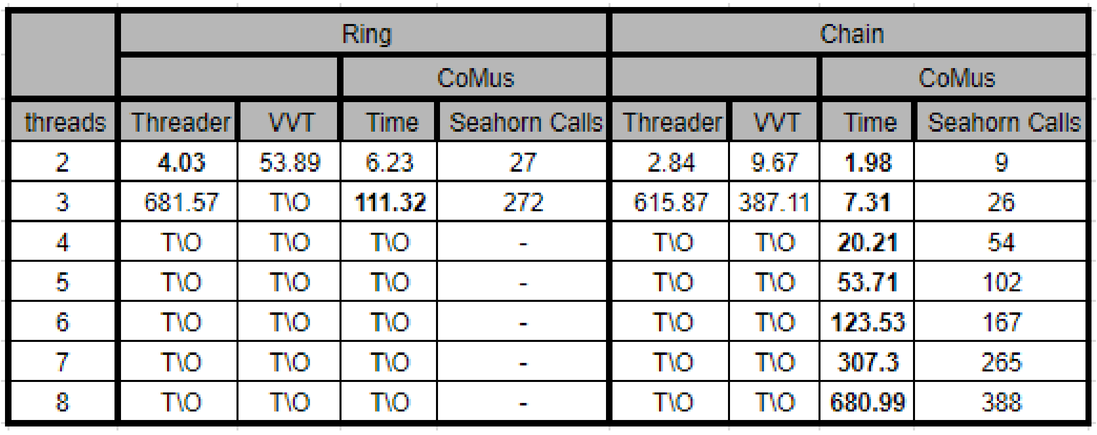
Figure 9.1: Run times [secs] for all four tools for verifying concurrent dynamic programming algorithms.

timeout was set to 1200 seconds.

For the ring case, all tools fail to verify programs with ≥ 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.



	Ring				Chain			
			CoMus				CoMus	
threads	Threader	VVT	Time	Seahorn Calls	Threader	VVT	Time	Seahorn Calls
2	**4.03**	53.89	6.23	27	2.84	9.67	**1.98**	9
3	681.57	T\O	**111.32**	272	615.87	387.11	**7.31**	26
4	T\O	T\O	T\O	-	T\O	T\O	**20.21**	54
5	T\O	T\O	T\O	-	T\O	T\O	**53.71**	102
6	T\O	T\O	T\O	-	T\O	T\O	**123.53**	167
7	T\O	T\O	T\O	-	T\O	T\O	**307.3**	265
8	T\O	T\O	T\O	-	T\O	T\O	**680.99**	388

Figure 9.2: Run times [secs] for fib_bench programs with ring topology vs. chain topology.

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 transofrmed 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

[1] J. Alglave, D. Kroening, and M. Tautschnig. Partial orders for efficient Bounded Model Checking of concurrent software. In *Computer Aided Verification (CAV)*, volume 8044 of *LNCS*, pages 141–157. Springer, 2013.

[2] R. Bellman. *Dynamic programming*. Courier Corporation, 2013.

[3] E. Bendersky. https://github.com/eliben/pycparser.

[4] D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, pages 184–190, 2011.

[5] A. Biere, C. Artho, and V. Schuppan. Liveness checking as safety checking. *Electronic Notes in Theoretical Computer Science*, 66(2):160–177, 2002.

[6] A. R. Bradley. Sat-based model checking without unrolling. In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, pages 70–87, 2011.

[7] J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. Learning assumptions for compositional verification. In *TACAS*, pages 331–346, 2003.

[8] A. Cohen and K. S. Namjoshi. Local proofs for global safety properties. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, pages 55–67, 2007.

[9] B. Cook, A. Podelski, and A. Rybalchenko. Proving thread termination. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, pages 320–330, 2007.

[10] L. De Moura and N. Bjørner. Z3: An efficient smt solver. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[11] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[12] K. A. Elkader, O. Grumberg, C. S. Păsăreanu, and S. Shoham. Automated circular assume-guarantee reasoning. In *International Symposium on Formal Methods*, pages 23–39. Springer, 2015.

[13] C. Flanagan, S. N. Freund, and S. Qadeer. Thread-modular verification for shared-memory programs. In *Programming Languages and Systems, 11th European Symposium on Programming, ESOP 2002, held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, pages 262–277, 2002.

[14] C. Flanagan and S. Qadeer. Thread-modular model checking. In *Model Checking Software, 10th International SPIN Workshop. Portland, OR, USA, May 9-10, 2003, Proceedings*, pages 213–224, 2003.

[15] C. Flanagan and S. Qadeer. Transactions for software model checking. *Electr. Notes Theor. Comput. Sci.*, 89(3):518–539, 2003.

[16] I. Gavran, F. Niksic, A. Kanade, R. Majumdar, and V. Vafeiadis. Rely/guarantee reasoning for asynchronous programs. In *26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015*, pages 483–496, 2015.

[17] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.

[18] G. Gueta, C. Flanagan, E. Yahav, and M. Sagiv. Cartesian partial-order reduction. In *Model Checking Software, 14th International SPIN Workshop, Berlin, Germany, July 1-3, 2007, Proceedings*, pages 95–112, 2007.

[19] H. Günther, A. Laarman, and G. Weissenbacher. Vienna verification tool: IC3 for parallel software - (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, pages 954–957, 2016.

[20] A. Gupta, C. Popeea, and A. Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 331–344. ACM, 2011.

[21] A. Gupta, C. Popeea, and A. Rybalchenko. Threader: A constraint-based verifier for multi-threaded programs. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 412–417. Springer, 2011.

[22] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The seahorn verification framework. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 343–361. Springer, 2015.

[23] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. You assume, we guarantee: Methodology and case studies. In *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings*, pages 440–451, 1998.

[24] S. K. Lahiri, A. Malkis, and S. Qadeer. Abstract threads. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 231–246. Springer, 2010.

[25] A. Lal and T. W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, pages 37–51, 2008.

[26] K. R. M. Leino and P. Müller. A basis for verifying multi-threaded programs. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, pages 378–393, 2009.

[27] A. Malkis, A. Podelski, and A. Rybalchenko. Thread-modular verification is cartesian abstract interpretation. In *Theoretical Aspects of Computing - ICTAC 2006, Third International Colloquium, Tunis, Tunisia, November 20-24, 2006, Proceedings*, pages 183–197, 2006.

[28] K. L. McMillan. Lazy abstraction with interpolants. In *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, pages 123–136, 2006.

[29] T. L. Nguyen, B. Fischer, S. La Torre, and G. Parlato. Unbounded lazy-cseq: A lazy sequentialization tool for C programs with unbounded context switches - (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 461–463, 2015.

[30] T. L. Nguyen, B. Fischer, S. La Torre, and G. Parlato. Lazy sequentialization for the safety verification of unbounded concurrent programs. In *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings*, pages 174–191, 2016.

[31] C. S. Pasareanu, M. B. Dwyer, and M. Huth. Assume-guarantee model checking of software: A comparative case study. In *SPIN*, volume 99, pages 168–183. Springer, 1999.

[32] C. S. Pasareanu, D. Giannakopoulou, M. G. Bobaru, J. M. Cobleigh, and H. Barringer. Learning to divide and conquer: applying the L* algorithm to automate assume-guarantee reasoning. *Formal Methods in System Design*, 32(3):175–205, 2008.

[33] G. L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, 1981.

[34] A. Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and models of concurrent systems*, pages 123–144. Springer, 1985.

[35] C. Popeea and A. Rybalchenko. Compositional termination proofs for multi-threaded programs. In *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7214 of *Lecture Notes in Computer Science*, pages 237–251. Springer, 2012.

[36] C. Popeea and A. Rybalchenko. Threader: A verifier for multi-threaded programs - (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7795 of *Lecture Notes in Computer Science*, pages 633–636. Springer, 2013.

[37] C. Popeea, A. Rybalchenko, and A. Wilhelm. Reduction for compositional verification of multi-threaded programs. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*, pages 187–194. IEEE, 2014.

[38] S. Qadeer and D. Wu. Kiss: Keep it simple and sequential. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 14–24, New York, NY, USA, 2004. ACM.

[39] I. Rabinovitz and O. Grumberg. Bounded model checking of concurrent programs. In *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, pages 82–97, 2005.

[40] A. J. Robinson and A. Voronkov. *Handbook of automated reasoning*, volume 1. Elsevier, 2001.

[41] N. Sinha and E. M. Clarke. Sat-based compositional verification using lazy learning. In *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, pages 39–54, 2007.

[42] E. Tomasco, O. Inverso, B. Fischer, S. La Torre, and G. Parlato. Verifying concurrent programs by memory unwinding. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 551–565, 2015.

[43] E. Tomasco, T. L. Nguyen, O. Inverso, B. Fischer, S. La Torre, and G. Parlato. Lazy sequentialization for TSO and PSO via shared memory abstractions. In *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, pages 193–200, 2016.

[44] B. Wachter, D. Kroening, and J. Ouaknine. Verifying multi-threaded software with impact. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 210–217. IEEE, 2013.

[45] M. Zheng, J. G. Edenhofner, Z. Luo, M. J. Gerrard, M. S. Rogers, M. B. Dwyer, and S. F. Siegel. CIVL: applying a general concurrency verification framework to c/pthreads programs (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, pages 908–911, 2016.

(המשתמש בבדיקת מודל חסומה לצורך חיפוש שגיאות) מציג את התוצאות הטובות ביותר. עם זאת, CoMuS עדיין מציג תוצאות טובות יותר משל החוטים האחרים.

ההשוואה השנייה בדקה תכניות דומות תחבירית, אשר נבחרו בהשראת התכניות מסדרת fib_bench* מהתחרות SV-COMP. לצורך ההשוואה, בדקנו תכניות בהן זרימת המידע בין החוטים השונים משרה טופולוגיה של טבעת, וכן תכניות בהן שינוי תחבירי קל "שובר" את הטבעת, ומשרה טופולוגיה של שרשרת (כלומר, ניתן לסדר את החוטים בסדר היררכי מלא, בו החוטים תלויים רק בעוקבים שלהם). עבור תכניות אלו, ביצענו השוואה של אימות בלבד. עבור טופולוגיית הטבעת, כל הכלים נכשלו באימות תכניות עם ארבעה חוטים ומעלה. כמו כן, Threader פעל בצורה דומה בשני המקרים, ואילו VVT הציג שיפור עבור טופולוגיית השרשרת (בה התלות קטנה יותר). לעומת זאת, CoMuS, שתוכנן לנצל היררכיה, הציג שיפור משמעותי לעומת שאר הכלים עבור טופולוגיית השרשרת, והצליח לאמת תכניות בעלות שמונה חוטים. UL-CSeq לא הציג ביצועים טובים עבור אף אחת מהתכניות במשפחה זו.

ההשוואה השלישית בדקה את הכלים על תכניות מקביליות כלליות. על תכניות אלו, לרוב הביצועים של CoMuS היו פחותים משל הכלים האחרים.

לסיכום, בעבודה זו פיתחנו שיטה אוטומטית, מודולרית והיררכית להוכחת או הפרכת טענות נכונות בתכניות מקביליות, ע"י ניצול בודקי מודל סדרתיים. השיטה תומכת בתכניות עם מרחב מצבים אינסופי. היא נאותה ולא חסומה. מימשנו את השיטה בכלי ניסיוני בשם CoMuS, אשר בולט לטובה בהשוואה מול בודקי מודל מובילים אחרים, עבור מחלקה מסוימת של תכניות, כפי שתוארו לעיל. בעתיד, אנו מתכננים לנצל מידע פנימי אשר נאסף ע"י בודק המודל הסדרתי (למשל, SeaHorn) כדי להאיץ יותר את תוצאותינו.

העובדה שהאלגוריתם שלנו מייצר תכניות סדרתיות תקניות בבדיקות הביניים שלו, מאפשרת לנו להשתמש בכל בודק מודל "מהמדף". בפרט, אנו מסוגלים לטפל בתכניות מקביליות עם מרחב מצבים אינסופי, בתנאי שבודק המודל הסדרתי תומך בתכניות כאלו.
מימשנו את השיטה בכלי ניסיוני בשם "CoMuS", ובחנו את הכלי כנגד כלים מובילים עבור אימות בלתי חסום של תכניות C מקביליות. אנו משתמשים ב- SeaHorn כבודק המודל הסדרתי שלנו. SeaHorn מקבל תכניות C המסומנות ב"טענות" (assertions) ובודק האם אחת הטענות ניתנת להפרה. במידה וכן, הוא מחזיר מסלול המוביל להפרה. אחרת, הוא מודיע כי אף הפרה לא מתרחשת.
אמנם, שיטתנו מעוצבת כך שתעבוד לכל תכנית מקבילית, אך הניסויים שביצענו הראו כי היא עובדת טוב במיוחד עבור תכניות בהן החוטים מסודרים בטופולוגית שרשרת, וכאשר כל חוט תלוי רק במידע המיוצר ע"י החוט העוקב בשרשרת. סידור כזה משרה היררכיה טבעית בה החוט הראשון בשרשרת הוא החוט הראשי, החוט השני הוא החוט הראשי בסביבה, וכן הלאה. מבנה זה מופיע במקרים רבים במימושים מקביליים של אלגוריתמי תכנון דינמי.

לסיכום, התרומות העיקריות של עבודתנו הן להלן:
- אנו מציגים גישה מודולרית חדשה לאימות תכניות מקביליות, אשר מבצעת רדוקציה מאימות התכנית המקבילית לסדרה של בעיות אימות עבור תכניות סדרתיות. לכן, ניתן להשתמש בכל בודק-מודל עבור תכניות סדרתיות.
- הגישה שלנו מנצלת נקודת מבט היררכית, כאשר כל חוט לומד לגבי החוטים הבאים בהיררכיה, ומצויד בהנחות ע"י החוטים הקודמים בהיררכיה על מנת להנחות את למידתו.
- המידע הדרוש לגבי הסביבה של חוט נאגר בקוד התכנית, באופן אוטומטי ועצל, במהלך ריצת האלגוריתם.
- מימשנו את שיטתנו והראינו שכאשר מספר החוטים גדל, היא מציגה ביצועים טובים יותר מכלים מוכרים עבור תכניות בהן יש מבנה היררכי, כגון מימושים מקביליים של אלגוריתמי תכנון דינמי.

מימשנו את שיטתנו בכלי בשם CoMuS, אשר משתמש ב- SeaHorn כבודק מודל סדרתי. השווינו את CoMuS מול שלושה כלים מוכרים אחרים: VVT, Threader ו- UL-CSeq. T. מבין הכלים האחרים, Threader פועל בצורה הדומה ביותר לשיטתנו. זהו כלי מודולרי, אשר גם כן מבצע הפשטה להתנהגויות סביבה, אך הוא מתייחס לכל החוטים בצורה סימטרית, ומחשב הפשטה זו לכל זוג חוטים (כיצד חוט מסוים יכול להשפיע על חוט אחר). הכלים VTT ו- UL-CSeq אינם מודולרים. אלו הכלים שדורגו במקום הגבוה ביותר באימות תכניות מקביליות, מבין הכלים המבצעים בדיקת מודל לא חסומה, בתחרות SV-COMP, בשנים 2016 ו- 2017 (בהתאמה).
ביצענו שלוש השוואות לצורך הערכת הכלים. הראשונה, בודקת את התנהגות הכלים על שלושה אלגוריתמי תכנון דינמי מקביליים. לאלגוריתמים אלו יש הגדרה טבעית לכל מספר סופי של חוטים. באופן טיפוסי, האימות הוא קשה יותר כאשר מספר החוטים עולה. לכל אחד משלושת האלגוריתמים, בדקנו את הכלים על תכניות עם מספר משתנה של חוטים, ובחנו כיצד מספר זה משפיע על משך האימות. שלושת האלגוריתמים שנבחרו מציגים מקביליות לא טריוויאלית (כלומר, החוטים אכן רצים במקביל, וחוטים לא נדרשים להמתין שחוטים בהם הם תלויים יסיימו את ריצתם לחלוטין) וכן כוללים מבנה היררכי ברור, בו ניתן לסדר את החוטים לפי סדר מלא, והחוטים תלויים אך ורק בעוקבים שלהם. לצורך ההשוואה, בדקנו גם תכניות בטוחות וגם תכניות שכללו שגיאה. התוצאות מראות יתרון ברור ל- CoMuS באימות תכניות, כאשר מספר החוטים עולה. עבור מציאת שגיאות, VTT

תקציר מורחב

אימות תכניות מקביליות היא בעיה קשה ביותר. בנוסף לאתגרים המובנים באימות תכניות סדרתיות, נוסף הצורך להתחשב במספר גבוה (לרוב לא חסום) של חישובים, בהם החוטים רצים לסירוגין בסדרים משתנים. עבור תכניות כאלו, מתבקש לנסות ולנצל את המבנה המודולרי שלהן לצורך אימותן. אלא שלרוב, תכונה של התכנית כולה לא ניתנת לחלוקה לקבוצת תכונות כך שכל אחת מקומית עבור אחד החוטים. לכן, הוכחת התכונה עבור אחד החוטים דורשת ידע מסוים לגבי האינטראקציה שלו עם סביבתו.

בעבודה זו, אנו מפתחים גישה חדשה אשר מנצלת את שפע העבודות בנושא אימות תכניות סדרתיות למטרת אימות *מודולרי* של תכניות מקביליות. השיטה שלנו מבצעת באופן אוטומטי רדוקציה מהבעיה של אימות תכנית מקבילית, לסדרת בעיות אימות של תכניות סדרתיות. לפיכך, נוכל להרוויח מכל התפתחות היסטורית, וכן מהתפתחויות עתידיות, בתחום של אימות סדרתי.

הגישה שלנו מודולרית, במובן שכל משימת אימות סדרתית דומה בקירוב לאימות של חוט בודד, עם אי אילו נתונים נוספים לגבי הסביבה בה הוא פועל. נתונים אלו מתגלים באופן *אוטומטי* במהלך ריצת האלגוריתם, לפי הצורך.

פן ייחודי של גישתנו, הוא שהיא מנצלת נקודת מבט היררכית לגבי התכנית, לפיה אחד החוטים נחשב ל"חוט הראשי", והשאר נחשבים לסביבתו. אנו מנתחים את החוט הראשי באמצעות אימות סדרתי, כאשר עבור נאותות, כל התערבויות הסביבה מופשטות (over-approximated), ע"י שגרה בשם "צעד-סביבה". השגרה תקרא ע"י החוט הראשי בכל שלב בו יש צורך לשקול החלפת הקשר (context switch).

תחילה, שגרת "צעד-סביבה" כותבת ערכים אי-דטרמיניסטיים (באמצעות פקודת havoc) לכל המשתנים המשותפים של החוט הראשי והסביבה. במהלך ריצת האלגוריתם, השגרה מעודנת כדי לייצג את הסביבה בצורה מדויקת יותר.

כאשר בודק המודל הסדרתי מגלה הפרה של טענת נכונות בחוט הראשי, הוא מחזיר מסלול אשר מוביל להפרה זו. המסלול עשוי לכלול קריאות ל"צעד סביבה", ובמקרה זה ההפרה עשויה להיות מדומה (כלומר, כזו שלא קיימת בתכנית המקורית), עקב ההפשטה של הסביבה. לכן, האלגוריתם יוזם *שאילתות* לסביבה של החוט הראשי, שמטרתן לבדוק האם התערבויות *מסוימות*, אשר נצפו לאורך המסלול המפר, אכן אפשריות בסביבה. כאשר התערבות מתגלה כבלתי-אפשרית, השגרה "צעד סביבה" מעודנת כך שלא תכלול התערבות זו.

לבסוף, "צעד סביבה" נעשית מדויקת מספיק על מנת לאפשר אימות מלא של התכונה הרצויה באמצעות החוט הראשי וההפשטה של הסביבה. לחילופין, האלגוריתם עשוי לגלות דוגמא נגדית אמיתית (שאינה מדומה) בחוט הראשי.

השאילתות נבדקות עבור הסביבה (שעשויה לכלול בעצמה מספר חוטים) באותו אופן מודולרי. לכן, אנו מקבלים *אימות מודולרי היררכי*. האופי ההיררכי של השיטה שלנו מבטיח שבמהלך ריצת האלגוריתם, כל חוט לומד מידע רלוונטי לגבי החוטים הבאים בהיררכיה, ומצויד בהנחות מחוטים קודמים בהיררכיה על-מנת להנחות את למידתו.

השיטה שלנו אוטומטית לחלוטין ומבצעת אימות בלתי-חסום. כלומר, היא יכולה גם להוכיח נכונות וגם למצוא שגיאות בתכניות מקביליות. היא עובדת ברמת קוד-המקור של תכניות. המידע המתגלה עבור הסביבה נאגר בקוד של החוט הראשי, בצורה של טענות ("assertions") והנחות (" assumptions") בתוך השגרה "צעד סביבה".

המחקר בוצע בהנחייתן של פרופסור ארנה גרימברג וד"ר שרון שוהם, בפקולטה למדעי המחשב.

תודות

בראש ובראשונה, אני רוצה להודות למנחה שלי, פרופ. ארנה גרימברג, על היותה גם מנחה מדהימה וגם אדם נפלא. תודה על הפגישות השבועיות מעוררות ההשראה, על כך שתמיד עזרת לי להשתפר בעבודתי, ועל כך שעודדת אותי לחקור ולהבין מהו מחקר. תודה על כך שדלתך תמיד הייתה פתוחה, עזרת להפוך את חווית הלימודים למהנה מאוד. זה היה כבוד מאוד גדול עבורי לעבוד איתך וללמוד ממך.
אני רוצה להודות למנחה הנוספת שלי, ד"ר שרון שוהם, על חלקה העצום בעבודה זו. תודה על המסירות שלך, ועל היכולת להפוך כל פגישה לפוריה עם רעיונות חדשים וחשיבה חדה. זה תמיד היה מהנה עבורי לפגוש אותך גם בת"א וגם בטכניון. את מנחה נפלאה ואדם נפלא, ואני אסיר תודה על עזרתך והכוונתך.
אני רוצה להודות להורים שלי, לינה ולב, על אהבתם ותמיכתם האינסופית. תודה שלימדתם אותי לכוון גבוה, ועל כך שהייתם שם בשבילי בכל שלב בחיי.
לבסוף, אני רוצה להודות לחברה שלי ורוניקה. ליווית אותי במהלך התקופה הזו גם בימים הטובים יותר בהם הצלחתי להתקדם, וגם בימים טובים פחות ומתסכלים. תמיד תמכת בי ועזרת לי להמשיך, ואהיה אסיר תודה על כך לנצח.

אני מודה לטכניון על התמיכה הכספית הנדיבה בהשתלמותי.

אימות מודולרי של תכניות מקביליות באמצעות בדיקת מודל סדרתית

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
מגיסטר למדעים במדעי המחשב

דן רסין

הוגש לסנט הטכניון - מכון טכנולוגי לישראל

כסלו התשע"ח   חיפה   נובמבר 2017

אימות מודולרי של תכניות מקביליות באמצעות בדיקת מודל סדרתית

דן רסין