Automatically Verifying Temporal Properties of Heap Programs with Cyclic Proof

Gadi de Leon Tellez Espinosa

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

of

University College London.

Department of Computer Science

University College London

February 2, 2019

I, Gadi de Leon Tellez Espinosa, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the work.

Abstract

This work proposes a deductive reasoning approach to the automatic verification of temporal properties of pointer programs, based on *cyclic proof*. We present a proof system whose judgements express that a program has a certain temporal property, given a suitable precondition, and whose rules operate directly on the temporal modalities as well as symbolically executing programs. Cyclic proofs in our system are, as elsewhere, finite rooted proof graphs subject to a natural, decidable soundness condition, encoding a form of proof by infinite descent.

We present two variants of our proof system, one for CTL (branching time) properties and one for LTL (linear time) properties, and show them both to be sound. We have implemented both variants in the CYCLIST theorem prover, yielding an automated tool that is capable of automatically discovering proofs of temporal properties of our programs. Evaluation of our tool on well-known benchmarks in the model checking community indicates that our approach is viable, and offers an interesting alternative to traditional model checking techniques.

Impact Statement

For the last 20+ years, most of the research in verification of temporal logic has been focused on model checking while only a minor part has been dedicated to deductive verification. With our work we have demonstrated that, in practical application terms, software verification of temporal properties based on deductive verification has a comparable performance when compared model checking approaches. Achieving similar capabilities and performance to those of model checking tools in a much shorter span of time, with a comparably lower manpower, seems to indicate that deductive verification has a place in current temporal verification of software. The work presented in this thesis could lay down the basis on a resurgent interest in the study of deductive verification techniques for temporal logic.

A substantial component of our work is the implementation of an automated tool for the verification of temporal properties of heap-aware programs. The readily availability of this tool could be put to use in verifying a wide range of software components. Moreover, the provision of its source code facilitates its improvement and extensions, empowering its potential application to software programs beyond the scope of those exercised so far.

Finally, the class of computer programs considered for our study, in particular the use of heap memory, constitutes a significant segment of systems found in commercial software, spanning over several domains of science and technology. As such, our research could have an immediate impact on the quality and safety of ubiquitous computer systems, greatly increasing our confidence in its correctness and minimising their probability of failure.

Acknowledgements

I am infinitely grateful to my supervisor James Brotherston who has provided invaluable guidance, support and encouragement. Words cannot describe how much you have impacted my life inside and outside of academia. Thanks for your time and for guiding me towards producing this thesis.

A very special gratitude goes to all the people involved in the vibrant community around PPLV research group. Being surrounded by leaders in the field of verification was greatly encouraging. I will always appreciate the inspiring talks and discussions surrounding the group.

I am also grateful to my friends for keeping me sane during this period of my life. After all, not all in life is about career advancement. You are a very dear reminder of that, for which I am glad.

Most importantly, I would like to express my biggest gratitude to my family. It is invaluable to feel the support of the most amazing persons I will ever know. To my siblings, parents, grandparents, aunties, uncles, cousins... I know I could have not accomplished this without your eternal love and support. I love you all.

Finally, a very special mention to my departed grandfather Fernando Tellez Santana. Your life is the greatest example of commitment, dedication, discipline, and, most importantly, love. Here's to you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Formal methods constitute a mathematical approach for ensuring the reliability and correctness of computer programs. The central aim of formal methods is to be able to rigorously guarantee the behaviour of a given computer system. To this effect, a set of formalisms and mathematical tools for modelling, specifying and reasoning about systems is devised.

Naturally, the reliability and correctness of a system only make sense with respect to a given set of requirements, which gives rise to the notion of *specification*: a description of the desired behaviour of the system expressed in some formal language expressive enough to state the behaviour.

Historically, there has been a wide variety of proposed languages to serve as the formal specification language for computer systems. Among these, *temporal logic* has gathered a lot of attention for its ability to express a wide range of system properties.

1.1 Temporal logic

In the late 70's, Pnueli observed that the common notions in the early work of system specifications such as termination, partial and total correctness were not suitable for expressing properties of a class of computer systems referred to as *reactive systems* [76]. These systems are characterised by their ongoing interaction with their environment. Due to their non-terminating nature, Pnueli observed that a different approach was needed to specify these systems.

Properties of reactive systems, instead, involve notions of *invariance* (an assertion always holds in every state of the execution), *eventuality* (an assertion eventually holds in some reachable future state) and *fairness* (if a process is enabled, it should eventually be scheduled for execution). Temporal logics were introduced by Pnueli as a formalism for characterising such properties.

One of the key aspects of temporal logics is their underlying model of time. Two possible views regarding the nature of time induce a categorisation of temporal logics into *linear time* and *branching time* logics. In linear time temporal logics, time is treated as if each moment in time has a unique possible future. In branching temporal logics, each moment in time may split into many possible futures. Linear time logic formulas are interpreted over linear structures and are regarded as describing the behaviour of a single computation of a program. Conversely, the structures over which branching temporal logics are interpreted have a tree-like shape, where each branch corresponds to a possible execution path.

In their various forms, temporal logics are commonly composed of a non-temporal part for specifying basic properties of states, called *atomic properties*, plus a set of temporal operators for specifying temporal properties. Atomic properties can be checked by observing a specific program state, whereas verifying temporal properties involves investigating *execution paths* (i.e. sequences of states).

The most commonly used temporal operators are denoted by X,U,F,G with the following intuitive meaning:

Xp	:	Assertion p will hold in the next state.
pUq	:	Assertion p will hold until assertion q holds.
Fp	:	Assertion p will hold at some point in the future.
Gp	:	Assertion p holds always.

The level of expressivity induced by these temporal operators make temporal logic a popular and widely studied specification formalism; a wide variety of *safety* (“something bad cannot happen”) and *liveness* properties (“something good eventually happens”) can be captured in these languages [68].

Establishing a specification in a formal language that describes the desired behaviour is a critical aspect of any formal method but is not sufficient to *ensure* a certain behaviour. Rather than simply constructing specifications and models, one is interested in proving properties about them; this is the realm of *software verification*.

There exists a wide range of approaches for guaranteeing the correctness of computer programs, that vary in the choice of formal language in which the specification is described, and in the different techniques used to show the program has correct behaviour. Depending on these factors, most approaches to software verification can be classified into two large families: *model checking* and *deductive verification*.

1.1.1 Model checking

Historically, perhaps the most popular approach to ensuring that a program exhibits a given temporal behaviour has been *model checking*, where one first builds an abstract model that over-approximates all possible executions of the program, and then checks that the desired temporal property holds in this model (see e.g. [41, 38, 32]).

Model checking is a technique for the verification of finite state systems developed in the early 80's by Clarke et.al. [34]. Soon after the introduction of temporal logics, it was observed that temporal properties can be checked automatically for *finite* computations [37]. A procedure which checks if a computer program is a model of a temporal logic formula is called a *model checker*.

The idea of model checking is that the expected properties of the model are expressed by formulae of a temporal logic, and a *symbolic execution* algorithm is used to traverse the model in its entirety so as to check whether all possible states in the execution of a program satisfy those properties. The set of all program states is called the *state space*.

Algorithms for *concrete enumerative* model checking essentially traverse the graph of program states and transitions using various graph search techniques. The term "concrete" indicates that the technique represents program states exactly. The term “enumerative” indicates that these methods manipulate individual states of the program.

The *execution-based* model checking approach is a special case of enumerative model checking. This approach uses the runtime system of a programming language implementation to carry out enumerative state space exploration. The idea is that the outcome of a program execution is determined by the inputs from the environment and the scheduling choices made by the scheduler. Therefore, the set of all behaviours can be explored by analysing the behaviour of the process under all possible inputs and schedules. A variety of tools have emerged as the result of this approach, such as VERISOFT[59], JavaPathFinder[98], CMC[79] and Chess[80].

A serious drawback of enumerative state representation is the *state explosion* problem, in which the transition graph grows exponentially in the size of the system. The immediate consequence of this problem is that no matter how efficient the traversing algorithm is, the exploration of the state space soon becomes intractable. Different techniques have been studied to tackle this problem, among which *reduction-based* methods are a popular choice.

Reduction-based techniques compute equivalence relations on the program behaviours, and explore one candidate from each equivalence class. Reduction-based techniques include *partial-order reduction* [95, 58], *symmetry reduction* [35, 55, 62, 90] and minimisation based on behavioural equivalences such as simulation or bisimulation [14, 70, 26]. Partial order reductions are mainly applicable to parallel threads of execution, where they take advantage of the notion of independence between threads. That is, if two transitions in parallel threads access independent sets of variables, the final state reached after executing both transitions is independent of the order in which the transitions were executed. Symmetry reduction determines symmetries in the program, and explores one element from each symmetry class. Behavioural equivalences such as similarity and bisimilarity construct a quotient graph that preserves reachability and then performs reachability analysis on the quotient.

Despite the efforts of reduction-based techniques to lower the size of the state space, in practice, their use is still often hampered by severe state space explosion. This limitation of enumerative model checking led to research on *symbolic* algorithms which manipulate (representations of) *sets* of states, rather than individual states, and perform state exploration through the symbolic transformation of these representations. For example, the constraint $1 < x \leq 10 \wedge 1 \leq y \leq 8$ represents the set of all states over variables $\{x, y\}$ satisfying the constraint. Hence, the constraint implicitly represents the list of all 80 states that would be enumerated in enumerative model checking.

The power of symbolic techniques comes from advances in the performance of *constraint solvers* that underlie effective symbolic representations for propositional logics [89, 53], binary decision diagrams (BDDs) [23] and, more recently, combinations of first order theories [82, 47].

BDDs have been instrumental in scaling hardware model checkers to extremely large state spaces. Nevertheless, each boolean operation and existential quantification of a single variable can be quadratic in the size of the BDD, and the size of the BDD can be exponential in the number of variables in the worst case. Moreover, the size of the BDD is sensitive to the order in which variables appear in the formula. Finally, many functions do not have a feasible BDD representation. This is the symbolic analogue of the state explosion problem, and has been a major research direction in model checking [65].

As in enumerative model checking, trading off soundness for effective bug finding is a common approach in symbolic model checking. A popular approach is called *bounded model checking* (BMC), a technique that unrolls the symbolic representation of a program for a fixed number of steps.

Tools for bounded model checking come in two flavours. The first, including CMBC [33], F-SOFT [64], SATURN [99], and Calysto [5] generate constraints in propositional logic and use SAT solvers to discharge the constraints. Scalability of this technique depends on the scalability of the underlying SAT solver, as well as carefully tuned heuristics which keep the size of the constraints small. The second class of tools generates constraints in an appropriate first order theory and uses decision procedures for such theories [78, 3]. The basic algorithm is identical to SAT-based bounded model checking, but the constraints are interpreted over more expressive theories.

Work on symbolic model checking has led to verification tools that can work on infinite state systems [1, 2] including concurrent programs [24, 25, 48]. Exploring all possible states of an infinite execution is necessarily uncomputable. However, the use of heuristics and semi-procedures sometimes allows us to verify such systems. For infinite state programs, symbolic model checking might not terminate, or take an unfeasible amount of time and/or memory to terminate. *Abstract model checking* trades off precision of the analysis for efficiency. In abstract model checking, model checking is performed on an *abstract domain* which captures some but not all the information about an execution [42]. Examples of common abstract domains include:

- *polyhedral domains*, which have been successfully used to check for array bounds [44];
- *predicate abstraction*, the underlying technique behind well-known model checkers SLAM [7] and BLAST [12];
- *control abstraction*, which (as its name suggests) focuses on merging different execution paths into equivalence classes [12]; and
- combinations of the above, with most common model checking tools following this approach, e.g. IMPACT [77], ASTREE [43] and BLAST [12].

In general, due to the over-approximation generated by the use of abstractions, abstract model checking generates *false negatives*; i.e., the abstract analysis can return a counterexample even though the program is correct. In this case, there exist techniques to automatically *refine* the abstract domain, that is, construct a new abstract domain that represents strictly more sets of concrete program states. The intent is to provide a more precise analysis which rules out the current counterexample and possibly others. This iterative strategy was proposed as *localisation reduction* in [67] and generalised to *counterexample-guided refinement* in [6, 36].

The input to the counterexample analysis algorithm is a path in the control-flow graph ending in the error location. The path represents a possible counterexample produced by abstract analysis. The first step of the algorithm constructs a logical formula, called the *trace formula* of the path, such that the formula is satisfiable if the path is executed by the concrete program. Then, a decision procedure is used to check if the trace formula is satisfiable. If satisfiable, the path is reported as a concrete counterexample to the property. If not, the proof of unsatisfiability is mined for new predicates that can rule out the current counterexample when the abstract domain is augmented with these predicates.

Improving the elimination of individual counterexamples, *interpolation-based refinement* was suggested in [60] to find predicates that capture the implicit relationships of counterexamples with the concrete program, hence eliminating multiple counterexamples at once. Another advantage of interpolation-based refinement is that it not only discovers new predicates but also determines the control locations at which these predicates are useful. Therefore, instead of keeping a global set of predicates, one can keep a map from locations to sets of predicates and perform predicate abstraction with respect to local set of predicates, resulting in an order of magnitude improvement in the running times of model checking.

Recent advances in abstraction refinement have led to the implementation of these techniques in a variety of model checkers. The SLAM model checker [7] was the first implementation of refinement model checking for C programs. The BLAST model checker [12] implements an optimisation to abstract refinement for constructing the abstract model on the fly and locally refining the model on demand. The MAGIC model checker implements a predicate abstraction that yields a finite state machine representing the program behaviour and infers new predicates which yield refined state machines [29]. F-SOFT [63] combines abstraction refinement for predicate abstraction with several other abstract domains to check standard runtime errors in C programs such as buffer overflows and null dereferences. Finally, the ARMC model checker [85] implements abstract refinement using a constraint-based logic programming language. ARMC can generate refinements for linear arithmetic constraints which allows it to handle programs with intensive operations on numerical data.

1.1.2 Deductive verification

An alternative approach for demonstrating that a program exhibits a specific temporal behaviour has been *deductive verification*, where one attempts to construct a formal object (i.e. a proof) by deducing statements from premises using a formal *proof system*.

Deductive verification has been linked to temporal logics from their early beginnings. Some of the earliest examples of proof systems for verification come from the proposers of CTL and LTL, Manna and Pnueli [74, 73]. The complexity of devising a proof system for verification of temporal logic is witnessed in this early attempt, as its capability was limited to demonstrate safety and liveness formulas of a very restricted form. These early papers perhaps influenced the direction of work of deductive verification, where the emphasis was highly inclined towards showing the systems' (relative) completeness whilst placing less importance on practical implementation.

Attempting to overcome the initial limitations, an extended version of the initial work was soon to come in [75, 76] where the set of properties that could be verified was extended to include *reactivity formulas*, stating that some program behaviours occurs as the consequence of previous actions. This extension provided full coverage for the entire linear temporal logic since it was demonstrated that any property specifiable by LTL can be expressed as a reactivity formula.

Having a proof system capable of proving linear temporal properties of programs, a natural extension was to tackle the problem of branching temporal logics. Some of the first approaches aimed to leverage existing linear temporal logic proof systems by restricting the shape of the temporal formulas that could be verified, disallowing nested path quantifiers [84, 66, 57]. This approach required a two-fold transformation: 1) replacing temporal formulas by assertions which contain no path quantifiers or temporal operators, and 2) replacing the resulting formula by a single Boolean variable, at the price of augmenting the original program with auxiliary variables. Hence, the problem of verifying an arbitrary branching temporal logic formula was reduced to verifying a linear temporal logic formula on a transformed program.

These early pieces raised some concerns regarding the practical use of their proof systems in particular, and that of deductive verification in general. Transformation of arbitrary formulas into a canonical form was far from trivial, and at least exponential [84]. Moreover, the resulting verification conditions required by the rules in the proof systems also involved complex verification tasks.

In light of these concerns, deductive verification techniques that operated directly on the temporal formula were studied. Samples of this technique are found in [15, 13, 93] where the proof structure itself is constructed from the original property formula. When compared to previous approaches, this technique yields perhaps less succinct proof systems, with a larger set of rules (including one for each temporal operator). On the other hand, the complexity of the verification conditions generated by the system is reduced as the proof progresses (reducing the complexity of the temporal formula), resulting in verification of simpler assertions.

Unfortunately, due to its generality, this technique presents very difficult challenges in constructing proofs in a fully automatic way, requiring human insight to complete the construction of the proof. This observation meant that the use of these proof systems was usually deemed unsuitable for practical purposes.

1.1.3 Current open problems in temporal verification

The importance of the temporal verification techniques and approaches so far mentioned is undeniable, but this does not mean there is no room for improvement. In particular, most of the techniques and tools related to model checking mentioned so far either ignore the effect of mutable data structures, focus on restricted problems regarding the heap (i.e. alias analysis and reachability analysis) or presume the safe execution of a given program with respect to heap operations. However, the effects of an unbounded heap represent one of the biggest challenges to scalable and precise software verification [65, 38].

One of the main complications of heap analysis arises from *aliasing* where two syntactically distinct expressions might refer to the same memory location, and hence updating the memory by writing to one of the locations requires updating information about the contents of a syntactically different location. One proposed solution to this problem is simply to translate such heap-aware programs into integer variables, in such a way that properties such as memory safety or termination of the original program follows from a corresponding property in its integer translation [72, 41, 38]. However, for more general temporal properties, this technique might produce unsound results. In general, it is not clear whether it is feasible to provide suitable translations from heap to integer programs for any temporal property we might wish to prove. Moreover, even when a suitable abstraction is found, important information about the shape of the heap data structures is typically lost, which might break the verification of temporal properties that rely on such shape information.

Example 1.1.1. *Consider the following nonterminating program that nondeterministically alternates between emptying the heap and appending an arbitrary number of elements to the head of a list structure:*

```
while(true){
    if(*) {
        while(x!=nil) {
          temp:=x.next; free(x); x:=temp;
        }
    } else {
        while(*) {
            y:=new(); y.next:=x; x:=y;
        } } }
```

Proving memory safety of this program (i.e. no null dereferences / deallocations) could be achieved by means of a simple numeric abstraction that tracks emptiness / nonemptiness of the list. Attempting to prove instead the more interesting property that it is always possible for the heap to become empty would require us to produce a different numeric abstraction, requiring the user to provide a notion *of size that represents the length of the list. Even when such an abstraction is provided, we cannot prove the stronger property that throughout the execution of the program the heap is always a nil-terminating acyclic list, as the number of elements in a list is not enough to show that the list is not acyclic.*

Whereas it *might* be possible to provide numeric abstractions to suit more complex temporal properties, it is not clear that this transformation is more beneficial than a direct treatment of the original program.

With regards to verification of heap-aware programs, deductive verification has presented different lines of work that advocated the use of non-standard specification languages that build on temporal logic concepts. In particular, aiming to tackle the problem of non-terminating heap manipulating programs resulted in solutions with varied levels of expressivity and automation. Navigational Temporal Logic (NTL) [49] introduces an extension of linear temporal logic that allows one to express the creation, adaptation and removal of heap structures and proposes a tableau-based model checking algorithm to verify these properties automatically, but with limited support for data structures. Evolution Temporal Logic [100] proposes a similar but more expressible language in which arbitrary predicates on heap locations are allowed. This approach is less automatic than NTL since the user must first examine the code to provide suitable ranking functions. [10, 18] introduces a formalism to reason about termination of heap manipulating programs.

Despite these efforts, these approaches present their own set of limitations, where practical application has so far been limited to non-automated solutions.

1.2 Our proposal

Following the previously stated limitations in the state-of-the-art verification of infinite heap-aware programs, we aim to devise a sound and fully automated temporal verification framework for such programs. In this pursuit, we set out a list of objectives for such a framework:

Generality - Driven by the nature of the problem at hand, we aim to step aside from properties of finite state program properties in favour of verifying the whole spectrum of safety, liveness and fairness properties of programs. In particular, we seek to produce a temporal logic framework applicable to both linear and branching views of time.

Soundness - Without fully sacrificing scalability and speed, the cornerstone of the framework is to be a sound system that prevents false negatives, where the validity of the product is rigorously checked.

Memory awareness - Witnessing a critical shortcoming of previous temporal logic verification frameworks based on model checking, we aim to tackle programs with full access to the heap. Observing the limitations of previous deductive verification approaches to tackle this problem we seek to avoid devising ad-hoc logics. We instead favour the use of the established and well-studied *separation logic* as a key element of our proposed solution.

Infinite-state - Addressing limitations of previous analyses designed for heap-aware programs, we intend to provide a verification framework capable of analysing infinite state programs.

Full automation - Finally, seeking to reduce the burden of deductive verification's practical applicability, our framework endeavours to achieve full automation.

In summary, the aim of this thesis is to provide a verification framework that combines the expressivity of temporal logic to describe properties about the evolution of program behaviours with the expressivity of separation logic to elegantly handle the manipulation of the heap. To this effect, we will formulate proof systems which manipulate temporal judgements about programs, and attempt to directly construct a proof that a program has a given temporal property by means of an automatic proof search. To handle the fact that the proof search can be done *ad infinitum*, we will employ the increasingly popular technique of *cyclic proof* [92, 17, 19, 22], in which proofs are finite cyclic graphs subject to a global soundness condition.

Having previously discussed the concepts regarding temporal logic and proof systems, the next two section are dedicated to introducing the concepts of the two other major ingredients of our framework: separation logic and cyclic proofs.

1.2.1 Separation logic

Separation logic is a formal system for specifying and reasoning about heap-aware programs. Like Hoare logic, it uses annotations that serve as pre- and post-conditions of commands. Unlike Hoare logic, it provides support for the principle of *locality*, where an assertion holds on a particular part of the heap, or *heaplet*. This is particularly useful when handling memory *aliases*, meaning that two memory pointers are pointing to the same memory address.

In a broad sense, separation logic is often understood as both an assertion language and a specification language that is applied to programs in an imperative language.

A common feature of the programming languages to which separation logic reasoning is applied is the use of commands for the manipulation of mutable shared data structures where memory management is explicit.

$$<C>::= \ldots \mid var := [exp] \mid [exp] := exp \mid var := \mathbf{alloc}() \mid \mathbf{free}(exp) \mid \ldots$$

Intuitively, the command $var := [exp]$ reads a value from memory and assigns it to a variable; this command would cause a memory fault if the memory location is not accessible. The command for mutation $[exp] := exp$ stores a value into a given memory address; this command would also cause a memory fault if the memory location is not accessible. The memory allocation command $\mathrm{var} := \mathbf{alloc}()$ allows us to allocate fresh memory to be handled by the program. Finally, the command for memory deallocation $\mathbf{free}(exp)$ allows us to give up memory locations when no longer needed; this command would also cause a memory fault when the memory location is not accessible.

The model of heaps on which these assertions are interpreted extends computational states, previously limited to the store $store : \mathsf{Var} \rightarrow \mathsf{Val}$, mapping variables to values, with a heap $heap : \mathsf{Loc} \rightharpoonup_{\mathrm{fin}} \mathsf{Val}$, mapping finitely many memory locations to values.

As in Hoare logic, assertions of separation logic describe program states. Since the model of program states has been extended with a heap, separation logic extends the usual operators of propositional logic with three new forms of assertions that describe the heap. These are

- the *empty* heap assertion **emp** indicating that the heap is empty;
- the *points-to* assertion $e \mapsto e'$ indicating that the heap consists of a single memory cell with address e and contents e';
- the *separating conjunction* assertion $P * Q$ indicating that the memory can be split into two disjoint parts such that P holds for one part and Q holds for the other; and
- the *separating implication* $P \mathrel{-\!\!*} Q$, asserting that whenever a heap satisfies the property P, its composition with the current heap satisfies the property Q.

The use of the separating implication is particularly useful when a piece of code mutates memory locally, and we want to state some property of the entire heap. Nevertheless it has been shown that the inclusion of the separating implication makes separation logic undecidable and hard to reason with [28], with most systems and tools excluding it from the assertion language. In this work, we will follow this trend and exclude the separating implication from the language we will use later on to describe program states.

While assertions describe program states, *specifications* in separation logic describe the behaviour of programs. Specifications in separation logic are *Hoare triples*, of the form

$$\{<assertion>\}\ \texttt{<command>}\ \{<assertion>\}$$

The initial assertion is called the *precondition* and the second assertion is called the *postcondition*. The partial correctness specification $\{P\}$ `C` $\{Q\}$ is true if and only if, starting in any state that satisfies the assertion P no execution of `C` leads to a memory fault, and when the execution of `C` terminates in a final state, then this final state satisfies the assertion Q.

This form of specification is so similar to standard Hoare logic that it is important to note the differences. Unlike Hoare logic, separation logic provides support for the principle of *locality*, where an assertion holds on a particular part of the program's heap. This is particularly useful when handling memory *aliases* (i.e. two memory locations pointing to the same location).

To illustrate this, consider the following simple program:

```
[x] := 4;
[y] := 2;
```

It would be very tempting to say that at the end of the execution of this program (i.e. the postcondition) we have a program state where variable x contains the value 4 (i.e. $x \mapsto 4$) while variable y contains the value 2 (i.e. $y \mapsto 2$). Nevertheless, this is not the case when x and y are aliases for the same memory location (i.e. $x = y$). If this was the case, the resulting state after executing this program would witness $x \mapsto 2 \wedge y \mapsto 2$. To avoid this situation, we would have to explicitly state the relation held between program variables $x \neq y$. It is evident from this simple program that this solution scales poorly.

In separation logic, however, this kind of difficulty can be avoided by using the separating conjunction that states that the two components hold for disjoint portions of the addressable storage. A more general advantage of this operation is the support that separation logic gives to *local reasoning*, which underlies the scalability of the approach. For example, given the specification

$$\{x \mapsto 4\}\ \texttt{[x]:=2}\ \{x \mapsto 2\}$$

it is implied that not only that the program expects to find the value 4 assigned to variable x, but also that this memory location is the only memory location accessed by the execution of the program (commonly called its *footprint*). If `[x]:=2` is part of a larger codebase that manipulates some separate memory addresses, one can infer directly that the additional storage is not modified by `[x]:=2`.

In a realistic situation, the program under analysis can be a much more substantial program and the separate memory storage can be much larger. Nevertheless, one can still reason *locally* about it, while ignoring the separate memory locations.

This locality property is what gives rise to the *frame rule*, to which most of the modularity success of separation logic is due:

$$\frac{\{P\}\ \texttt{C}\ \{Q\}}{\{P * R\}\ \texttt{C}\ \{Q * R\}}\ \text{(Frame)}$$

The triple $\{P\}$ C $\{Q\}$ implicitly states that the execution of C depends only on the part of the heap described by the assertion P. Any other part of the heap R remains unchanged by the execution of C.

The frame rule allows us to extend a local specification, involving only the variables and heap cells that may actually be used by C, by adding arbitrary assertions about variables and heap cells that are not modified or mutated by C.

In any valid specification $\{P\}$ C $\{Q\}$, the precondition states that the heap contains every cell in the footprint of C, except for those that are locally allocated; *locality* is the converse implication that every cell described in the precondition belongs to the footprint. The role of the frame rule is therefore to infer from a local specification of a command the more global specification appropriate to the possibly larger footprint of an enclosing command.

Beyond the rules pertaining to Hoare logic and the frame rule, separation logic presents an axiom for each of the new heap-manipulating commands.

$$\frac{}{\{e \mapsto e'\}\ \texttt{x:=[e]}\ \{x = e' \wedge e \mapsto e'\}}\ \text{(Read)}$$

$$\frac{}{\{e \mapsto -\}\ \texttt{[e]:=e'}\ \{e \mapsto e'\}}\ \text{(Write)}$$

$$\frac{}{\{\textbf{emp}\}\ \texttt{x:=alloc()}\ \{x \mapsto -\}}\ \text{(Alloc)}$$

$$\frac{}{\{e \mapsto -\}\ \texttt{free(e)}\ \{\textbf{emp}\}}\ \text{(Free)}$$

The extensive study of separation logic (and similar approaches that rely on program contracts) provided a significant advance in the automation of the verification process of programs that access the heap. Smallfoot [9] was the first implementation to use separation logic; its goal was to investigate the extent to which proofs and specifications made by hand could be treated automatically. The automation in Smallfoot is related to the assertion checking, but the user has to provide preconditions, postconditions and loop invariants. A major step was to show that the method is indeed scalable in practice [101]. This led the way to multiple academic tools using separation logic as their assertion language, such as SpaceInvader [50], Thor [71], Xisa [30], SmallFoot [9], JStar [51] and CYCLIST [22], that are able to demonstrate specific properties of programs such as *memory safety*, termination and the shape of data structures used by the program.

Much of the success of these tools is thanks to the mechanisation of verification-related questions regarding separation logic, such as satisfiability and entailment of separation formulae in various tools, including Asterix[81], CYCLIST[22], SLSAT[21] and SLEEK[31]. A notable fact about these tools is the diversity of techniques used by these solvers, from reduction to SAT and SMT problems, resolution-based, to reduction to tree automata membership.

In a sense, the question about the practical utility of separation logic was quickly answered, leading to a new generation of industrial strength tools in the last 5 years. Examples of these include Facebook INFER [27], a tool that uses separation logic to analyse mobile applications and report problems caused by null pointer access and resource memory leaks, and Microsoft Slayer [11], a tool designed to prove memory safety of industrial system code reporting dangling pointer dereferences, double frees and memory leaks.

Leveraging on the strengths of separation logic in the analysis of heap-aware programs, primarily its expressivity, scalability and popularity, we seek to include it in our proposed temporal logic framework to replace previous predicate-logic based assertions with separation logic assertions as the pure part of our logic.

1.2.2 Cyclic proofs

Cyclic proofs have been recently proposed as an alternative to traditional proof by explicit induction for fixed point logics. In contrast to standard proofs, which are simply derivation trees, a cyclic proof is a derivation tree with back-links, subject to a global soundness condition ensuring that the proof can be read as a proof by *infinite descent* à la Fermat [16]. This allows explicit induction rules to be dropped in favour of simple unfolding rules.

Broadly speaking, the soundness condition states that every infinite path in the derivation must have a syntactic *trace* following the path, which progresses infinitely often; informally, a trace can be thought of as a well-founded measure while its progress corresponds to strict decreases of this measure.

Cyclic proof systems seem to have first been used in computer science as tableaux for the propositional μ-calculus [88]. Since then, cyclic proof systems have been proposed for a number of applications, including theorem provers that span from interactive theorem provers for specific logical systems to fully automated generic theorem provers.

Regarding interactive theorem provers, the QUODLIBET tool [4], based on first-order logic with inductive datatypes, uses a version of infinite descent to prove inductive theorems whereby a proof node is annotated with a *weight*, which must strictly decrease at back-link sites. This weight effectively serves the purpose of the syntactic trace condition following infinite paths in cyclic proofs.

Advances on the automation of cyclic proofs led to the work presented in [20], where Brotherston et al. describe an automated cyclic prover for entailments of separation logic implemented in HOL Light. Generalising the previous work, in [22], Brotherston et al. present CYCLIST: a generic theory of cyclic proof and an unrestricted implementation in a fully automated theorem prover. Along with the framework, multiple applications to concrete logical systems, including automated proof search procedures, are also introduced. In its various instantiations, the prover is capable of automatically proving theorems with a complex inductive structure. Although CYCLIST does not claim to be an industrial-strength theorem prover, the results of the experiments carried out in CYCLIST are encouraging, presenting the potential for developing new instantiations of automated theorem provers to other fixed-point logics in this framework.

The study of cyclic proofs in the last decade has lead to the implementation of program verification tools that employ cyclic proofs in some form or another.

In [18], Brotherston et al. propose a novel approach to proving the termination of pointer programs, which combines separation logic with cyclic proofs within a Hoare-style proof system. The logical preconditions in this system employ inductively defined predicates to describe heap properties, and proofs are cyclic derivations in which some inductive predicate is unfolded infinitely often along every infinite path, thus allowing to discard such infinite paths in the proof by an infinite descent argument.

Extending the previous work, Rowe and Brotherston describe a formal verification framework and implementation, based upon cyclic proofs, for certifying the safe termination of imperative pointer programs with recursive procedures in [87].

Despite the recent rise in cyclic proof works, their application for the automated verification of temporal properties of heap manipulating programs has not, to the best of our knowledge, been studied before. It is our aim to address this gap.

1.3 Synopsis

The remainder of this thesis is structured as follows.

Chapter 2 We introduce the syntax and semantics of our programming language used to implement the programs we aim to verify. This language includes constructs to directly manipulate the heap by allowing memory allocation/deallocation and reading/writing to memory locations. We then introduce our language of assertions for memory states based on separation logic. Finally, we introduce our CTL and LTL languages for expressing temporal properties of programs.

Chapter 3 We formulate a cyclic proof system for verifying CTL properties of our programs. We show that the proof rules of our system are sound with respect to the operational semantics of our programming language and demonstrate the global soundness of our cyclic proof system.

Chapter 4 We define a second proof system to handle LTL temporal assertions. We review the concept of *prophecy variables* (cf. [40]) to determinise the execution of nondeterministic programs in order to induce a linearisation of time.

Equipped with this formalism, we present the proof rules of our LTL proof system, emphasising the introduction of new rules to handle nondeterminism.

After defining the system and formulating the global soundness condition we prove that our LTL cyclic proof system is sound.

Chapter 5 An important component in the verification of reactive systems is a set of *fairness constraints* to guarantee that no computation is neglected forever. We describe how our CTL and LTL cyclic proof systems can be modified to treat (strong) fairness constraints.

Chapter 6 We discuss the implementation details of the cyclic proof systems presented in this thesis. Our proof systems are implemented on top of the CYCLIST theorem prover [22]. Broadly speaking, our implementation performs iterative depth-first search, aimed at closing open nodes in the proof by either applying an axiom or forming a back-link. If an open node cannot be closed, we instead attempt to apply symbolic execution inference rule; if this is not possible, we try unfolding temporal operators and inductive predicates in the precondition to enable symbolic execution to proceed. Finally, after all open nodes in the proof have been closed, a global soundness check of the cyclic proof is performed automatically.

Chapter 7 Following the implementation of our cyclic proof systems, we evaluate these automated tools on handcrafted nondeterministic and nonterminating programs. Our test suite is an adaptation of the common model checking benchmarks presented in [40, 41] for the verification of temporal properties of nondeterministic programs, where operations/iterations on integer variables in the original benchmarks are replaced in favour of operations/iterations on heap data structures.

Chapter 8 We present our conclusions, summarising our contributions and proposing lines for future work.

Chapter 2

Background

In this chapter we lay out the technical background that serves as the foundation of our proof systems. In Section 2.1 we fix the syntax and semantics of the programming language used to implement the programs we aim to verify. We then introduce the formalism of *symbolic heaps* extended with *inductive predicates* as our language of assertions about *memory states* in Section 2.2. We bring this chapter to an end by introducing two languages for expressing *temporal properties* of programs in Section 2.3. The first language is based on Computation Tree Logic (CTL) and the second based on Linear Temporal Logic (LTL).

2.1 Programming language

In this section, we introduce a simple language of **while** programs which includes constructs to directly manipulate the heap by allowing memory allocation/deallocation and reading/writing to memory locations.

We assume a countably infinite set Var of *variables* and a first-order language Σ_{exp} of *expressions* over Var, satisfying $\Sigma_{exp} \supseteq \mathsf{Var}$ and containing a distiguished constant symbol nil.

Definition 2.1.1 (Programming language)**.** The syntax of *expressions E branching conditions B* and *commands C* of our programming language is given by the following grammar:

$$
\begin{array}{rcl}
E & ::= & x \mid f(E_1,\dots,E_n) \\
B & ::= & E = E \mid E \neq E \mid * \\
C & ::= & \mathrm{x} := [E] \mid [E] := E \mid \mathrm{x} := \mathbf{alloc}() \mid \mathbf{free}(E) \mid \mathrm{x} := E \mid \\
 & & \mathbf{skip} \mid \mathbf{if}\, B\, \mathbf{then}\, C\, \mathbf{else}\, C\, \mathbf{fi} \mid \mathbf{while}\, B\, \mathbf{do}\, C\, \mathbf{od} \mid C;C \mid \varepsilon
\end{array}
$$

where x ranges over variables and f over function symbols. Note that we write ε for the empty command, $*$ for a nondeterministic branching condition, and $[E]$ for dereferencing of expression E.

We define the semantics of the programming language in a standard *stack-and-heap model* employing heaps of records. We fix an infinite set Val of *values*, and a set $\mathsf{Loc} \subset \mathsf{Val}$ of addressable memory *locations*, and assume a distiguished null value $nil \in \mathsf{Val} - \mathsf{Loc}$. A *stack* is simply a map $s : \mathsf{Var} \to \mathsf{Val}$ from variables to values. The semantics $[\![E]\!]s$ of expression E under stack s is then given in the standard way; in particular, $[\![\mathsf{nil}]\!]s = nil$ and $[\![x]\!]s = s(x)$ for $x \in \mathsf{Var}$. Assuming some fixed interpretation for any function symbols in the expression language, s can then be extended to all expressions in the usual way $[\![f(E_1,\dots,E_n)]\!]s = f([\![E_1]\!]s,\dots,[\![E_n]\!]s)$. We extend stacks pointwise to act on tuples of terms.

A *heap* is a partial, finite-domain function $h : \mathsf{Loc} \rightharpoonup_{\text{fin}} (\mathsf{Val}\ \mathsf{List})$, mapping finitely many memory locations to *records*, i.e. arbitrary-but-finite-length tuples of values; we write $\mathrm{dom}(h)$ for the set of locations on which h is defined. We write emp for the empty heap, undefined on all locations, and $\uplus$ to denote composition of *domain-disjoint* heaps: $h_1 \uplus h_2$ is the union of h_1 and h_2 when $\mathrm{dom}(h_1) \cap \mathrm{dom}(h_2) = \emptyset$ (and undefined otherwise). If f is a stack or a heap then we write $f[x \mapsto v]$ for the "updated" environment defined by:

$$
f[x \mapsto v](y) = \begin{cases} v & \text{if } y = x \\ f(y) & \text{otherwise} \end{cases}
$$

A stack paired with a heap, (s,h), is called a *(memory) state*.

Lemma 2.1.2 (Expression substitution)**.** *For all expressions E, stacks s and vari-*

ables x, x', $[\![E[x'/x]]\!]s = [\![E]\!]s[x \mapsto [\![x']\!]s]$

Proof. By structural induction on E.

Base case $E = x$, where $x \in \mathsf{Var}$.

$$\begin{array}{rcll} [\![x[x'/x]]\!]s & = & [\![x]\!]s[x \mapsto [\![x']\!]s] & \\ [\![x']\!]s & = & [\![x]\!]s[x \mapsto [\![x']\!]s] & \text{by rewriting} \\ [\![x']\!]s & = & [\![x']\!]s & \text{by construction} \end{array}$$

Base case $E = y$, where $y \in \mathsf{Var}$ and $y \neq x$.

$$\begin{array}{rcll} [\![y[x'/x]]\!]s & = & [\![y]\!]s[x \mapsto [\![x']\!]s] & \\ [\![y]\!]s & = & [\![y]\!]s[x \mapsto [\![x']\!]s] & \text{by rewriting} \\ [\![y]\!]s & = & [\![y]\!]s & \text{by construction} \end{array}$$

Inductive case $E = f(E_1, \ldots, E_n)$ for any function f.

$$\begin{array}{cll} & [\![f(E_1, \ldots, E_n)[x'/x]]\!]s & \\ = & f([\![E_1[x'/x]]\!]s, \ldots, [\![E_n[x'/x]]\!]s) & \text{interpretation of f} \\ = & f([\![(E_1]\!]s[x \mapsto [\![x']\!]s], \ldots, [\![E_n]\!]s[x \mapsto [\![x']\!]s]) & \text{induction hypothesis} \\ = & [\![f(E_1, \ldots, E_n)]\!]s[x \mapsto [\![x']\!]s] & \text{interpretation of f} \end{array}$$

□

A *(program) configuration* γ is a triple $\langle C, s, h \rangle$ where C is a command as per Definition 2.1.1, s a stack and h a heap. If γ is a configuration, we write γ_C, γ_s, and γ_h respectively for its first, second and third components. A configuration γ is called *final* if $\gamma_C = \varepsilon$. The small-step operational semantics of programs is given by a binary relation $\rightsquigarrow$ on program configurations, where $\gamma \rightsquigarrow \gamma'$ holds if the execution of the command γ_C in the state (γ_s, γ_h) can result in a new program configuration γ'. We write $\rightsquigarrow^*$ for the reflexive-transitive closure of $\rightsquigarrow$. The special configuration fault is used to denote a memory fault, e.g., if a command tries to access non-allocated memory. The operational semantics of our programming language are shown in Figure 2.1.

$$\frac{}{\langle \mathbf{skip}; C,s,h\rangle \rightsquigarrow \langle C,s,h\rangle} \qquad \frac{}{\langle \varepsilon; C,s,h\rangle \rightsquigarrow \langle C,s,h\rangle}$$

$$\frac{}{\langle x := E; C,s,h\rangle \rightsquigarrow \langle C,s[x \mapsto [\![E]\!]s],h\rangle} \qquad \frac{[\![E]\!]s \in \mathsf{dom}(h)}{\langle x := [E]; C,s,h\rangle \rightsquigarrow \langle C,s[x \mapsto h([\![E]\!]s)],h\rangle}$$

$$\frac{[\![E]\!]s \in \mathsf{dom}(h)}{\langle [E] := E'; C,s,h\rangle \rightsquigarrow \langle C,s,h[[\![E]\!]s \mapsto [\![E']\!]s]\rangle} \qquad \frac{\ell \in \mathsf{Loc} \setminus \mathsf{dom}(h) \quad v \in \mathrm{Val}}{\langle x := alloc(); C,s,h\rangle \rightsquigarrow \langle C,s[x \mapsto \ell],h[\ell \mapsto v]\rangle}$$

$$\frac{[\![E]\!]s \in \mathsf{dom}(h)}{\langle free(E); C,s,h\rangle \rightsquigarrow \langle C,s,(h \mid \mathsf{dom}(h) \setminus \{[\![E]\!]s\})\rangle} \qquad \frac{[\![B]\!]s}{\langle \mathbf{if}\, B\, \mathbf{then}\, C\, \mathbf{else}\, C'\, \mathbf{fi}; C'',s,h\rangle \rightsquigarrow \langle C; C'',s,h\rangle}$$

$$\frac{\neg[\![B]\!]s}{\langle \mathbf{if}\, B\, \mathbf{then}\, C\, \mathbf{else}\, C'\, \mathbf{fi}; C'',s,h\rangle \rightsquigarrow \langle C'; C'',s,h\rangle} \qquad \frac{\neg[\![B]\!]s}{\langle \mathbf{while}\, B\, \mathbf{do}\, C\, \mathbf{od}; C',s,h\rangle \rightsquigarrow \langle C',s,h\rangle}$$

$$\frac{[\![B]\!]s}{\langle \mathbf{while}\, B\, \mathbf{do}\, C\, \mathbf{od}; C',s,h\rangle \rightsquigarrow \langle C; \mathbf{while}\, B\, \mathbf{do}\, C\, \mathbf{od}; C',s,h\rangle} \qquad \frac{[\![E]\!]s \notin \mathsf{dom}(h)}{\langle x := [E]; C,s,h\rangle \rightsquigarrow \mathsf{fault}}$$

$$\frac{[\![E]\!]s \notin \mathsf{dom}(h)}{\langle [E] := E'; C,s,h\rangle \rightsquigarrow \mathsf{fault}} \qquad \frac{[\![E]\!]s \notin \mathsf{dom}(h)}{\langle free(E); C,s,h\rangle \rightsquigarrow \mathsf{fault}}$$

Figure 2.1: Small-step operational semantics of programs, given by the binary relation $\rightsquigarrow$ over program configurations.

An *execution path* is a (maximal finite or infinite) sequence $(\gamma_i)_{i\geq 0}$ of configurations such that $\gamma_i \rightsquigarrow \gamma_{i+1}$ for all $i \geq 0$. If $\pi = \gamma_0 \rightsquigarrow \gamma_1 \rightsquigarrow \gamma_2 \rightsquigarrow \ldots$ is a path, then we write π_i to denote the i^{th} suffix of π (e.g. $\pi_1 = \gamma_1 \rightsquigarrow \gamma_2 \rightsquigarrow \ldots$). We also write $\pi[i]$ to denote the i^{th} configuration of π (e.g. $\pi[1] = \gamma_1$). A path π *starts from* configuration γ if $\pi[0] = \gamma$.

Remark 2.1.3. In temporal program verification, it is relatively common to consider all program execution paths to be infinite, and all temporal properties to quantify over infinite paths. This can be achieved either (i) by modifying programs to contain an infinite loop at every exit point, or (ii) by modifying the operational semantics so that final configurations loop infinitely (i.e. $\langle \varepsilon,s,h\rangle \rightsquigarrow \langle \varepsilon,s,h\rangle$).

We take a slightly different, but equivalent approach, by quantifying over paths that are either infinite or else maximally finite. This has the same effect as directly modifying programs or their operational semantics.

2.2 Memory state assertions.

In this section we introduce the formalism of *symbolic-heap* separation logic [8] extended with user-defined (inductive) predicates [17], typically needed to express complex shape properties of the memory, that will serve as our assertion language to express properties of memory states.

We assume a fixed first-order *logic language* Σ_{log} that extends the expression language of our programming language (i.e. $\Sigma_{log} \supseteq \Sigma_{exp}$). The *terms* of Σ_{log} are defined as usual, with variables drawn from Var. We write $t(x_1,\ldots,x_k)$ for a term t all whose variables occur in $\{x_1,\ldots,x_k\}$ and we use vector notation to abbreviate sequences, e.g. $\mathbf{x}$ for $(x_1,\ldots,x_k)$. The interpretation $[\![t]\!]s$ of a term t of Σ_{log} in a stack s is then defined in the same way as expressions, provided we are given an interpretation for any constant or function symbol that is not in Σ_{exp}.

We designate finitely many of the predicate symbols of our logic language as *inductive symbols*. For each predicate symbol Q of arity k we assign an interpretation $[\![Q]\!] \in \mathsf{Pow}(\mathsf{Heaps} \times \mathsf{Val\ List}^{\mathsf{k}})$

Definition 2.2.1. A *symbolic heap* is given by a disjunction of assertions each of the form $\Pi : \Sigma$, where Π is a finite set of *pure formulas* ϖ given by the following grammar:

$$\varpi \quad ::= \quad E = E \mid E \neq E$$

and Σ is a *spatial formula* given by the following grammar:

$$\Sigma \quad ::= \quad \top \mid \bot \mid \mathsf{emp} \mid E \mapsto \mathbf{E} \mid \Sigma * \Sigma \mid \Psi(\mathbf{E})$$

where E ranges over expressions, $\mathbf{E}$ over tuples of expressions and Ψ over predicate symbols.

Definition 2.2.2. Given a state (s,h) and symbolic heap $\Pi : \Sigma$, we write $s,h \models \Pi : \Sigma$ if $s,h \models \varpi$ for all pure formulas $\varpi \in \Pi$, and $s,h \models \Sigma$, where the relation $s,h \models A$ between states and formulas is defined by

$$
\begin{array}{rcl}
s,h \models \top & \Leftrightarrow & \text{always} \\
s,h \models \bot & \Leftrightarrow & \text{never} \\
s,h \models E_1 = E_2 & \Leftrightarrow & [\![E_1]\!]s = [\![E_2]\!]s \\
s,h \models E_1 \neq E_2 & \Leftrightarrow & [\![E_1]\!]s \neq [\![E_2]\!]s
\end{array}
$$

$$
\begin{array}{rcl}
s,h \models \mathsf{emp} & \Leftrightarrow & \mathrm{dom}(h) = \emptyset \\
s,h \models E \mapsto \mathbf{E} & \Leftrightarrow & \mathrm{dom}(h) = \{[\![E]\!]s\} \mathrm{and} h([\![E]\!]s) = [\![\mathbf{E}]\!]s \\
s,h \models \Psi(\mathbf{E}) & \Leftrightarrow & ([\![\mathbf{E}]\!]s,h) \in [\![\Psi]\!] \\
s,h \models \Sigma_1 * \Sigma_2 & \Leftrightarrow & h = h_1 \uplus h_2 \text{ and } s,h_1 \models \Sigma_1 \text{ and } s,h_2 \models \Sigma_2 \\
s,h \models \Omega_1 \vee \Omega_2 & \Leftrightarrow & s,h \models \Omega_1 \text{ or } s,h \models \Omega_2
\end{array}
$$

Symbolic heaps denote memory states via the satisfaction relation shown before. However, we insist that the interpretation $[\![\Psi]\!]$ of each inductive predicate symbol Ψ is fixed by a given inductive definition for Ψ. Our inductive definition schema follows closely the one formulated in [17] and is given by the following definition:

Definition 2.2.3 (Inductive definition)**.** An *inductive definition* of an inductive predicate symbol Ψ is a finite set of inductive rules, each of the form $\Pi{:}\Sigma \Rightarrow \Psi(\mathbf{E})$ where $\Pi{:}\Sigma$ is a symbolic heap formula and $\Psi(\mathbf{E})$ is a predicate formula.

The standard interpretation of an inductive predicate symbol Ψ is then the least prefixed point of a monotone operator constructed from the inductive definitions.

Definition 2.2.4 (Definition set operator)**.** Let the inductive predicate symbols of Σ_{log} be $\Psi_1,\ldots,\Psi_n$ with arities $a_1,\ldots,a_n$ respectively, and suppose we have a unique inductive definition for each predicate symbol Ψ_i. Then for each $i \in \{1,\ldots,n\}$, from the inductive definition for Ψ_i, say $\Pi_1 : \Sigma_1 \Rightarrow \Psi_i(\mathbf{E}_1),\ldots,\Pi_k : \Sigma_k \Rightarrow \Psi_i(\mathbf{E}_k)$ we obtain a corresponding n-ary function $\chi_i : (\mathsf{Pow}(\mathsf{Heaps} \times \mathsf{Val\ List}^{\mathsf{a}_1}) \times \ldots \times \mathsf{Pow}(\mathsf{Heaps} \times \mathsf{Val\ List}^{\mathsf{a}_n})) \to \mathsf{Pow}(\mathsf{Heaps} \times \mathsf{Val\ List}^{\mathsf{a}_i})$ as follows:

$$\chi_i(\mathbf{X}) = \bigcup_{1 \le j \le k} \{(h, [\![\mathbf{E}_j]\!](s[\mathbf{E}_j \mapsto \mathbf{d}])) \mid (s[\mathbf{E}_j \mapsto \mathbf{d}, h]) \models_{[\![\Psi]\!] \mapsto \mathbf{X}} \Pi_j : \Sigma_j\}$$

where s is an arbitrary stack and $\vDash_{[\![\Psi]\!] \mapsto \mathbf{X}}$ is the satisfaction relation defined exactly as Definition 2.2.2 except that $[\![\Psi_i]\!] = \pi_i^n(\mathbf{X})$ for each $i \in \{1,\dots,n\}$ (where π_i. is the ith projection function on n-tuples of sets defined by $\pi_i^n(X_1,\dots,X_n) = X_i$). Then the *definition set operator* for $\Psi_i,\dots,\Psi_n$ is the operator χ_Ψ defined by:

$$\chi_\Psi(\mathbf{X}) = (\chi_1(\mathbf{X}),\dots,\chi_n(\mathbf{X}))$$

Example 2.2.5. *For example, consider the following inductive definition for a binary inductive predicate symbol* ls *that denoted singly-linked list segments*

$$\begin{array}{rcl} \mathsf{emp} & \Rightarrow & ls(x,x) \\ x \mapsto x' * ls(x',y) & \Rightarrow & ls(x,y) \end{array}$$

Then $[\![ls]\!]$ is the least prefixed point of the following operator, with domain and codomain $\mathsf{Pow}(\mathsf{Heaps} \times \mathsf{Val\ List}^2)$:

$$\begin{array}{rcl} \chi_{ls}(X) & = & \{(\mathsf{emp},(v,v)) \mid v \in \mathsf{Val}\} \\ & \cup & \{(h_1 \uplus h_2,(v,v')) \mid \exists w \in \mathsf{Val}.\mathsf{dom}(h_1) = v \text{ and } h(v) = w \\ & & \qquad \text{and } (h_2,(w,v')) \in X\} \end{array}$$

Note that the operator generated from a set of inductive definitions by Definition 2.2.4 is monotone [17] and consequently has a least prefixed point, which gives the standard interpretation for the inductively defined predicates of the language. Moreover, this least prefixed point can be iteratively approached by *approximants*.

Definition 2.2.6 (Approximants)**.** Let χ_Ψ be the definition set operator for the inductive predicates $\Psi_1,\dots,\Psi_n$ as in Definition 2.2.4. Define a chain of ordinal-indexed sets $(\chi_\Psi^\alpha)_{\alpha \geq 0}$ by transfinite induction : $\chi_\Psi^\alpha = \bigcup_{\beta<\alpha} \chi_\Psi(\chi_\Psi^\beta)$. Then, for each $i \in 1,\dots,n$, the set $\Psi_i^\alpha = \pi_i^n(\chi_\Psi^\alpha)$ is called the *α-th approximant of* Ψ_i.

Lemma 2.2.7 (Substitution)**.** $\forall P : \mathit{SymbolicHeap}, x : \mathit{Var}, x' : \mathit{Var}, s : \mathsf{Stack}, h : \mathsf{Heap}.$ $(s,h) \vDash P[x'/x]$ *if and only if* $(s[x \mapsto [\![x']\!]s],h) \vDash P$.

Proof. By structural induction on P.

Base cases $P = \top$ and $P = \bot$ and $P = \mathsf{emp}$ are trivial since variable substitution has no effect on the satisfaction of these formulas.

Base case $P = (E_1 = E_2)$:

$$\begin{array}{lll} & (s,h) \models (E_1 = E_2)[x'/x] & \\ \Leftrightarrow & [\![E_1[x'/x]]\!]s = [\![E_2[x'/x]]\!]s & \text{Definition 2.2.2} \\ \Leftrightarrow & [\![E_1]\!]s[x \mapsto [\![x']\!]s] = [\![E_2]\!]s[x \mapsto [\![x']\!]s] & \text{Lemma 2.1.2} \\ \Leftrightarrow & (s[x \mapsto [\![x']\!]s], h) \models (E_1 = E_2) & \text{Definition 2.2.2} \end{array}$$

Base case $P = (E_1 \neq E_2)$.

$$\begin{array}{lll} & (s,h) \models (E_1 \neq E_2)[x'/x] & \\ \Leftrightarrow & [\![E_1[x'/x]]\!]s \neq [\![E_2[x'/x]]\!]s & \text{Definition 2.2.2} \\ \Leftrightarrow & [\![E_1]\!]s[x \mapsto [\![x']\!]s] \neq [\![E_2]\!]s[x \mapsto [\![x']\!]s] & \text{Lemma 2.1.2} \\ \Leftrightarrow & (s[x \mapsto [\![x']\!]s], h) \models (E_1 \neq E_2) & \text{Definition 2.2.2} \end{array}$$

Base case $P = (E \mapsto \mathbf{E})$.

$$\begin{array}{lll} & (s,h) \models (E \mapsto \mathbf{E})[x'/x] & \\ \Leftrightarrow & \mathrm{dom}(h) = \{[\![E[x'/x]]\!]s\} \text{ and } h([\![E[x'/x]]\!]s) = [\![\mathbf{E}[\mathbf{x}'/\mathbf{x}]]\!]s & \text{Definition 2.2.2} \\ \Leftrightarrow & \mathrm{dom}(h) = \{[\![E]\!]s[x \mapsto [\![x']\!]s]\} \text{ and} & \\ & h([\![E]\!]s[x \mapsto [\![x']\!]s]) = [\![\mathbf{E}]\!]s[x \mapsto [\![x']\!]s] & \text{Lemma 2.1.2} \\ \Leftrightarrow & (s[x \mapsto [\![x']\!]s], h) \models (E \mapsto \mathbf{E}) & \text{Definition 2.2.2} \end{array}$$

Inductive case $P = \Psi(\mathbf{E})$.

$$\begin{array}{lll} & (s,h) \models \Psi(\mathbf{E})[x'/x] & \\ \Leftrightarrow & ([\![\mathbf{E}[x'/x]]\!]s, h) \in [\![\Psi[x'/x]]\!] & \text{Definition 2.2.2:} \\ \Leftrightarrow & ([\![\mathbf{E}[x'/x]]\!]s, h) \in \bigcup_{j=1}^{k} \{(h, [\![\mathbf{E}[x'/x]_j]\!](s[\mathbf{x}_j \mapsto \mathbf{d}])) \mid & \\ & (s[\mathbf{x}_j \mapsto \mathbf{d}, h]) \models_{[\![\Psi]\!] \mapsto \mathrm{x}} (\Pi_j : \Sigma_j)[x'/x]\} & \text{Definition 2.2.4} \\ \Leftrightarrow & ([\![\mathbf{E}]\!]s[x \mapsto [\![x']\!]s], h) \in \bigcup_{j=1}^{k} \{(h, [\![\mathbf{E}_j]\!](s[x \mapsto [\![x']\!]s][\mathbf{x}_j \mapsto \mathbf{d}])) \mid & \\ & (s[x \mapsto [\![x']\!]s][\mathbf{x}_j \mapsto \mathbf{d}, h]) \models_{[\![\Psi]\!] \mapsto \mathrm{x}} \Pi_j : \Sigma_j\} & \text{Induction hypothesis} \\ \Leftrightarrow & ([\![\mathbf{E}]\!]s[x \mapsto [\![x']\!]s], h) \in [\![\Psi]\!] & \text{Definition 2.2.4} \\ \Leftrightarrow & (s[x \mapsto [\![x']\!]s], h) \models \Psi(\mathbf{E}) & \text{Definition 2.2.2} \end{array}$$

Inductive case $P = (\Sigma_1 * \Sigma_2)$.

$$\begin{array}{lll}
 & (s,h) \vDash (\Sigma_1 * \Sigma_2)[x'/x] & \\
\Leftrightarrow & h = h_1 \uplus h_2 \text{ and } (s,h_1) \vDash \Sigma_1[x'/x] \text{ and} & \\
 & (s,h_2) \vDash \Sigma_2[x'/x] & \text{Definition 2.2.2} \\
\Leftrightarrow & h = h_1 \uplus h_2 \text{ and } (s[x \mapsto [\![x']\!]s],h_1) \vDash \Sigma_1 \text{ and} & \\
 & (s[x \mapsto [\![x']\!]s],h_2) \vDash \Sigma_2 & \text{Induction hypothesis} \\
\Leftrightarrow & (s[x \mapsto [\![x']\!]s],h) \vDash \Sigma_1 * \Sigma_2 & \text{Definition 2.2.2}
\end{array}$$

Inductive case $P = (\Omega_1 \vee \Omega_2)$.

$$\begin{array}{lll}
 & (s,h) \vDash (\Omega_1 \vee \Omega_2)[x'/x] & \\
\Leftrightarrow & (s,h) \vDash \Omega_1[x'/x] \text{ or } (s,h) \vDash \Omega_2[x'/x] & \text{Definition 2.2.2} \\
\Leftrightarrow & (s[x \mapsto [\![x']\!]s],h) \vDash \Omega_1 \text{ or } (s[x \mapsto [\![x']\!]s],h) \vDash \Omega_2 & \text{Induction hypothesis} \\
\Leftrightarrow & (s[x \mapsto [\![x']\!]s],h) \vDash (\Omega_1 \vee \Omega_2) & \text{Definition 2.2.2}
\end{array}$$

□

2.3 Temporal assertions.

We describe temporal properties of our programs using *temporal assertions*, built from the memory state assertions given above using standard operators of temporal logic [83]. We examine two concrete assertion languages, the first based on *computation tree logic* (CTL) [34], whose temporal operators quantify over possible execution paths, and the second on *linear time logic* (LTL) [83], whose temporal operators quantify over events along a given execution path.

2.3.1 CTL assertions

Computational Tree Logic is a branching temporal logic in the sense that a computation starting from a state is viewed as a tree, where each branch corresponds to a possible execution path. Apart from the usual logical operators ($\wedge$ and $\vee$), CTL offers temporal operators ($\Box, \Diamond, EF, AF, EG, AG, EU, AU$) to describe the behaviour of computer programs and its evolution over time.

Definition 2.3.1 (CTL assertions)**.** *CTL assertions* are described by the grammar:

$$\begin{aligned}\varphi \ ::= \ & P \,|\, \mathsf{error} \,|\, \mathsf{final} \,|\, \varphi \wedge \varphi \,|\, \varphi \vee \varphi \,|\, \Diamond\varphi \,|\, \Box\varphi \,| \\ & EF\varphi \,|\, AF\varphi \,|\, EG\varphi \,|\, AG\varphi \,|\, E(\varphi U \varphi) \,|\, A(\varphi U \varphi)\end{aligned}$$

where P ranges over memory state assertions (Definition 2.2.1).

Note that we use $\Diamond$ in favour of EX for the existential quantification over the immediate next configuration. Similarly, we use $\Box$ in favour of AX for the universally quantified immediate next configuration. Also, final and error denote final and faulting configurations respectively.

Definition 2.3.2. A (program) configuration γ is a *model* of the CTL assertion φ if the relation $\gamma \vDash \varphi$ holds, defined by structural induction on φ:

$$\begin{array}{rcl}
\gamma \vDash P & \Leftrightarrow & (\gamma_s, \gamma_h) \vDash P \\
\gamma \vDash \mathsf{error} & \Leftrightarrow & \gamma = \mathsf{fault} \\
\gamma \vDash \mathsf{final} & \Leftrightarrow & \gamma_C = \varepsilon \\
\gamma \vDash \varphi_1 \wedge \varphi_2 & \Leftrightarrow & \gamma \vDash \varphi_1 \text{ and } \gamma \vDash \varphi_2 \\
\gamma \vDash \varphi_1 \vee \varphi_2 & \Leftrightarrow & \gamma \vDash \varphi_1 \text{ or } \gamma \vDash \varphi_2 \\
\gamma \vDash \Diamond\varphi & \Leftrightarrow & \exists \gamma'.\ \gamma \rightsquigarrow \gamma' \text{ and } \gamma' \vDash \varphi \\
\gamma \vDash \Box\varphi & \Leftrightarrow & \forall \gamma'.\ \gamma \rightsquigarrow \gamma' \text{ implies } \gamma' \vDash \varphi \\
\gamma \vDash EF\varphi & \Leftrightarrow & \exists \gamma'.\ \gamma \rightsquigarrow^* \gamma' \text{ and } \gamma' \vDash \varphi \\
\gamma \vDash AF\varphi & \Leftrightarrow & \forall \pi \text{ starting from } \gamma.\ \exists \gamma' \in \pi. \gamma' \vDash \varphi \\
\gamma \vDash EG\varphi & \Leftrightarrow & \exists \pi \text{ starting from } \gamma.\ \forall \gamma' \in \pi. \gamma' \vDash \varphi \\
\gamma \vDash AG\varphi & \Leftrightarrow & \forall \gamma'.\ \text{ if } \gamma \rightsquigarrow^* \gamma' \text{ then } \gamma' \vDash \varphi \\
\gamma \vDash E(\varphi_1 U \varphi_2) & \Leftrightarrow & \exists \pi \text{ starting from } \gamma. \\
 & & \exists i \geq 0.\ \pi[i] \vDash \varphi_2 \text{ and } \forall j{:}0 \leq j < i. \pi[j] \vDash \varphi_1 \\
\gamma \vDash A(\varphi_1 U \varphi_2) & \Leftrightarrow & \forall \pi \text{ starting from } \gamma. \\
 & & \exists i \geq 0.\ \pi[i] \vDash \varphi_2 \text{ and } \forall j{:}0 \leq j < i. \pi[j] \vDash \varphi_1
\end{array}$$

2.3.2 LTL assertions

In contrast to branching temporal logics, in linear time logics, the truth of temporal logic formulas are defined on execution paths. This supports the linear view of time, where a computation starting from a state is seen as a single sequence of states.

Definition 2.3.3. *LTL assertions* are described by the grammar:

$$\psi ::= P \mid \mathsf{error} \mid \mathsf{final} \mid \psi \wedge \psi \mid \psi \vee \psi \mid X\psi \mid F\psi \mid G\psi \mid (\psi U \psi)$$

where P ranges over memory state assertions (Definition 2.2.1) extended with prophecy variables to correctly handle nondeterminism; see Chapter 4 for details.

Definition 2.3.4. An execution path π is a *model* of an LTL temporal formula ψ if the relation $\pi \vDash \psi$ holds, defined by structural induction on ψ:

$$\begin{array}{rcl}
\pi \vDash Q & \Leftrightarrow & (\pi[0]_s, \pi[0]_h) \vDash Q \\
\pi \vDash \mathsf{error} & \Leftrightarrow & \pi[0] = \mathsf{fault} \\
\pi \vDash \mathsf{final} & \Leftrightarrow & \pi[0]_C = \varepsilon \\
\pi \vDash \psi_1 \wedge \psi_2 & \Leftrightarrow & \pi \vDash \psi_1 \text{ and } \pi \vDash \psi_2 \\
\pi \vDash \psi_1 \vee \psi_2 & \Leftrightarrow & \pi \vDash \psi_1 \text{ or } \pi \vDash \psi_2 \\
\pi \vDash X\psi & \Leftrightarrow & \pi_1 \vDash \psi \\
\pi \vDash F\psi & \Leftrightarrow & \exists k \geq 0.\, \pi_k \vDash \psi \\
\pi \vDash G\psi & \Leftrightarrow & \forall k \geq 0.\, \pi_k \vDash \psi \\
\pi \vDash \psi_1 U \psi_2 & \Leftrightarrow & \exists k \geq 0.\, \pi_k \vDash \psi_2 \text{ and } \forall 0 \leq j \leq k.\, \pi_j \vDash \psi_1
\end{array}$$

The subtle difference between considering computation as a tree of executions (in CTL) and as a collection of executions (in LTL) can be exemplified as follows.

Example 2.3.5. *Assume the following program starts its execution from an initial program state where $x = true$.*

```
while(*) {
  x:=true;
}
x:=false;
x:=true;
while(true) {
  skip;
}
```

Suppose that we attempt to demonstrate that for every execution in the program, x will become true and remain true for the rest of the program execution. In CTL, this property would be expressed as $AFAG(x = true)$, which is invalid due to the existence of an execution path which fails to exhibit $AG(x = true)$. Concretely, consider the path π that executes the loop n times, reaching a configuration with command **while** $*$ **do** $x := true$ **od**; $x := false$, *say γ_n. The possibility of exiting the loop and reaching the configuration with command $x := false$ establishes that $\gamma_n \nvDash AG(x = true)$.*

However, the analogous LTL property $FG(x = true)$ does *hold. This is because when considering each individual execution of the program, we find that in each one there is a point from which $x = true$ is always true onwards; for the executions that reach $x := false$ the condition will also hold once the execution reaches the succeeding assignment $x := true$.*

Chapter 3

CTL Proof System

In this chapter, we present our cyclic proof system for establishing CTL properties of programs. In Section 3.1 we give the proof rules of our system, which we categorise into *symbolic execution*, *faulting execution* and *logical rules*. The symbolic execution rules are adapted from those in [19], accounting for whether an existentially quantified ($\Diamond$) or universally quantified ($\Box$) path formula property is being established. The faulting execution rules allow us to prove that a program execution faults. The logical rules comprise standard rules for the logical connectives and standard unfolding rules for the temporal operators and inductive predicates. In Section 3.2 we formulate the global soundness condition that ensures that cyclic proofs in our system are sound and demonstrate its applicability on a couple of examples. Finally, in Section 3.3 we show that our proof rules are sound with respect to the operational semantics for programs described in the previous chapter and show the global soundness of our cyclic proof system.

3.1 CTL proof rules

In this section we give the proof rules of our system. We use Hoare logic due to its common use as an elegant framework in which to write formal proofs as well as its common use underlying automated theorem-proving tools.

Judgements in our system are of the form $P \vdash C : \varphi$, where P is a symbolic heap formula as per Definition 2.2.1; C is a sequence of commands as per Definition 2.1.1 (i.e. a computer program); and φ is a temporal assertion written in the

CTL language described in Definition 2.3.1. A proof rule (R) is written as:

$$\frac{P_1 \vdash C_1 : \varphi_1 \ldots P_n \vdash C_n : \varphi_n}{P \vdash C : \varphi}\ (R)$$

where $n \in \mathbb{N}$; the sequents above the line are called the *premises* of the rule while the sequent below the line is called the *conclusion* of the rule. A rule with no premises is called an *axiom*. We write s_P, s_C and s_φ to refer to the precondition, program command and temporal property components of sequent s, respectively.

The interpretation of judgements for CTL is as follows:

Definition 3.1.1 (CTL judgement)**.** A CTL judgement $P \vdash C : \varphi$ is *valid* if and only if, for all memory states (s,h) such that $s,h \models P$, we have $\langle C,s,h\rangle \models \varphi$.

Our proof rules for CTL judgements are shown in Figure 3.1. The *symbolic execution* rules for commands are adapted from those in the proof system for program termination in [19], accounting for whether a diamond $\Diamond$ or box $\Box$ property is being established. The dichotomy between $\Diamond$ and $\Box$ is only visible for the nondeterministic components of a program. In the specific case of our language, the nondeterministic constructs are (i) nondeterministic while; (ii) nondeterministic if; and (iii) memory allocation; it is only for these constructs that we need a specific rule for each case. Incidentally, the difference between E properties and A properties is basically the same as the difference between $\Diamond$ and $\Box$, but extended to execution paths rather than individual steps.

We introduce *faulting execution* rules to allow us to prove that a program faults due to attempting to access memory locations outside the program heap. In the case of our programming language, the constructs that require to access memory that has been previously allocated are (i) $x := [E]$; (ii) $[E] := E$; and (iii) $\mathbf{free}(E)$; we provide a faulting execution rule for each of these cases. The side condition for these three rules requires that the separating conjunction of the precondition P with the point-to formula $E \mapsto nil$ is unsatisfiable. Intuitively this condition precludes the memory location $[\![E]\!]s$ from being in the domain of the heap for models that satisfy P, hence causing a memory violation that results in a faulting execution.

The logical rules comprise standard rules for the logical connectives and standard unfolding rules for the temporal operators and inductive predicates. As described in Section 2.1, we consider an execution path to be a maximally finite or infinite sequence of configurations. To correctly account for maximally finite paths we introduce (EG-Finite), otherwise we would not be able to prove EG properties over finite paths (as the (EG) rule requires the existence of an infinite succession of configurations). Note that a corresponding (AG-Finite) rule is not required since any $\square$ property is trivially true of any configuration with no successive configurations.

The (Unfold-Pre) rule performs a case-split on an inductive predicate in the precondition by replacing the predicate with the body of each clause of its inductive definition. For example, consider the inductive predicate for list segments from Example 2.2.5; its inductive definition determines the following (Unfold-Pre) rule:

$$\frac{x = y, \Pi : \mathsf{emp} * F \vdash C : \varphi \qquad \Pi : F * x \mapsto x' * ls(x', y) \vdash C : \varphi}{\Pi : F * ls(x, y) \vdash C : \varphi} \text{ (Unfold-Pre)}$$

3.2 CTL cyclic proofs

Proofs in our system are *cyclic proofs*: standard derivation trees in which open subgoals can be closed either by applying an axiom or by forming a *back-link* to an identical interior node. To ensure that such structures correspond to sound proofs, a global soundness condition is imposed. The following definitions, adaptations of similar notions in e.g. [17, 19, 22], formalise this notion.

Definition 3.2.1 (Derivation graph)**.** Let Seqs denote the set of all well-formed CTL judgements and Rules denote the set of rules of the CTL proof system. Then a *derivation graph* $\mathcal{G}$ is given by the tuple (V, s, r, p), where V is a finite set of nodes, $s : V \to \text{Seqs}$ is a total function mapping nodes to sequents, $r : V \to \text{Rules}$ is a partial function mapping nodes to rules, and $p : \mathbb{N} \times V \to V$ is a partial function defined just

Symbolic execution rules:

$$\frac{}{P \vdash C : \mathsf{final}}\ (\text{Final}) \qquad \frac{P \vdash C : \varphi}{P \vdash (\textbf{skip} \ ; C) : \bigcirc\varphi}\ (\text{Skip})$$

$$\frac{x = E[x'/x], P[x'/x] \vdash C : \varphi}{P \vdash (x := E \ ; C) : \bigcirc\varphi}\ (\text{Assign}) \qquad \frac{x = E'[x'/x], (P * E \mapsto E')[x'/x] \vdash C : \varphi}{P * E \mapsto E' \vdash (x := [E] \ ; C) : \bigcirc\varphi}\ (\text{Read})$$

$$\frac{P * E \mapsto E' \vdash C : \varphi}{P * E \mapsto - \vdash ([E] := E' \ ; C) : \bigcirc\varphi}\ (\text{Write}) \qquad \frac{B, P \vdash C_1 \ ; C_3 : \varphi \quad \neg B, P \vdash C_2 \ ; C_3 : \varphi}{P \vdash (\textbf{if}\, B \,\textbf{then}\, C_1 \,\textbf{else}\, C_2 \,\textbf{fi} \ ; C_3) : \bigcirc\varphi}\ (\text{If})$$

$$\frac{B, P \vdash (C_1 \ ; \textbf{while}\, B \,\textbf{do}\, C_1 \,\textbf{od} \ ; C_2) : \varphi \quad \neg B, P \vdash C_2 : \varphi}{P \vdash (\textbf{while}\, B \,\textbf{do}\, C_1 \,\textbf{od} \ ; C_2) : \bigcirc\varphi}\ (\text{Wh}) \qquad \frac{P \vdash C : \varphi}{P * E \mapsto - \vdash (\text{free}(E) \ ; C) : \bigcirc\varphi}\ (\text{Free})$$

$$\frac{P[x'/x] * x \mapsto v \vdash C : \varphi \quad v \in \text{Val}}{P \vdash (x := alloc() \ ; C) : \Diamond\varphi}\ (\text{Alloc}\Diamond) \qquad \frac{P[x'/x] * x \mapsto v \vdash C : \varphi}{P \vdash (x := alloc() \ ; C) : \Box\varphi}\ v \text{ fresh } (\text{Alloc}\Box)$$

$$\frac{P \vdash C_1 \ ; C_3 : \varphi}{P \vdash (\textbf{if} * \textbf{then}\, C_1 \,\textbf{else}\, C_2 \,\textbf{fi} \ ; C_3) : \Diamond\varphi}\ (\text{If* }\Diamond 1) \qquad \frac{P \vdash C_2 \ ; C_3 : \varphi}{P \vdash (\textbf{if} * \textbf{then}\, C_1 \,\textbf{else}\, C_2 \,\textbf{fi} \ ; C_3) : \Diamond\varphi}\ (\text{If* }\Diamond 2)$$

$$\frac{P \vdash C_1 \ ; C_3 : \varphi \quad P \vdash C_2 \ ; C_3 : \varphi}{P \vdash (\textbf{if} * \textbf{then}\, C_1 \,\textbf{else}\, C_2 \,\textbf{fi} \ ; C_3) : \Box\varphi}\ (\text{If*}\Box) \qquad \frac{P \vdash (C_1 \ ; \textbf{while} * \textbf{do}\, C_1 \,\textbf{od} \ ; C_2) : \varphi \quad P \vdash C_2 : \varphi}{P \vdash (\textbf{while} * \textbf{do}\, C_1 \,\textbf{od} \ ; C_2) : \Box\varphi}\ (\text{Wh*}\Box)$$

$$\frac{P \vdash (C_1 \ ; \textbf{while} * \textbf{do}\, C_1 \,\textbf{od} \ ; C_2) : \varphi}{P \vdash (\textbf{while} * \textbf{do}\, C_1 \,\textbf{od} \ ; C_2) : \Diamond\varphi}\ (\text{Wh* }\Diamond 1) \qquad \frac{P \vdash C_2 : \varphi}{P \vdash (\textbf{while} * \textbf{do}\, C_1 \,\textbf{od} \ ; C_2) : \Diamond\varphi}\ (\text{Wh* }\Diamond 2)$$

Faulting execution rules:

$$\frac{P * E \mapsto \mathsf{nil} \not\models \bot}{P \vdash (x := [E] \ ; C) : \bigcirc\mathsf{error}}\ (\text{R}\bot) \qquad \frac{P * E \mapsto \mathsf{nil} \not\models \bot}{P \vdash ([E] := E' \ ; C) : \bigcirc\mathsf{error}}\ (\text{W}\bot) \qquad \frac{P * E \mapsto \mathsf{nil} \not\models \bot}{P \vdash (\text{free}(E) \ ; C) : \bigcirc\mathsf{error}}\ (\text{Free}\bot)$$

Logical rules:

$$\frac{P \models Q}{P \vdash C : Q}\ (\text{Check}) \qquad \frac{}{\bot \vdash C : \varphi}\ (\text{Ex.Falso}) \qquad \frac{\Omega_1 \vdash C : \varphi \quad \Omega_2 \vdash C : \varphi}{\Omega_1 \vee \Omega_2 \vdash C : \varphi}\ (\text{Split})$$

$$\frac{P \vdash C : \varphi \quad x \notin \text{vars}(C)}{P[E/x] \vdash C : \varphi[E/x]}\ (\text{Subst}) \qquad \frac{P \vdash C : \varphi_1 \quad P \vdash C : \varphi_2}{P \vdash C : \varphi_1 \wedge \varphi_2}\ (\text{Conj}) \qquad \frac{P \vdash C : \varphi_i \quad i \in \{1,2\}}{P \vdash C : \varphi_1 \vee \varphi_2}\ (\vee)$$

$$\frac{P \vdash C : \varphi \vee \Diamond EF\varphi}{P \vdash C : EF\varphi}\ (\text{EF}) \qquad \frac{P \vdash C : \varphi \quad P \vdash C : \Diamond EG\varphi}{P \vdash C : EG\varphi}\ (\text{EG}) \qquad \frac{P \vdash C : \psi \vee (\varphi \wedge \Diamond E(\varphi U \psi))}{P \vdash C : E(\varphi U \psi)}\ (\text{EU})$$

$$\frac{P \vdash C : \varphi \vee \Box AF\varphi}{P \vdash C : AF\varphi}\ (\text{AF}) \qquad \frac{P \vdash C : \varphi \quad P \vdash C : \Box AG\varphi}{P \vdash C : AG\varphi}\ (\text{AG}) \qquad \frac{P \vdash C : \psi \vee (\varphi \wedge \Box A(\varphi U \psi))}{P \vdash C : A(\varphi U \psi)}\ (\text{AU})$$

$$\frac{P \vdash \varepsilon : \varphi}{P \vdash \varepsilon : EG\varphi}\ (\text{EG-Finite}) \qquad \frac{P \vdash Q \quad Q \vdash C : \psi \quad \psi \vdash \varphi}{P \vdash C : \varphi}\ (\text{Cons})$$

$$\frac{(\Pi \cup \Pi'_i : \Sigma * \Sigma'_i \vdash C : \varphi)_{1 \le i \le k}}{\Pi : \Psi(\mathbf{E}) * \Sigma \vdash C : \varphi} \left(\begin{array}{c} \Pi_1 : \Sigma_1 \Rightarrow \Psi(\mathbf{E}_1), \ldots, \Pi_k : \Sigma_k \Rightarrow \Psi(\mathbf{E}_k) \\ \Pi'_i : \Sigma'_i = \Pi_i : \Sigma_i \text{ with existential variables freshened and} \\ \text{arguments } \mathbf{E} \text{ substituted for parameters } \mathbf{E}_i \end{array} \right)\ (\text{Unfold-Pre})$$

Figure 3.1: Proof rules for CTL judgements. We write $\bigcirc\varphi$ to mean "either $\Box\varphi$ or $\Diamond\varphi$".

in case $r(v)$ is a rule with j premises, $1 \le i \le j$ and

$$\frac{s(p(1,v)) \quad s(p(2,v)) \quad \dots \quad s(p(j,v))}{s(v)}\ r(v)$$

Definition 3.2.2 (Derivation tree)**.** A derivation graph $\mathcal{D} = (V,s,r,p)$ is a *derivation tree* if there is a distinguished node $\text{root}(\mathcal{D}) \in V$ such that for all $v \in V$, there is a unique path in $\mathcal{D}$ from $\text{root}(\mathcal{D})$ to v.

Definition 3.2.3 (Bud nodes)**.** Let $\mathcal{D} = (V,s,r,p)$ be a finite derivation tree. A *bud node* of $\mathcal{D}$ is a vertex $b \in V$ such that b is not the conclusion of any proof rule instance (i.e. $r(b)$ is undefined). We write $\text{Bud}(\mathcal{D})$ to denote the set of all bus nodes ocurring in $\mathcal{D}$.

Definition 3.2.4 (Companion)**.** Let $\mathcal{D} = (V,s,r,p)$ be a derivation tree and let $b \in \text{Bud}(\mathcal{D})$. A node $c \in V$ is said to be a *companion* for b if $s(c) = s(b)$ and $r(c)$ is defined.

Definition 3.2.5 (Pre-proof)**.** A *pre-proof* is a pair $\mathcal{P} = (\mathcal{D},\mathcal{L})$, where $\mathcal{D}$ is a finite derivation tree constructed according to the proof rules and $\mathcal{L} : V \to V$ is a partial function assigning to every bud node $b \in \text{Bud}(\mathcal{D})$ one of its companions.

As usual, a pre-proof $\mathcal{P} = (\mathcal{D},\mathcal{L})$ can be understood as a finite cyclic graph by identifying each open leaf of $\mathcal{D}$ with its companion, and a *path* in $\mathcal{P}$ is then just a path in this graph.

Definition 3.2.6 (Pre-proof graph)**.** Let $\mathcal{P} = (\mathcal{D},\mathcal{L})$ be a pre-proof, where $\mathcal{D} = (V,s,r,p)$. Then, the *graph* of $\mathcal{P}$, written $\mathcal{G}_\mathcal{P}$ is the derivation graph obtained from $\mathcal{D}$ by identifying each bud node $b \in \text{Bud}(\mathcal{D})$ with its companion $\mathcal{L}(b)$. In other words, $\mathcal{G}_\mathcal{P} = (V',s,r,p')$, where $V' = V \setminus \text{Bud}(\mathcal{D})$ and p' is defined by

$$p'(j,v) = \begin{cases} \mathcal{L}(p(j,v)) & \text{if } p(j,v) \in \text{Bud}(\mathcal{D}) \\ p(j,v) & \text{otherwise} \end{cases}$$

for each $j \in \mathbb{N}$. That is to say $\mathcal{G}_\mathcal{P}$ contains no bud nodes and the rule labelling function r is total on V'.

To qualify as a proof, a cyclic pre-proof must satisfy a global soundness condition, defined using the notion of a *trace* along a path in a pre-proof.

Definition 3.2.7 (Temporal trace)**.** Let $(J_i = P_i \vdash C_i : \varphi_i)_{i\geq 0}$ be a path in a pre-proof $\mathcal{P}$. The sequence of temporal formulas along the path, $(\varphi_i)_{i\geq 0}$, is a $\Box$*-trace (*$\Diamond$*-trace)* following that path $(J_i)_{i\geq 0}$ if there exists a formula ψ such that, for all $i \geq 0$, the following both hold:

(i) the formula φ_i is of the form $AG\psi$ $(EG\psi)$ or $\Box AG\psi$ $(\Diamond EG\psi)$; and

(ii) $\varphi_i = \varphi_{i+1}$ whenever J_i is the conclusion of the consequence rule (Cons).

We say that a temporal trace *progresses* whenever a symbolic execution rule is applied. A temporal trace is *infinitely progressing* if it progresses at infinitely many points.

We also take account of *precondition traces* arising from inductive predicates in the precondition, analogous to [19]. Roughly speaking, a precondition trace tracks an occurrence of a predicate in the preconditions of the judgements along the path, progressing whenever the predicate occurrence is unfolded.

Definition 3.2.8 (Precondition trace)**.** Let $(J_i = P_i \vdash C_i : \varphi_i)_{i\geq 0}$ be a path in a pre-proof $\mathcal{P}$. The sequence of symbolic heap formulas along the path, $(\Psi_i)_{i\geq 0}$, is a *precondition trace* following that path $(J_i)_{i\geq 0}$ if:

(i) Whenever J_i is the conclusion of the (Unfold-Pre) rule, the predicate $\Psi(\mathbf{E})$ is the predicate in the spatial formula of P_i being unfolded and $\Psi_{i+1} = \Psi'(\mathbf{E})$, where $\Psi'(\mathbf{E})$ is obtained in the premise J_{i+1} by unfolding $\Psi(\mathbf{E})$; and

(ii) $\Psi_i = \Psi_{i+1}$ (modulo any rewriting done by rules (Assign), (Read), (Alloc$\Box$), (Alloc$\Diamond$), (Subst)) for all other rules.

We say that a precondition trace *progresses* whenever (Unfold-Pre) is applied. A precondition trace is *infinitely progressing* if it progresses at infinitely many points.

Example 3.2.9. *The following examples are intended to show how traces are tracked for both conditions of Definition 3.2.8.*

For condition (i) we show the (progressing) trace induced as result of unfolding the ls inductive predicate defined in Example 2.2.5:

$$\frac{x = y : \mathsf{emp} \vdash C : \varphi \qquad x \mapsto x' * \underline{ls(x',y)} \vdash C : \varphi}{\underline{ls(x,y)} \vdash C : \varphi}\ (\textit{Unfold-Pre})$$

where $\Psi_i = \underline{ls(x,y)}$ *and* $\Psi_{i+1} = \underline{ls(x',y)}$.

For condition (ii) we show the (non-progressing) precondition trace involved in the application of the rule (Assign) (the treatment for other rules is analogous):

$$\frac{tmp = true : \underline{ls(x,y)} \vdash C : \varphi}{\underline{ls(x,y)} \vdash tmp := true\ ;\ C : \bigcirc\varphi}\ (\textit{Assign})$$

where $\Psi_i = \Psi_{i+1} = \underline{ls(x,y)}$.

Definition 3.2.10 (Proof)**.** A pre-proof $\mathcal{P}$ is a *proof* if it satisfies the following *global soundness condition*: for every infinite path $(P_i \vdash C_i : \varphi_i)_{i \geq 0}$ in $\mathcal{P}$, there is an infinitely progressing temporal ($\Box$- or $\Diamond$-) trace or precondition trace following some tail $(P_i \vdash C_i : \varphi_i)_{i \geq n}$ of the path.

Example 3.2.11. *The following simple example has been designed to illustrate complex cycle structures that arise from the nesting of temporal operators to provide a non-trivial satisfaction of the soundness condition.*

Consider the following program:

```
1: if(*){
2:   x:=1
 } else {
3:   skip;
 }
4: while(x=x) {
5:   skip;
}
```

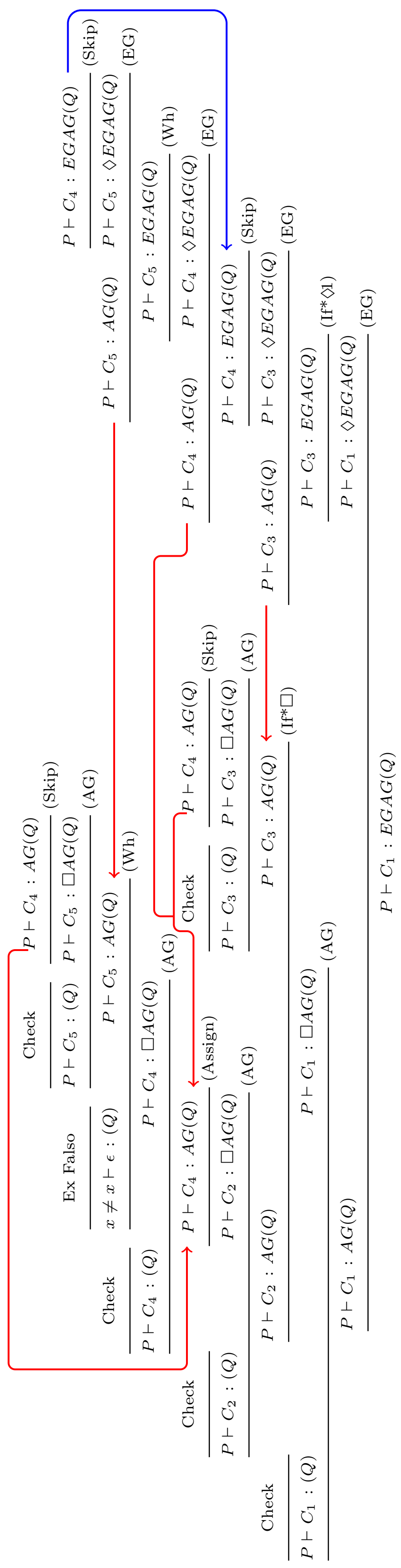


Figure 3.2: Nested temporal operators example

where each atomic command is labelled with a program counter.

One can observe that, under the precondition $P = (x = 1)$, the program has the invariant property AG(x=1), since the assignment command on line 2 does not break the invariant and the variable will not be updated throughout the rest of the program execution. Moreover, since there exists at least one program execution in which the invariant holds, the program satisfies the formula $EGAG(x = 1)$. Figure 3.2 shows the proof of this property in our system including the 6 cycles that are formed during the proof search along with the traces that follow the infinite paths.

For the right-most cycle we note that the temporal component of the sequents in the infinite proof path are of the form $EG\psi$ or $\Diamond EG\psi$, where $\psi = AG(x = 1)$, so that there is a $\Diamond$-trace following the proof path as per Definition 3.2.7. Moreover, due to the application of symbolic execution rules (Wh) and $(Skip)$ along the infinite proof path, the trace progresses infinitely often.

Similarly, for the left-most cycle we note that the temporal component of the sequents in the infinite proof path are of the form $AG\psi$ or $\Box AG\psi$, where $\psi = (x = 1)$, so that there is a $\Box$-trace following the proof path as per Definition 3.2.7. Moreover, due to the application of symbolic execution rules (Wh) and $(Skip)$ along the infinite proof path, the trace progresses infinitely often.

Contrary to the previous two cycles, the remanding 4 back-links shown in the proof do not match their corresponding leaf node to a direct descendant. Nevertheless, these infinite paths are, too, followed by $\Box$-traces that progress infinitely often.

Consequently, our pre-proof qualifies as a valid cyclic proof since along every infinite path there is either a $\Box$- or a $\Diamond$- trace progressing infinitely often.

Example 3.2.12. *On a more realistic example, we now present a proof of a heap-aware server program that nondeterministically alternates between adding an arbitrary number of "job requests" to the head of a linked-list and processing job requests by means of deleting them from the list:*

```
while(true){
  if(*) {
    while(x!=nil) {
      temp:=x.next;
      free(x);
      x:=temp;
    }
  } else {
    while(*) {
      y:=new();
      y.next:=x;
      x:=y;
    }
  }
}
```

We can show that, given that the heap is initially a linked list from x *to* nil *(commonly written in separation logic as* $ls(x,nil)$*), it is always possible for the heap to become empty at any point during program execution. Writing C for our server program, this property is expressed as the judgement* $ls(x,nil) \vdash C : AGEF(\mathsf{emp})$.

Figure 3.3 shows a cyclic proof of this judgement in our system, where $AGEF(\mathsf{emp})$ *is replaced by* φ *and* $EF(\mathsf{emp})$ *is replaced by* ψ *due to space constraints.*

For the cycles depicted in red, we note that the temporal component of the sequents in the infinite proof path are of the form $AG\psi$ *or* $\Box AG\psi$*, where* $\psi = EF(\mathsf{emp})$*, so that there is a* $\Box$*-trace following the proof path as per Definition 3.2.7. Moreover, due to the application of symbolic executions rules along the proof path, the traces progresses infinitely often.*

Note that the back-links depicted in green do not form infinite loops as they all point to a companion that eventually leads to a $(Check)$ *axiom; as such, no trace is required to follow these paths (indeed there are no traces following these paths in the proof). These back-links resulting on a finite proof path could be intuitively seen* *as lemma applications of previously discovered proofs. Alternatively, one can think of simply replicating the tails of the path above the respective leaf node to produce a similarly sound proof.*

Consequently, the pre-proof qualifies as a valid cyclic proof since there is an infinitely progressing $\Box$-trace along every infinite path.

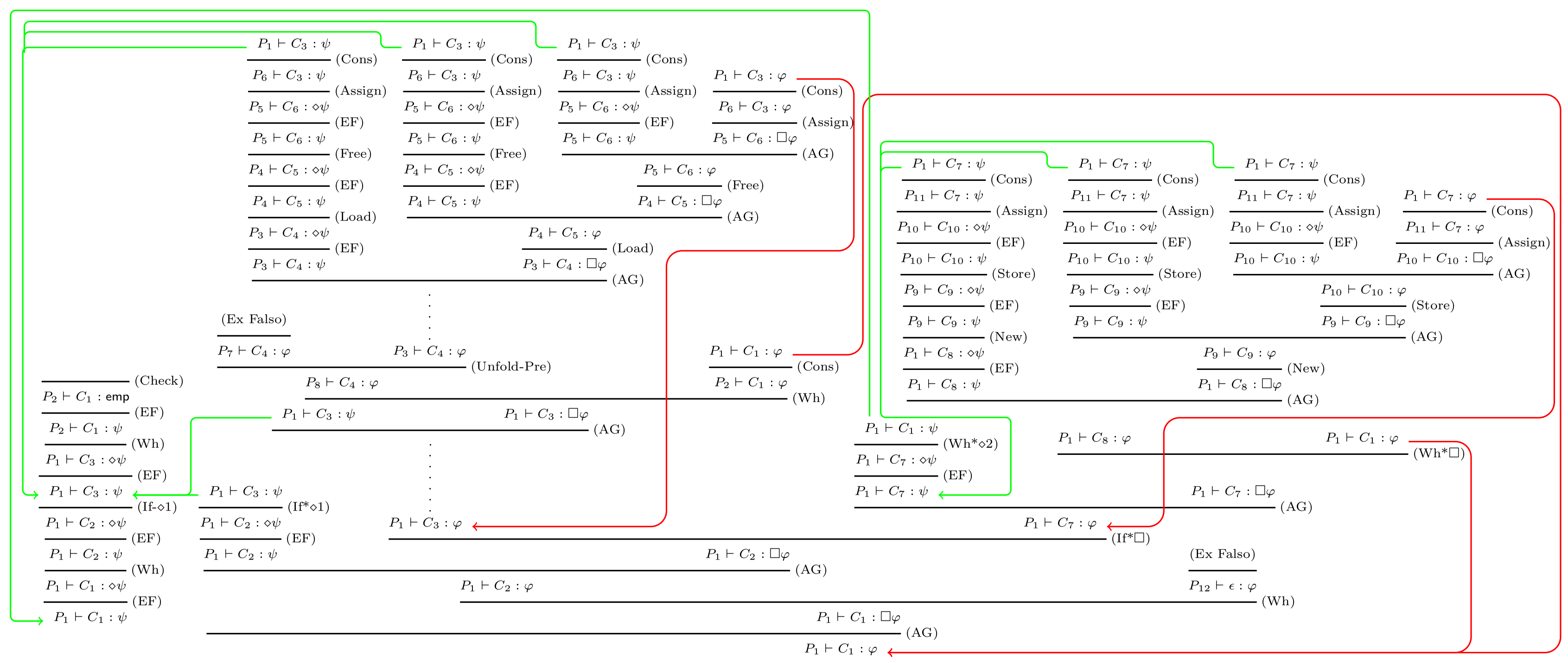
3.3 Soundness of CTL system

In this section we show that our proof system is sound. We first show local soundness of the proof rules along with the trace properties that are maintained by all derivation rules, as established in Definitions 3.2.7 and 3.2.8. For each proof rule, we show soundness from conclusion to premises by assuming that the conclusion is invalid (by Definition 3.1.1) and proceeding to establish the invalidity (of at least one) of the premise(s). In the case of the axioms, we show that the conclusion is a tautology. We then show the global soundness of our system, essentially, by extending the properties established for local soundness to paths in a pre-proof.

Lemma 3.3.1. *Let $J = (P \vdash C : \varphi)$ be the conclusion of a proof rule R. If J is invalid under (s,h), then there exists a premise of the rule $J' = (P' \vdash C' : \varphi')$ and a model (s',h') such that J' is not valid under (s',h') and, furthermore,*

1. *if there is a $\Box$-trace (φ,φ') following the edge (J,J') then, letting ψ be the unique formula given by Definition 3.2.7, there is a configuration γ such that $\gamma \not\models \psi$, and the finite execution path $\pi' = \langle C',s',h'\rangle \dots \gamma$ is well-defined and a subpath of $\pi = \langle C,s,h\rangle \dots \gamma$. Therefore* $\text{length}(\pi') \leq \text{length}(\pi)$. *Moreover,* $\text{length}(\pi) < \text{length}(\pi')$ *when R is a symbolic execution rule.*

2. *if there is a $\Diamond$-trace (φ,φ') following the edge (J,J') then, letting ψ be the unique formula given by Definition 3.2.7, there is a smallest finite execution tree κ with root $\langle C,s,h\rangle$, each of whose leaves γ satisfies $\gamma \not\models \psi$. Moreover, κ has a subtree κ' with root $\langle C',s',h'\rangle$ and whose leaves are all leaves of κ. Therefore* $\text{height}(\kappa') \leq \text{height}(\kappa)$. *Moreover,* $\text{height}(\kappa') < \text{height}(\kappa)$ *when R is a symbolic execution rule.*



$[P_1] = ls(x, \texttt{nil})$

$[P_2] = \texttt{nil} = x\colon ls(x, \texttt{nil})$

$[P_3] = \texttt{nil} \neq x\colon x \mapsto z' * ls(z', \texttt{nil})$

$[P_4] = \texttt{nil} \neq x\colon x \mapsto temp * ls(temp, \texttt{nil})$

$[P_5] = \texttt{nil} \neq x\colon ls(temp, \texttt{nil})$

$[P_6] = x = temp \wedge z' \neq \texttt{nil}\colon ls(temp, \texttt{nil})$

$[P_7] = \texttt{nil} = x \wedge \texttt{nil} \neq x$

$[P_8] = ls(x, \texttt{nil})$

$[P_9] = \texttt{nil} \neq y\colon y \mapsto y' * ls(x, \texttt{nil})$

$[P_{10}] = \texttt{nil} \neq y\colon y \mapsto x * ls(x, \texttt{nil})$

$[P_{11}] = x = y \wedge \texttt{nil} \neq y\colon y \mapsto z' * ls(z', \texttt{nil})$

$[P_{12}] = x \neq x\colon ls(x, \texttt{nil})$

Figure 3.3: Single threaded monolithic server example where where $AGEF(\text{emp})$ is replaced by φ and $EF(\text{emp})$ is replaced by ψ

3. *if there is a precondition trace* $(\Psi(\mathbf{E}), \Psi'(\mathbf{E}))$ *following the edge* (J, J') *then letting* α *(*β*) be the least approximant for which the inductive predicate* $\Psi(\mathbf{E})$ *(*$\Psi'(\mathbf{E})$*) is interpreted(i.e.* $(s,h) \vDash \Psi^{\alpha}(\mathbf{E})$ *and* $(s',h') \vDash \Psi'^{\beta}(\mathbf{E})$*), then the following relation holds and it is well-defined:* $\beta \le \alpha$*. Moreover* $\beta < \alpha$ *when* R *is the (Unfold-Pre) rule.*

Proof. We proceed by case analysis of the proof rule R.

Soundness of Final

$$\frac{}{P \vdash \varepsilon : \mathsf{final}}\ (\text{Final})$$

Pick an arbitrary program state s,h such that $(s,h) \vDash P$. We need to show that $\langle \varepsilon, s, h\rangle \vDash \mathsf{final}$. Trivial by Definition 2.3.2 (as $\langle \varepsilon, s, h\rangle_C = \varepsilon$).

Soundness of Skip

$$\frac{P \vdash C : \varphi}{P \vdash \mathbf{skip}\ ; C : \bigcirc\varphi}\ (\text{Skip})$$

Assume the conclusion of the rule is invalid. Pick an arbitrary program state s,h such that $(s,h) \vDash P$ but the program configuration $\langle \mathbf{skip}\ ; C, s, h\rangle \nvDash \bigcirc\varphi$. By Definition 2.3.2, if $\langle \mathbf{skip}\ ; C, s, h\rangle \nvDash \bigcirc\varphi$ then there exists a configuration γ such that $\langle \mathbf{skip}\ ; C, s, h\rangle \rightsquigarrow \gamma$ and $\gamma \nvDash \varphi$. By the operational semantics of our programming language we know that $\gamma = \langle C, s, h\rangle$ since $\langle \mathbf{skip}\ ; C, s, h\rangle \rightsquigarrow \langle C, s, h\rangle$. Consequently, since by our assumption $(s,h) \vDash P$ and $\gamma_s = s$ and $\gamma_h = h$ then $(\gamma_s, \gamma_h) \vDash P$. On the other hand, since $\gamma \nvDash \varphi$ then it is the case that the premise of the rule is invalid.

If there is a $\Box$-trace following the edge, then by Definition 3.2.7, $\varphi = AG\psi$. Furthermore, since by our previous invalidity result $\langle C, s, h\rangle \nvDash AG\psi$, then by Definition 2.3.2 we know that there exists a configuration $\gamma_{\neg}$ such that $\langle C, s, h\rangle \rightsquigarrow^{*} \gamma_{\neg}$ and $\gamma_{\neg} \nvDash \psi$. Finally, since by the operational semantics there is a single possible transition from $\langle \mathbf{skip}\ ; C, s, h\rangle$, namely $\langle \mathbf{skip}\ ; C, s, h\rangle \rightsquigarrow \langle C, s, h\rangle$, then every execution path π' starting from $\langle C, s, h\rangle$ is a subpath of a path π starting from $\langle \mathbf{skip}\ ; C, s, h\rangle$. Consequently letting $\pi = \langle \mathbf{skip}\ ; C, s, h\rangle \rightsquigarrow^{*} \gamma_{\neg}$ and $\pi' = \langle C, s, h\rangle \rightsquigarrow^{*} \gamma_{\neg}$ then $\text{length}(\pi') < \text{length}(\pi)$.

If there is a $\Diamond$-trace following the edge, then by Definition 3.2.7, $\varphi = EG\psi$. Furthermore, since by our previous invalidity result $\langle C, s, h\rangle \nvDash EG\psi$, then by Definition 2.3.2 we know that for all paths π starting from $\langle C,s,h\rangle$ there exists a configuration $\gamma_\neg \in \pi$ such that $\gamma_\neg \not\models \psi$. In other words, there is a finite tree κ' with root in $\langle C,s,h\rangle$ whose all leaves $\gamma_\neg \not\models \psi$. Finally, since by the operational semantics there is a single possible transition from $\langle \mathbf{skip}\ ; C,s,h\rangle$, namely $\langle \mathbf{skip}\ ; C,s,h\rangle \rightsquigarrow \langle C,s,h\rangle$, then every tree κ' with root in $\langle C,s,h\rangle$ is a subtree of a tree κ with root in $\langle \mathbf{skip}\ ; C,s,h\rangle$, hence for all leaves $\gamma_\neg \in \kappa$ then $\gamma_\neg \in \kappa'$. Therefore $\text{height}(\kappa') < \text{height}(\kappa)$.

If there is a precondition trace $(\Psi(\mathbf{E}),\Psi'(\mathbf{E}))$ following the edge, where $(s,h) \models \Psi^\alpha(\mathbf{E})$ and $(s',h') \models \Psi'^\beta(\mathbf{E})$, then, since the precondition for both conclusion and premise is the same, then the inductive predicates $\Psi(\mathbf{E})$ and $\Psi'(\mathbf{E})$ in both conclusion and premise are interpreted by the same least approximant. Hence $\alpha = \beta$.

Soundness of Assign

$$\frac{x = E[x'/x], P[x'/x] \vdash C : \varphi}{P \vdash (x := E\ ; C) : \bigcirc\varphi}\ \text{(Assign)}$$

Assume the conclusion of the rule is invalid. Pick an arbitrary program state s,h such that $(s,h) \models P$ but the program configuration $\langle x := E\ ; C,s,h\rangle \not\models \bigcirc\varphi$. By Definition 2.3.2, if $\langle x := E\ ; C,s,h\rangle \not\models \bigcirc\varphi$ then there exists a configuration γ such that $\langle x := E\ ; C,s,h\rangle \rightsquigarrow \gamma$ and $\gamma \not\models \varphi$. By the operational semantics of our programming language we know that $\gamma = \langle C, s[x \mapsto [\![E]\!]s], h\rangle$ since $\langle x := E\ ; C,s,h\rangle \rightsquigarrow \langle C, s[x \mapsto [\![E]\!]s], h\rangle$.

By construction and Lemma 2.2.7 we know that $(s[x \mapsto [\![E]\!]s], h) \models x = E[x'/x]$ and by our assumption that $(s,h) \models P$ and Lemma 2.2.7 we know that $(s[x \mapsto [\![E]\!]s], h) \models P[x'/x]$. Consequently, since $(s[x \mapsto [\![E]\!]s], h) \models x = E[x'/x] \wedge P[x'/x]$ but $\langle C, s[x \mapsto [\![E]\!]s], h\rangle \not\models \varphi$ then it is the case that the premise is invalid.

If there is a $\Box$-trace following the edge, then by Definition 3.2.7, $\varphi = AG\psi$. Furthermore, by our previous invalidity result $\langle C,s,h\rangle \not\models AG\psi$, then by Definition 2.3.2 we know that there exists a configuration $\gamma_\neg$ such that $\langle C, s[x \mapsto [\![E]\!]s], h\rangle \rightsquigarrow^* \gamma_\neg$ and $\gamma_\neg \not\models \psi$. Finally, since by the operational semantics there is a single possible transition from $\langle x := E\ ; C, s, h\rangle$, namely $\langle x := E\ ; C, s, h\rangle \rightsquigarrow \langle C, s[x \mapsto [\![E]\!]s], h\rangle$, then every path starting from $\langle C, s[x \mapsto [\![E]\!]s], h\rangle$ is a subpath of a path starting from $\langle x := E\ ; C, s, h\rangle$. Consequently letting $\pi = \langle x := E\ ; C, s, h\rangle \rightsquigarrow^* \gamma_\neg$ and $\pi' = \langle C, s[x \mapsto [\![E]\!]s], h\rangle \rightsquigarrow^* \gamma_\neg$ then $\mathrm{length}(\pi') < \mathrm{length}(\pi)$.

If there is a $\diamond$-trace following the edge, then by Definition 3.2.7, $\varphi = EG\psi$. Furthermore, since by our previous invalidity result $\langle C, s[x \mapsto [\![E]\!]s], h\rangle \nvDash EG\psi$, then by Definition 2.3.2 we know that for all paths π starting from $\langle x := E\ ; C, s, h\rangle$ there exists a configuration $\gamma_\neg \in \pi$ such that $\gamma_\neg \nvDash \psi$. In other words, there is a finite tree κ' with root in $\langle x := E\ ; C, s, h\rangle$ whose all leaves $\gamma_\neg \nvDash \psi$. Finally, since by the operational semantics there is a single possible transition from $\langle x := E\ ; C, s, h\rangle$, namely $\langle x := E\ ; C, s, h\rangle \rightsquigarrow \langle C, s[x \mapsto [\![E]\!]s], h\rangle$, then every tree κ' with root in $\langle C, s[x \mapsto [\![E]\!]s], h\rangle$ is a subtree of a tree κ with root in $\langle x := E\ ; C, s, h\rangle$ and for all leaves $\gamma_\neg \in \kappa$ then $\gamma_\neg \in \kappa'$. Therefore $\mathrm{height}(\kappa') < \mathrm{height}(\kappa)$.

If there is a precondition trace following the edge, then since the precondition for both conclusion and premise is the same (modulo the substitution of variables $P[x'/x]$), then the inductive predicates $\Psi(\mathbf{E})$ in P for both conclusion and premise are defined by the same least approximant. Hence $\alpha = \beta$.

Soundness of Read

$$\frac{x = E' \wedge (P * E \mapsto E')[x'/x] \vdash C : \varphi}{P * E \mapsto E' \vdash (x := [E]\ ; C) : \circ\varphi}\ (\text{Read})$$

Assume the conclusion of the rule is invalid. Pick an arbitrary program state s, h such that $(s, h) \vDash P * E \mapsto E'$ but $\langle x := [E]\ ; C, s, h\rangle \nvDash \circ\varphi$. By Definition 2.3.2, if $\langle x := [E]\ ; C, s, h\rangle \nvDash \circ\varphi$ then there exists a configuration γ such that $\langle x := [E]\ ; C, s, h\rangle \rightsquigarrow \gamma$ and $\gamma \nvDash \varphi$. By the operational semantics of our programming language, since $[\![E]\!]s \in \mathsf{dom}(h)$, we know that $\langle x := [E]\ ; C, s, h\rangle \rightsquigarrow \langle C, s[x \mapsto h([\![E]\!]s]), h\rangle$ so $\gamma = \langle C, s[x \mapsto h([\![E]\!]s)], h\rangle$.

By construction we know that $(s[x \mapsto h([\![E]\!]s)], h) \vDash x = E'$ and by Lemma 2.2.7 and our assumption that $(s, h) \vDash P * E \mapsto E'$ we know that $(s[x \mapsto h([\![E]\!]s)], h) \vDash (P *$

$E \mapsto E')[x'/x]$. Consequently, since $(s[x \mapsto h(\llbracket E \rrbracket s)], h) \vDash x = E', (P * E \mapsto E')[x'/x]$ but $\langle C, s[x \mapsto h(\llbracket E \rrbracket s)], h\rangle \nvDash \varphi$ then it is the case that the premise is invalid.

If there is a $\square$-trace following the edge, then by Definition 3.2.7, $\varphi = AG\psi$. Furthermore, since by our previous invalidity result $\langle C, s[x \mapsto h(\llbracket E \rrbracket s)], h\rangle \nvDash AG\psi$, then by Definition 2.3.2 we know that there exists a configuration $\gamma_\neg$ such that $\langle C, s[x \mapsto h(\llbracket E \rrbracket s)], h\rangle \rightsquigarrow^* \gamma_\neg$ and $\gamma_\neg \nvDash \psi$. Finally, since by the operational semantics there is a single possible transition from $\langle x := [E]\ ; C, s, h\rangle$, namely $\langle x := [E]\ ; C, s, h\rangle \rightsquigarrow \langle C, s[x \mapsto h(\llbracket E \rrbracket s)], h\rangle$, then every path starting from $\langle C, s[x \mapsto h(\llbracket E \rrbracket s)], h\rangle$ is a subpath of a path starting from $\langle x := [E]\ ; C, s, h\rangle$. Consequently letting $\pi = \langle x := E\ ; C, s, h\rangle \rightsquigarrow^* \gamma_\neg$ and $\pi' = \langle C, s[x \mapsto \llbracket E \rrbracket s], h\rangle \rightsquigarrow^* \gamma_\neg$ then $\text{length}(\pi') < \text{length}(\pi)$.

If there is a $\diamond$-trace following the edge, then by Definition 3.2.7, $\varphi = EG\psi$. Furthermore, since by our previous invalidity result $\langle C, s[x \mapsto h(\llbracket E \rrbracket s)], h\rangle \nvDash EG\psi$, then by Definition 2.3.2 we know that for all paths π starting from $\langle C, s[x \mapsto \llbracket E \rrbracket s], h\rangle$ there exists a configuration $\gamma_\neg \in \pi$ such that $\gamma_\neg \nvDash \psi$. In other words, there is a finite tree κ' with root in $\langle C, s[x \mapsto \llbracket E \rrbracket s], h\rangle$ whose all leaves $\gamma_\neg \nvDash \psi$. Finally, since by the operational semantics there is a single possible transition from $\langle x := [E]\ ; C, s, h\rangle$, namely $\langle x := [E]\ ; C, s, h\rangle \rightsquigarrow \langle C, s[x \mapsto \llbracket E \rrbracket s], h\rangle$, then every tree with root in $\langle C, s[x \mapsto \llbracket E \rrbracket s], h\rangle$ is a subtree of a tree κ with root in $\langle x := [E]\ ; C, s, h\rangle$ and for all leaves $\gamma_\neg \in \kappa$ then $\gamma_\neg \in \kappa'$. Therefore $\text{height}(\kappa') < \text{height}(\kappa)$.

If there is a precondition trace following the edge then since the precondition for both conclusion and premise is the same (modulo the substitution of variables $P[x'/x]$), then the inductive predicates $\Psi(\mathbf{E})$ in P for both conclusion and premise are defined by the same least approximant. Hence $\alpha = \beta$.

Soundness of Write

$$\frac{P * E \mapsto E' \vdash C : \varphi}{P * E \mapsto - \vdash ([E] := E'\ ; C) : \bigcirc\varphi}\ \text{(Write)}$$

Assume the conclusion of the rule is invalid. Pick an arbitrary program state s,h such that $(s,h) \vDash P * E \mapsto -$ but $\langle [E] := E\ ; C,s,h\rangle \nvDash \bigcirc\varphi$. By Definition 2.3.2, if $\langle [E] := E\ ; C,s,h\rangle \nvDash \bigcirc\varphi$ then there exists a configuration γ such that $\langle [E] := E\ ; C,s,h\rangle \rightsquigarrow \gamma$ and $\gamma \nvDash \varphi$. By the operational semantics of our programming language, since $[\![E]\!]s \in \mathsf{dom}(h)$, then $\langle [E] := E'\ ; C,s,h\rangle \rightsquigarrow \langle C,s,h[[\![E]\!]s \mapsto [\![E']\!]s]\rangle$, hence we know that $\gamma = \langle C,s,h[[\![E]\!]s \mapsto [\![E']\!]s]\rangle$.

By construction we know that $(s,h[[\![E]\!]s \mapsto [\![E']\!]s]) \vDash E \mapsto E'$ and by our assumption that $(s,h) \vDash P * E \mapsto -$ we know that $(s,h[[\![E]\!]s \mapsto [\![E']\!]s]) \vDash (P * E \mapsto E')$. Consequently, since $(s,h[[\![E]\!]s \mapsto [\![E']\!]s]) \vDash P * E \mapsto E'$ but $\langle C,s,h[[\![E]\!]s \mapsto [\![E']\!]s])\rangle \nvDash \varphi$ then it is the case that the premise is invalid.

If there is a $\Box$-trace following the edge, then by Definition 3.2.7, $\varphi = AG\psi$. Furthermore, since by our previous invalidity result $\langle C,s,h[[\![E]\!]s \mapsto [\![E']\!]s])\rangle \nvDash AG\psi$, then by Definition 2.3.2 we know that there exists a configuration $\gamma_\neg$ such that $\langle C,s,h[[\![E]\!]s \mapsto [\![E']\!]s])\rangle \rightsquigarrow^* \gamma_\neg$ and $\gamma_\neg \nvDash \psi$. Finally, since by the operational semantics there is a single possible transition from $\langle [E] := E'\ ; C,s,h\rangle$, namely $\langle [E] := E'\ ; C,s,h\rangle \rightsquigarrow \langle C,s,h[[\![E]\!]s \mapsto [\![E']\!]s])\rangle$, then every path starting from $\langle C,s,h[[\![E]\!]s \mapsto [\![E']\!]s])\rangle$ is a subpath of a path starting from $\langle [E] := E';C,s,h\rangle$. Consequently letting $\pi = \langle x := E\ ; C,s,h\rangle \rightsquigarrow^* \gamma_\neg$ and $\pi' = \langle C,s[x \mapsto [\![E]\!]s],h\rangle \rightsquigarrow^* \gamma_\neg$ then $\text{length}(\pi') < \text{length}(\pi)$.

If there is a $\Diamond$-trace following the edge, then by Definition 3.2.7, $\varphi = EG\psi$. Furthermore, since by our previous result $\langle C,s,h[[\![E]\!]s \mapsto [\![E']\!]s])\rangle \nvDash EG\psi$, then by Definition 2.3.2 we know that for all paths π starting from $\langle C,s,h[[\![E]\!]s \mapsto [\![E']\!]s])\rangle$ there exists a configuration $\gamma_\neg \in \pi$ such that $\gamma_\neg \nvDash \psi$. In other words, there is a finite tree κ' with root in $\langle C,s,h[[\![E]\!]s \mapsto [\![E']\!]s])\rangle$ whose all leaves $\gamma_\neg \nvDash \psi$. Finally, since by the operational semantics there is a single possible transition from $\langle [E] := E'\ ; C,s,h\rangle$, namely $\langle [E] := E'\ ; C,s,h\rangle \rightsquigarrow \langle C,s,h[[\![E]\!]s \mapsto [\![E']\!]s])\rangle$, then every tree with root in $\langle C,s,h[[\![E]\!]s \mapsto [\![E']\!]s])\rangle$ is a subtree of a tree κ with root in $\langle [E] := E'\ ; C,s,h\rangle$ and for all leaves $\gamma_\neg \in \kappa$ then $\gamma_\neg \in \kappa'$. Therefore $\text{height}(\kappa') < \text{height}(\kappa)$.

If there is a precondition trace following the edge then since the precondition for both conclusion and premise is the same (modulo the separate conjunct $E \mapsto -$), then the inductive predicates $\Psi(\mathbf{E})$ in P for both conclusion and premise are defined by the same least approximant. Hence $\alpha = \beta$.

Soundness of If

$$\frac{B, P \vdash C_1 \;;\; C_3 : \varphi \qquad \neg B, P \vdash C_2 \;;\; C_3 : \varphi}{P \vdash (\mathbf{if}\, B\, \mathbf{then}\, C_1\, \mathbf{else}\, C_2\, \mathbf{fi} \;;\; C_3) : \bigcirc\varphi} \text{(If)}$$

Case $\bigcirc = \Box$.

Assume the conclusion of the rule is invalid. Pick an arbitrary program state s, h such that $(s,h) \vDash P$ but $\gamma = \langle \mathbf{if}\, B\, \mathbf{then}\, C_1\, \mathbf{else}\, C_2\, \mathbf{fi} \;;\; C_3, s, h\rangle \nvDash \Box\varphi$. By Definition 2.3.2, if $\gamma \nvDash \Box\varphi$ then there exists a configuration γ' such that $\gamma \rightsquigarrow \gamma'$ and $\gamma' \nvDash \varphi$. By the operational semantics of our programming language we know that either $\gamma' = \langle C_2 \;;\; C_3, s, h\rangle$ or $\gamma' = \langle C_1 \;;\; C_3, s, h\rangle$. We show the details of the foremost while omitting the latter due to their similarity.

Subcase $\gamma' = \langle C_1 \;;\; C_3, s, h\rangle$:

By our assumption we know that $(s,h) \vDash P$. Moreover, by the side condition of the operational semantics $[\![B]\!]s$ holds, then it is the case that $(s,h) \vDash B$. Consequently, since $(s,h) \vDash B, P$ but $\langle C_1 \;;\; C_3, s, h\rangle \nvDash \varphi$ then it is the case that the left-most premise is invalid.

If there is a $\Box$-trace following the edge, then by Definition 3.2.7, $\varphi = AG\psi$. Furthermore, since by our previous invalidity result $\gamma' \nvDash AG\psi$, then by Definition 2.3.2 we know that there exists a configuration $\gamma_\neg$ such that $\gamma' \rightsquigarrow^* \gamma_\neg$ and $\gamma_\neg \nvDash \psi$. Finally, since by the operational semantics $\gamma \rightsquigarrow \gamma'$, then every path starting from γ' is a subpath of a path starting from γ. Consequently letting $\pi = \gamma \rightsquigarrow^* \gamma_\neg$ and $\pi' = \gamma' \rightsquigarrow^* \gamma_\neg$ then $\text{length}(\pi') < \text{length}(\pi)$.

By the structure of the temporal formula in the conclusion ($\Box\varphi$), there cannot be a $\Diamond$-trace following the edge.

If there is a precondition trace $(\Psi(\mathbf{E}), \Psi'(\mathbf{E}))$ following the edge, where $(s,h) \vDash \Psi^\alpha(\mathbf{E})$ and $(s',h') \vDash \Psi'^\beta(\mathbf{E})$, then, since the precondition for both conclusion and premise is the same (modulo the addition of formula B to the pure part of the symbolic heap), then the inductive predicates $\Psi(\mathbf{E})$ and $\Psi'(\mathbf{E})$ in both conclusion and premise are interpreted by the same least approximant. Hence $\alpha = \beta$.

Subcase $\gamma' = \langle C_2 \,;\, C_3, s, h\rangle$:

Similar to above.

Case $\bigcirc = \Diamond$.

Assume the conclusion of the rule is invalid. Pick an arbitrary program state s,h such that $(s,h) \vDash P$ but $\gamma = \langle \textbf{if}\, B \,\textbf{then}\, C_1 \,\textbf{else}\, C_2 \,\textbf{fi} \,;\, C_3, s, h\rangle \nvDash \Diamond\varphi$. By Definition 2.3.2, if $\gamma \nvDash \Diamond\varphi$ then for all configurations γ' such that $\gamma \rightsquigarrow \gamma'$ then $\gamma' \nvDash \varphi$. By the operational semantics of our programming language we know that there are two possible transitions: $\gamma' = \langle C_2 \,;\, C_3, s, h\rangle$ and $\gamma' = \langle C_1 \,;\, C_3, s, h\rangle$. Consequently, since $(s,h) \vDash P$ but $\langle C_2 \,;\, C_3, s, h\rangle \nvDash \varphi$ and $\langle C_1 \,;\, C_3, s, h\rangle \nvDash \varphi$ then it is the case that either one of the premises is invalid, depending on which of $[\![B]\!]s$ or $[\![\neg B]\!]s$ holds.

By the structure of the temporal formula in the conclusion ($\Diamond\varphi$), there cannot be a $\Box$-trace following the edge.

If there is a $\Diamond$-trace following the left-most edge, then by Definition 3.2.7, $\varphi = EG\psi$. Furthermore, since by our previous invalidity result $\langle C_1 \,;\, C_3, s, h\rangle \nvDash EG\psi$, then by Definition 2.3.2 we know that for all paths π starting from $\langle C_1 \,;\, C_3, s, h\rangle$ there exists a configuration $\gamma_\neg \in \pi$ such that $\gamma_\neg \nvDash \psi$. In other words, there is a finite tree κ' with root in $\langle C_1 \,;\, C_3, s, h\rangle$ whose all leaves $\gamma_\neg \nvDash \psi$. Finally, since by the operational semantics $\langle \textbf{if}\, B \,\textbf{then}\, C_1 \,\textbf{else}\, C_2 \,\textbf{fi} \,;\, C_3, s, h\rangle \rightsquigarrow \langle C_1 \,;\, C_3, s, h\rangle$, then, by definition, every tree κ' with root in $\langle C_1 \,;\, C_3, s, h\rangle$ is a subtree of a tree κ with root in $\langle \textbf{if}\, B \,\textbf{then}\, C_1 \,\textbf{else}\, C_2 \,\textbf{fi} \,;\, C_3, s, h\rangle$. Hence height$(\kappa') \leq$ height(κ) and for all leaves $\gamma_\neg \in \kappa'$ then $\gamma_\neg \in \kappa$.

The case for a $\Diamond$-trace following the right-most edge is similar, accounting for configuration $\langle C_2 \,;\, C_3, s, h\rangle$ in place of $\langle C_1 \,;\, C_3, s, h\rangle$.

If there is a precondition trace $(\Psi(\mathbf{E}), \Psi'(\mathbf{E}))$ following the left-most edge, where $(s,h) \vDash \Psi^\alpha(\mathbf{E})$ and $(s',h') \vDash \Psi'^\beta(\mathbf{E})$, then, since the precondition for both conclusion and premise is the same (modulo the addition of formula B to the pure part of the symbolic heap), then the inductive predicates $\Psi(\mathbf{E})$ and $\Psi'(\mathbf{E})$ in both conclusion and premise are interpreted by the same least approximant. Hence $\alpha = \beta$. The same argument applies to the case where the precondition trace follows the right-most edge, with the only difference of the pure formula $\neg B$, in place of B added to the pure part of the symbolic heap precondition formula.

Soundness of Wh

$$\frac{B,P \vdash (C_1 \ ; \textbf{while}\, B\, \textbf{do}\, C_1\ \textbf{od} \ ; C_2) : \varphi \qquad \neg B, P \vdash C_2 : \varphi}{P \vdash (\textbf{while}\, B\, \textbf{do}\, C_1\ \textbf{od} \ ; C_2) : \bigcirc\varphi} \ (\text{Wh})$$

Case $\bigcirc = \Box$.

Assume the conclusion of the rule is invalid. Pick an arbitrary program state s,h such that $(s,h) \vDash P$ but $\langle \textbf{while}\, B\, \textbf{do}\, C_1\ \textbf{od} \ ; C_2, s, h\rangle \nvDash \Box\varphi$. By Definition 2.3.2, if $\langle \textbf{while}\, B\, \textbf{do}\, C_1\ \textbf{od} \ ; C_2, s, h\rangle \nvDash \Box\varphi$ then there exists a configuration γ such that $\langle \textbf{while}\, B\, \textbf{do}\, C_1\ \textbf{od} \ ; C_2, s, h\rangle \rightsquigarrow \gamma$ and $\gamma \nvDash \varphi$. By the operational semantics of our programming language we know that either $\gamma = \langle (C_1 \ ; \textbf{while}\, B\, \textbf{do}\, C_1\ \textbf{od} \ ; C_2), s, h\rangle$ or $\gamma = \langle C_2, s, h\rangle$.

Subcase $\gamma = \langle (C_1 \ ; \textbf{while}\, B\, \textbf{do}\, C_1\ \textbf{od} \ ; C_2), s, h\rangle$:

By our assumption we know that $(s,h) \vDash P$. Moreover, by the side condition of the operational semantics $[\![B]\!]s$ holds, then it is the case that $(s,h) \vDash B,P$. Consequently, since $(s,h) \vDash B,P$ but $\langle (C_1 \ ; \textbf{while}\, B\, \textbf{do}\, C_1\ \textbf{od} C_2 \ ;), s, h\rangle \nvDash \varphi$ then it is the case that the left-most premise is invalid.

If there is a $\Box$-trace following the edge, then by Definition 3.2.7, $\varphi = AG\psi$. Furthermore, since by our previous result $\langle (C_1 \ ; \textbf{while}\, B\, \textbf{do}\, C_1\ \textbf{od} \ ; C_2), s, h\rangle \nvDash AG\psi$, then by Definition 2.3.2 we know that there exists a configuration $\gamma_\neg$ such that $\langle (C_1 \ ; \textbf{while}\, B\, \textbf{do}\, C_1\ \textbf{od} \ ; C_2), s, h\rangle \rightsquigarrow^* \gamma_\neg$ and $\gamma_\neg \nvDash \psi$. Finally, since by the operational semantics $\langle \textbf{while}\, B\, \textbf{do}\, C_1\ \textbf{od} \ ; C_2, s, h\rangle \rightsquigarrow \langle (C_1 \ ; \textbf{while}\, B\, \textbf{do}\, C_1\ \textbf{od} \ ; C_2), s, h\rangle$, then every path π' starting from $\langle (C_1 \ ; \textbf{while}\, B\, \textbf{do}\, C_1\ \textbf{od} \ ; C_2), s, h\rangle$ is a subpath of a path π starting from $\langle \textbf{while}\, B\, \textbf{do}\, C_1\ \textbf{od} \ ; C_2, s, h\rangle$. Consequently $\text{length}(\pi') < \text{length}(\pi)$.

By the structure of the temporal formula in the conclusion ($\Box\varphi$), there cannot be a $\Diamond$-trace following the edge.

If there is a precondition trace $(\Psi(\mathbf{E}), \Psi'(\mathbf{E}))$ following the edge, where $(s,h) \vDash \Psi^\alpha(\mathbf{E})$ and $(s',h') \vDash \Psi'^\beta(\mathbf{E})$, then, since the precondition for both conclusion and premise is the same (modulo the addition of formula B to the pure part of the symbolic heap), then the inductive predicates $\Psi(\mathbf{E})$ and $\Psi'(\mathbf{E})$ in both conclusion and premise are interpreted by the same least approximant. Hence $\alpha = \beta$.

Subcase $\gamma = \langle C_2, s, h\rangle$:

By our assumption we know that $(s,h) \models P$. Moreover, by the side condition of the operational semantics $\neg[\![B]\!]s$ holds, then it is the case that $(s,h) \models \neg B, P$. Consequently, since $(s,h) \models \neg B, P$ but $\langle C_2, s, h\rangle \not\models \varphi$ then it is the case that the right-most premise is invalid.

If there is a $\Box$-trace following the edge, then by Definition 3.2.7, $\varphi = AG\psi$. Furthermore, since by our previous invalidity result $\langle C_2, s, h\rangle \not\models AG\psi$, then by Definition 2.3.2 we know that there exists a configuration $\gamma_\neg$ such that $\langle C_2, s, h\rangle \rightsquigarrow^* \gamma_\neg$ and $\gamma_\neg \not\models \psi$. Finally, since by the operational semantics $\langle \textbf{while}\, B\, \textbf{do}\, C_1\, \textbf{od}\, ; C_2, s, h\rangle \rightsquigarrow \langle C_2, s, h\rangle$, then every path starting from $\langle C_2, s, h\rangle$ is a subpath of a path starting from $\langle \textbf{while}\, B\, \textbf{do}\, C_1\, \textbf{od}\, ; C_2, s, h\rangle$. Consequently letting $\pi = \langle \textbf{while}\, B\, \textbf{do}\, C_1\, \textbf{od}\, ; C_2, s, h\rangle \rightsquigarrow^* \gamma_\neg$ and $\pi' = \langle C_2, s, h\rangle \rightsquigarrow^* \gamma_\neg$ then $\text{length}(\pi') < \text{length}(\pi)$.

By the structure of the temporal formula in the conclusion ($\Box\varphi$), there cannot be a $\Diamond$-trace following the edge.

If there is a precondition trace $(\Psi(\mathbf{E}), \Psi'(\mathbf{E}))$ following the edge, where $(s,h) \models \Psi^{\alpha}(\mathbf{E})$ and $(s',h') \models \Psi'^{\beta}(\mathbf{E})$, then, since the precondition for both conclusion and premise is the same (modulo the addition of formula $\neg B$ to the pure part of the symbolic heap), then the inductive predicates $\Psi(\mathbf{E})$ and $\Psi'(\mathbf{E})$ in both conclusion and premise are interpreted by the same least approximant. Hence $\alpha = \beta$.

Case $\bigcirc = \Diamond$.

Assume the conclusion of the rule is invalid. Pick an arbitrary program state s,h such that $(s,h) \models P$ but $\langle \textbf{while}\, B\, \textbf{do}\, C_1\, \textbf{od}\, ; C_2, s, h\rangle \not\models \Diamond\varphi$. By Definition 2.3.2, if $\langle \textbf{while}\, B\, \textbf{do}\, C_1\, \textbf{od}\, ; C_2, s, h\rangle \not\models \Diamond\varphi$ then for all configuration γ such that $\langle \textbf{while}\, B\, \textbf{do}\, C_1\, \textbf{od}\, ; C_2, s, h\rangle \rightsquigarrow \gamma$ then $\gamma \not\models \varphi$. By the operational semantics of our programming language we know that there are two possible transitions: $\gamma = \langle (C_1\, ; \textbf{while}\, B\, \textbf{do}\, C_1\, \textbf{od}\, ; C_2), s, h\rangle$ and $\gamma' = \langle C_2, s, h\rangle$. Consequently, since $(s,h) \models P$

but $\langle C_2 ; C_3, s, h\rangle \not\models \varphi$ and $\langle C_1 ; C_3, s, h\rangle \not\models \varphi$ then it is the case that either one of the premises is invalid, depending on which of $[\![B]\!]s$ or $[\![\neg B]\!]s$ holds.

By the structure of the temporal formula in the conclusion ($\Diamond\varphi$), there cannot be a $\Box$-trace following the edge.

If there is a $\Diamond$-trace following the left-most edge, then by Definition 3.2.7, $\varphi = EG\psi$. Furthermore, since by our previous invalidity result $\gamma' \not\models EG\psi$, then by Definition 2.3.2 we know that for all paths π' starting from γ' there exists a configuration $\gamma_\neg \in \pi'$ such that $\gamma_\neg \not\models \psi$. In other words, there is a finite tree κ' with root in γ' whose all leaves $\gamma_\neg \not\models \psi$. Finally, since by the operational semantics $\gamma \rightsquigarrow \gamma'$, then, by definition, every tree with root in γ' is a subtree of the tree κ with root in γ. Hence $\text{height}(\kappa') \le \text{height}(\kappa)$ and for all leaves $\gamma_\neg \in \kappa'$ then $\gamma_\neg \in \kappa$.

If there is a $\Diamond$-trace following the rigth-most edge, then by Definition 3.2.7, $\varphi = EG\psi$. Furthermore, since by our previous invalidity result $\gamma' \not\models EG\psi$, then by Definition 2.3.2 we know that for all paths π starting from γ' there exists a configuration $\gamma_\neg \in \pi$ such that $\gamma_\neg \not\models \psi$. In other words, there is a finite tree κ' with root in γ' whose all leaves $\gamma_\neg \not\models \psi$. Finally, since by the operational semantics $\gamma \rightsquigarrow \gamma'$, then, by definition, every tree with root in γ' is a subtree of the tree κ with root in γ. Hence $\text{height}(\kappa') \le \text{height}(\kappa)$ and for all leaves $\gamma_\neg \in \kappa'$ then $\gamma_\neg \in \kappa$.

If there is a precondition trace $(\Psi(\mathbf{E}), \Psi'(\mathbf{E}))$ following the left-most edge, where $(s,h) \models \Psi^{\alpha}(\mathbf{E})$ and $(s',h') \models \Psi'^{\beta}(\mathbf{E})$, then, since the precondition for both conclusion and premise is the same (modulo the addition of formula B to the pure part of the symbolic heap), then the inductive predicates $\Psi(\mathbf{E})$ and $\Psi'(\mathbf{E})$ in both conclusion and premise are interpreted by the same least approximant. Hence $\alpha = \beta$. The same argument applies to the case where the precondition trace follows the right-most edge, with the only difference of the pure formula $\neg B$, in place of B added to the pure part of the symbolic heap precondition formula.

Soundness of Free

$$\frac{P \vdash C : \varphi}{P * E \mapsto - \vdash (\mathbf{free}(E) ; C) : \circ\varphi} \text{ (Free)}$$

Assume the conclusion of the rule is invalid. Pick an arbitrary program state s,h such that $(s,h) \vDash P * E \mapsto -$ but $\langle \texttt{free}(E)\,; C,s,h\rangle \nvDash \bigcirc\varphi$. By Definition 2.3.2, if $\langle \texttt{free}(E)\,; C,s,h\rangle \nvDash \bigcirc\varphi$ then there exists a configuration γ such that $\langle \texttt{free}(E)\,; C,s,h\rangle \rightsquigarrow \gamma$ and $\gamma \nvDash \varphi$. By the operational semantics of our programming language we know that $\gamma = \langle C,s,h|\mathsf{dom}(h) \setminus \{[\![E]\!]s\}\rangle$.

By our assumption $(s,h) \vDash P * E \mapsto -$ we know we can split h into h' and h'' so that $(s,h') \vDash P$ and separately $(s,h'') \vDash E \mapsto -$. Then, by construction we know that $h' = h|\mathsf{dom}(h) \setminus \{[\![E]\!]s\}$. Consequently $(s,h|\mathsf{dom}(h) \setminus \{[\![E]\!]s\}) \vDash P$. Finally, since $(s,h|\mathsf{dom}(h) \setminus \{[\![E]\!]s\}) \vDash P$ but $\langle C,s,h|\mathsf{dom}(h) \setminus \{[\![E]\!]s\}\rangle \nvDash \varphi$ then it is the case that the premise is invalid.

If there is a $\Box$-trace following the edge, then by Definition 3.2.7, $\varphi = AG\psi$. Furthermore, since by our previous invalidity result $\langle C,s,h|\mathsf{dom}(h) \setminus \{[\![E]\!]s\}\rangle \nvDash AG\psi$, then by Definition 2.3.2 we know that there exists a configuration $\gamma_\neg$ such that $\langle C,s,h|\mathsf{dom}(h) \setminus \{[\![E]\!]s\}\rangle \rightsquigarrow^* \gamma_\neg$ and $\gamma_\neg \nvDash \psi$. Finally, since by the operational semantics there is a single possible transition from $\langle \texttt{free}(E)\,; C,s,h\rangle$, namely $\langle \texttt{free}(E)\,; C,s,h\rangle \rightsquigarrow \langle C,s,h|\mathsf{dom}(h) \setminus \{[\![E]\!]s\}\rangle$, then every path starting from $\langle C,s,h|\mathsf{dom}(h) \setminus \{[\![E]\!]s\}\rangle$ is a subpath of a path starting from $\langle \texttt{free}(E);C,s,h\rangle$. Consequently letting $\pi = \langle \texttt{free}(E);C,s,h\rangle \rightsquigarrow^* \gamma_\neg$ and $\pi' = \langle C,s,h|\mathsf{dom}(h) \setminus \{[\![E]\!]s\}\rangle \rightsquigarrow^* \gamma_\neg$ then $\text{length}(\pi') < \text{length}(\pi)$.

If there is a $\Diamond$-trace following the edge, then by Definition 3.2.7, $\varphi = EG\psi$. Furthermore, since by our previous invalidity result $\langle C,s,h|\mathsf{dom}(h) \setminus \{[\![E]\!]s\}\rangle \nvDash EG\psi$, then by Definition 2.3.2 we know that for all paths π' starting from $\langle C,s,h|\mathsf{dom}(h) \setminus \{[\![E]\!]s\}\rangle$ there exists a configuration $\gamma_\neg \in \pi'$ such that $\gamma_\neg \nvDash \psi$. In other words, there is a finite tree κ' with root in $\langle C,s,h|\mathsf{dom}(h) \setminus \{[\![E]\!]s\}\rangle$ whose all leaves $\gamma_\neg \nvDash \psi$. Finally, since by the operational semantics there is a single possible transition from $\langle \texttt{free}(E)\,; C,s,h\rangle$, namely $\langle \texttt{free}(E)\,; C,s,h\rangle \rightsquigarrow \langle C,s,h|\mathsf{dom}(h) \setminus \{[\![E]\!]s\}\rangle$, then every tree κ' with root in $\langle C,s,h|\mathsf{dom}(h) \setminus \{[\![E]\!]s\}\rangle$ is a subtree of a tree κ with root in $\langle \texttt{free}(E)\,; C,s,h\rangle$ and for all leaves $\gamma_\neg \in \kappa$ then $\gamma_\neg \in \kappa'$. Therefore $\text{height}(\kappa') < \text{height}(\kappa)$.

If there is a precondition trace following the edge then since the precondition for both conclusion and premise is the same (modulo the removal of separate conjunct $E \mapsto -$), then the inductive predicates $\Psi(\mathbf{E})$ in P for both conclusion and premise are defined by the same least approximant. Hence $\alpha = \beta$.

Soundness of Alloc$\diamond$

$$\frac{P[x'/x] * x \mapsto v \vdash C : \varphi \quad v \in \mathrm{Val}}{P \vdash (x := alloc()\ ; C) : \diamond\varphi} \ (\mathrm{Alloc}\diamond)$$

Assume the conclusion of the rule is invalid. Pick an arbitrary program state s,h such that $(s,h) \vDash P$ but $\langle x := alloc()\ ; C, s, h\rangle \nvDash$. By Definition 2.3.2, if $\langle x := alloc()\ ; C, s, h\rangle \nvDash \diamond\varphi$ then for all configurations γ such that $\langle x := alloc()\ ; C, s, h\rangle \rightsquigarrow \gamma$ it is the case that $\gamma \nvDash \varphi$. By the operational semantics of our programming language we know that every such γ takes the shape of $\langle C, s[x \mapsto l], h[l \mapsto v]\rangle$ where $l \notin \mathsf{dom}(h)$ (where only the values of l and v change in each configuration).

Because $l \notin \mathsf{dom}(h)$ we can split γ_h into two subheaps so that $\gamma_h = h \uplus l \mapsto v$ for a given l and v. By construction we know that $(s[x \mapsto l], l \mapsto v) \vDash x \mapsto v$. Moreover, by our assumption that $(s,h \vDash P)$ and Lemma 2.2.7 we know that $(s[x \mapsto l], h) \vDash P[x'/x]$. Consequently, since $h \uplus l \mapsto v$ is defined, then we know that $(s[x \mapsto l], h[l \mapsto v]) \vDash P[x', x] * x \mapsto v$. Finally, since $(s[x \mapsto l], h[l \mapsto v) \vDash P[x'/x] * x \mapsto v$ but $\langle C, (s[x \mapsto l], h[l \mapsto v])\rangle \nvDash \varphi$ then it is the case that the premise is invalid.

By the structure of the temporal formula in the conclusion ($\diamond\varphi$), there cannot be a $\square$-trace following the edge.

If there is a $\diamond$-trace following the edge, then by Definition 3.2.7, $\varphi = EG\psi$. Furthermore, since by our previous invalidity result $\langle C, (s[x \mapsto l], h[l \mapsto v])\rangle \nvDash EG\psi$, then by Definition 2.3.2 we know that for all paths π' starting from $\langle C, (s[x \mapsto l], h[l \mapsto v])\rangle$ there exists a configuration $\gamma_\neg \in \pi'$ such that $\gamma_\neg \nvDash \psi$. In other words, there is a finite tree κ' with root in $\langle C, (s[x \mapsto l], h[l \mapsto v])\rangle$ whose all leaves $\gamma_\neg \nvDash \psi$. Finally, since by the operational semantics there is a single possible transition from $\langle x := alloc()\ ; C, s, h\rangle$, namely $\langle x := alloc()\ ; C, s, h\rangle \rightsquigarrow \langle C, (s[x \mapsto l], h[l \mapsto v])\rangle$, then every tree κ' with root in $\langle C, (s[x \mapsto l], h[l \mapsto v])\rangle$ is a subtree of a tree κ with root in $\langle x := alloc()\ ; C, s, h\rangle$ and for all leaves $\gamma_\neg \in \kappa$ then $\gamma_\neg \in \kappa'$. Hence height$(\kappa') \leq$ height(κ).

If there is a precondition trace following the edge then since the precondition for both conclusion and premise is the same (modulo the substitution of variables $P[x'/x]$ and the addition of separate conjunct $x \mapsto v$), then the inductive predicates $\Psi(\mathbf{E})$ in P for both conclusion and premise are defined by the same least approximant. Hence $\alpha = \beta$.

Soundness of Alloc□

$$\frac{P[x'/x] * x \mapsto v \vdash C : \varphi}{P \vdash (x := alloc()\ ; C) : \Box\varphi}\ v \text{ fresh (Alloc}\Box)$$

Assume the conclusion of the rule is invalid. Pick an arbitrary program state s, h such that $(s, h) \vDash P$ but $\langle x := alloc()\ ; C, s, h\rangle \nvDash \Box\varphi$. By Definition 2.3.2, if $\langle x := alloc()\ ; C, s, h\rangle \nvDash \Box\varphi$ then there exists a configuration γ such that $\langle x := alloc()\ ; C, s, h\rangle \rightsquigarrow \gamma$ and $\gamma \nvDash \varphi$. By the operational semantics of our programming language we know that $\gamma = \langle C, s[x \mapsto l], h[l \mapsto v]\rangle$ where $l \notin \mathsf{dom}(h)$.

Because $l \notin \mathsf{dom}(h)$ we can split γ_h into two subheaps so that $\gamma_h = h \uplus l \mapsto v$. By construction we know that $(s[x \mapsto l], l \mapsto v) \vDash x \mapsto v$. Moreover, by our assumption that $(s, h \vDash P)$ and Lemma 2.2.7 we know that $(s[x \mapsto l], h) \vDash P[x'/x]$. Consequently, since $h \uplus l \mapsto v$ is defined, then we know that $(s[x \mapsto l], h[l \mapsto v]) \vDash P[x', x] * x \mapsto v$. Finally, since $(s[x \mapsto l], h[l \mapsto v) \vDash P[x'/x] * x \mapsto v$ but $\langle C, (s[x \mapsto l], h[l \mapsto v])\rangle \nvDash \varphi$ then it is the case that the premise is invalid.

If there is a □-trace following the edge, then by Definition 3.2.7, $\varphi = AG\psi$. Furthermore, since by our previous invalidity result $\langle C, (s[x \mapsto l], h[l \mapsto v])\rangle \nvDash AG\psi$, then by Definition 2.3.2 we know that there exists a configuration $\gamma_\neg$ such that $\langle C, (s[x \mapsto l], h[l \mapsto v])\rangle \rightsquigarrow^* \gamma_\neg$ and $\gamma_\neg \nvDash \psi$. Finally, since by the operational semantics there is a single possible transition from $\langle x := alloc()\ ; C, s, h\rangle$, namely $\langle x := alloc()\ ; C, s, h\rangle \rightsquigarrow \langle C, (s[x \mapsto l], h[l \mapsto v])\rangle$, then every path starting from $\langle C, (s[x \mapsto l], h[l \mapsto v])\rangle$ is a subpath of a path starting from $\langle x := alloc(); C, s, h\rangle$. Consequently letting $\pi = \langle x := alloc(); C, s, h\rangle \rightsquigarrow^* \gamma_\neg$ and $\pi' = \langle C, (s[x \mapsto l], h[l \mapsto v])\rangle$

$\rightsquigarrow^* \gamma_\neg$ then length$(\pi') <$ length(π).

By the structure of the temporal formula in the conclusion ($\Diamond\varphi$), there cannot be a $\Box$-trace following the edge.

If there is a precondition trace following the edge then since the precondition for both conclusion and premise is the same (modulo the substitution of variables $P[x'/x]$ and the addition of separate conjunct $x \mapsto v$), then the inductive predicates $\Psi(\mathbf{E})$ in P for both conclusion and premise are defined by the same least approximant. Hence $\alpha = \beta$.

Soundness of If*◇1

$$\frac{P \vdash C_1\ ;C_3 : \varphi}{P \vdash (\textbf{if} * \textbf{then}\, C_1\ \textbf{else}\, C_2\ \textbf{fi}\ ; C_3) : \Diamond\varphi}\ (\text{If}^* \Diamond 1)$$

Assume the conclusion of the rule is invalid. Pick an arbitrary program state s,h such that $(s,h) \vDash P$ but $\gamma = \langle \textbf{if} * \textbf{then}\, C_1\ \textbf{else}\, C_2\ \textbf{fi}\ ; C_3, s, h\rangle \nvDash \Diamond\varphi$. By Definition 2.3.2, if $\gamma \nvDash \Diamond\varphi$ then for all configurations γ' such that $\gamma \rightsquigarrow \gamma'$ it is the case that $\gamma' \nvDash \varphi$. By the operational semantics of our programming language we know that there are two possible transitions from the current configuration $\gamma \rightsquigarrow \langle C_1\ ; C_3, s, h\rangle$ and $\gamma \rightsquigarrow \langle C_2\ ; C_3, s, h\rangle$. Hence, we know that $\langle C_1\ ; C_3, s, h\rangle \nvDash \varphi$ and $\langle C_2\ ; C_3, s, h\rangle \nvDash \varphi$. Hence, since by our assumption $(s,h) \vDash P$ but $\langle C_1\ ; C_3, s, h\rangle \nvDash \varphi$ then it is the case that the premise is invalid.

Given the structure of the temporal formula $\varphi / \Diamond\,\varphi$, there cannot be a $\Box$-trace following the edge.

If there is a $\Diamond$-trace following the edge, then by Definition 3.2.7, $\varphi = EG\psi$. Furthermore, since by our previous invalidity result $\langle C_1\ ; C_3, s, h\rangle \nvDash EG\psi$, then by Definition 2.3.2 we know that for all paths π' starting from $\langle C_1\ ; C_3, s, h\rangle$ there exists a configuration $\gamma_\neg \in \pi$ such that $\gamma_\neg \nvDash \psi$. In other words, there is a finite tree κ' with root in $\langle C_1\ ; C_3, s, h\rangle$ whose all leaves $\gamma_\neg \nvDash \psi$. Finally, since by the operational semantics $\langle \textbf{if} * \textbf{then}\, C_1\ \textbf{else}\, C_2\ \textbf{fi}\ ; C_3, s, h\rangle \rightsquigarrow \langle C_1\ ; C_3, s, h\rangle$, then, by definition, every tree κ' with root in $\langle C_1\ ; C_3, s, h\rangle$ is a subtree of the tree κ with root in $\langle \textbf{if} * \textbf{then}\, C_1\ \textbf{else}\, C_2\ \textbf{fi}\ ; C_3, s, h\rangle$. Hence height$(\kappa') \leq$ height(κ) and for all leaves

$\gamma_{\neg} \in \kappa'$ then $\gamma_{\neg} \in \kappa$.

If there is a precondition trace $(\Psi(\mathbf{E}), \Psi'(\mathbf{E}))$ following the edge, where $(s,h) \vDash \Psi^{\alpha}(\mathbf{E})$ and $(s',h') \vDash \Psi'^{\beta}(\mathbf{E})$, then, since the precondition for both conclusion and premise is the same, then the inductive predicates $\Psi(\mathbf{E})$ and $\Psi'(\mathbf{E})$ in both conclusion and premise are interpreted by the same least approximant. Hence $\alpha = \beta$.

Soundness of If*◇2

$$\frac{P \vdash C_2\ ;C_3 : \varphi}{P \vdash (\textbf{if} * \textbf{then}\, C_1\, \textbf{else}\, C_2\, \textbf{fi}\ ; C_3) : \Diamond\varphi}\ (\text{If* }\Diamond 2)$$

Similar to above, accounting for configuration $\langle C_2\ ;C_3, s, h\rangle$ in place of $\langle C_1\ ;C_3, s, h\rangle$.

Soundness of If*□

$$\frac{P \vdash C_1\ ;C_3 : \varphi \quad P \vdash C_2\ ;C_3 : \varphi}{P \vdash (\textbf{if} * \textbf{then}\, C_1\, \textbf{else}\, C_2\, \textbf{fi}\ ; C_3) : \Box\varphi}\ (\text{If*}\Box)$$

Assume the conclusion of the rule is invalid. Pick an arbitrary program state s,h such that the precondition is satisfied $(s,h) \vDash P$ but the program configuration $\gamma = \langle \textbf{if} * \textbf{then}\, C_1\, \textbf{else}\, C_2\, \textbf{fi}\ ; C_3, s, h\rangle$ does not satisfy the temporal property $\Box\varphi$. By Definition 2.3.2, if $\gamma \nvDash \Box\varphi$ then there exists a configuration γ' such that $\gamma \rightsquigarrow \gamma'$ and $\gamma' \nvDash \varphi$. By the operational semantics of our programming language we know that there are two possible transitions from the current configuration $\gamma' = \langle C_1\ ;C_3, s, h\rangle$ and $\gamma' = \langle C_2\ ;C_3, s, h\rangle$. Hence, either $\langle C_1\ ;C_3, s, h\rangle \nvDash \varphi$ or $\langle C_2\ ;C_3, s, h\rangle \nvDash \varphi$. We show the details of the foremost, omitting the latter due to uts similarity.

Case $\gamma' = \langle C_1\ ;C_3, s, h\rangle \nvDash \varphi$:

By our assumption $(s,h) \vDash P$ and invalidity result $\gamma' \nvDash \varphi$ then it is the case that the left-most premise is invalid.

If there is a □-trace following the edge, then by Definition 3.2.7, $\varphi = AG\psi$. Furthermore, since by our previous invalidity result $\gamma' \nvDash AG\psi$, then by Definition 2.3.2 we know that there exists a configuration $\gamma_{\neg}$ such that $\gamma' \rightsquigarrow^{*} \gamma_{\neg}$ and $\gamma_{\neg} \nvDash \psi$. Finally, since by the operational semantics $\gamma \rightsquigarrow \gamma'$, then every path π' starting from γ' is a subpath of a path π starting from γ. Consequently length$(\pi') <$ length(π).

Given the structure of the temporal formula $\varphi / \Box\varphi$, there cannot be a $\Diamond$-trace following the edge.

If there is a precondition trace $(\Psi(\mathbf{E}), \Psi'(\mathbf{E}))$ following the edge, where $(s,h) \vDash \Psi^{\alpha}(\mathbf{E})$ and $(s',h') \vDash \Psi'^{\beta}(\mathbf{E})$, then, since the precondition for both conclusion and premise is the same, then the inductive predicates $\Psi(\mathbf{E})$ and $\Psi'(\mathbf{E})$ in both conclusion and premise are interpreted by the same least approximant. Hence $\alpha = \beta$.

Case $\gamma' = \langle C_2 \,;\, C_3, s, h\rangle \nvDash \varphi$:

Similar to above, accounting for configuration $\langle C_2 \,;\, C_3, s, h\rangle$ in place of $\langle C_1 \,;\, C_3, s, h\rangle$.

Soundness of Wh*□

$$\frac{P \vdash (C_1 \,;\, \textbf{while}\, B\, \textbf{do}\, C_1\, \textbf{od} \,;\, C_2) : \varphi \qquad P \vdash C_2 : \varphi}{P \vdash (\textbf{while}\, B\, \textbf{do}\, C_1\, \textbf{od} \,;\, C_2) : \Box\varphi} \;(\text{Wh}^*\Box)$$

Assume the conclusion of the rule is invalid. Pick an arbitrary program state s,h such that the precondition is satisfied $(s,h) \vDash P$ but the program configuration $\langle \textbf{while}\, B\, \textbf{do}\, C_1\, \textbf{od} \,;\, C_2, s, h\rangle$ does not satisfy the temporal property $\Box\varphi$. By Definition 2.3.2, if $\langle \textbf{while}\, B\, \textbf{do}\, C_1\, \textbf{od} \,;\, C_2, s, h\rangle \nvDash \Box\varphi$ then there exists a configuration γ such that $\langle \textbf{while}\, B\, \textbf{do}\, C_1\, \textbf{od} \,;\, C_2, s, h\rangle \rightsquigarrow \gamma$ and $\gamma \nvDash \varphi$. By the operational semantics of our programming language we know that either $\gamma = \langle (C_1 \,;\, \textbf{while}\, B\, \textbf{do}\, C_1\, \textbf{od} \,;\, C_2), s, h\rangle$ or $\gamma = \langle C_2, s, h\rangle$.

Case $\gamma = \langle (C_1 \,;\, \textbf{while}\, B\, \textbf{do}\, C_1\, \textbf{od} \,;\, C_2), s, h\rangle$:

By our assumption we know that $(s,h) \vDash P$ but since $\gamma \nvDash \varphi$ then it is the case that the left-most premise is invalid.

If there is a $\Box$-trace following the edge, then by Definition 3.2.7, $\varphi = AG\psi$. Furthermore, since by our previous result $\langle (C_1 \,;\, \textbf{while} * \textbf{do}\, C_1\, \textbf{od} \,;\, C_2), s, h\rangle \nvDash AG\psi$, then by Definition 2.3.2 we know that there exists a configuration $\gamma_{\neg}$ such that $\langle (C_1 \,;\, \textbf{while} * \textbf{do}\, C_1\, \textbf{od} \,;\, C_2), s, h\rangle \rightsquigarrow^* \gamma_{\neg}$ and $\gamma_{\neg} \nvDash \psi$. Finally, since by the operational semantics $\langle \textbf{while} * \textbf{do}\, C_1\, \textbf{od}\, ; C_2, s, h\rangle \rightsquigarrow \langle (C_1\, ; \textbf{while} * \textbf{do}\, C_1\, \textbf{od}\, ; C_2), s, h\rangle$, then every path π' starting from $\langle (C_1\, ; \textbf{while} * \textbf{do}\, C_1\, \textbf{od}\, ; C_2), s, h\rangle$ is a subpath of a path π starting from $\langle \textbf{while} * \textbf{do}\, C_1\, \textbf{od}\, ; C_2, s, h\rangle$. Consequently $\text{length}(\pi') < \text{length}(\pi)$.

Given the structure of the temporal formula $\varphi / \Box\, \varphi$, there cannot be a $\Diamond$-trace following the edge.

If there is a precondition trace $(\Psi(\mathbf{E}), \Psi'(\mathbf{E}))$ following the edge, where $(s,h) \vDash \Psi^{\alpha}(\mathbf{E})$ and $(s',h') \vDash \Psi'^{\beta}(\mathbf{E})$, then, since the precondition for both conclusion and premise is the same, then the inductive predicates $\Psi(\mathbf{E})$ and $\Psi'(\mathbf{E})$ in both conclusion and premise are interpreted by the same least approximant. Hence $\alpha = \beta$.

Case $\gamma = \langle C_2, s, h\rangle$:

By our assumption we know that $(s,h) \vDash P$ but since $\langle C_2, s, h\rangle \nvDash \varphi$ then it is the case that the right-most premise is invalid.

If there is a $\Box$-trace following the edge, then by Definition 3.2.7, $\varphi = AG\psi$. Furthermore, since by our previous result $\langle C_2, s, h\rangle \nvDash AG\psi$, then by Definition 2.3.2 we know that there exists a configuration $\gamma_\neg$ such that $\langle C_2, s, h\rangle \rightsquigarrow^* \gamma_\neg$ and $\gamma_\neg \nvDash \psi$. Finally, since by the operational semantics $\langle \textbf{while} * \textbf{do}\, C_1\, \textbf{od}\, ; C_2, s, h\rangle \rightsquigarrow \langle C_2, s, h\rangle$, then every path π' starting from $\langle C_2, s, h\rangle$ is a subpath of a path π starting from $\langle \textbf{while} * \textbf{do}\, C_1\, \textbf{od}\, ; C_2, s, h\rangle$. Consequently $\text{length}(\pi') < \text{length}(\pi)$.

Given the structure of the temporal formula $\varphi / \Box\, \varphi$, there cannot be a $\Diamond$-trace following the edge.

If there is a precondition trace $(\Psi(\mathbf{E}), \Psi'(\mathbf{E}))$ following the edge, where $(s,h) \vDash \Psi^{\alpha}(\mathbf{E})$ and $(s',h') \vDash \Psi'^{\beta}(\mathbf{E})$, then, since the precondition for both conclusion and premise is the same, then the inductive predicates $\Psi(\mathbf{E})$ and $\Psi'(\mathbf{E})$ in both conclusion and premise are interpreted by the same least approximant. Hence $\alpha = \beta$.

Soundness of Wh*◇1

$$\frac{P \vdash (C_1\, ; \textbf{while}\, B\, \textbf{do}\, C_1\, \textbf{od}\, ; C_2) : \varphi}{P \vdash (\textbf{while} * \textbf{do}\, C_1\, \textbf{od}\, ; C_2) : \Diamond\varphi}\ (\text{Wh}^* \Diamond 1)$$

Assume the conclusion of the rule is invalid. Pick an arbitrary program state s,h such that the precondition is satisfied $(s,h) \vDash P$ but the program configuration $\gamma = \langle \textbf{while} * \textbf{do}\, C_1 \textbf{ od} \,;C_2,s,h\rangle$ does not satisfy the temporal property $\Diamond\varphi$. By Definition 2.3.2, if $\gamma \nvDash \Diamond\varphi$ then for all configurations γ' such that $\gamma \rightsquigarrow \gamma'$ it is the case that $\gamma' \nvDash \varphi$. By the operational semantics of our programming language we know that there are two possible transitions from the current configuration, $\gamma' = \langle (C_1 \,; \textbf{while} * \textbf{do}\, C_1 \textbf{ od} \,; C_2),s,h\rangle$ and $\gamma' = \langle C_2,s,h\rangle$. Hence, we know that $\langle (C_1 \,; \textbf{while}\, B\, \textbf{do}\, C_1 \textbf{ od} \,; C_2),s,h\rangle \nvDash \varphi$ and $\langle C_2,s,h\rangle \nvDash \varphi$. Consequently, since by our assumption $(s,h) \vDash P$ but $\gamma' \nvDash \varphi$ then it is the case that the premise is invalid.

Given the structure of the temporal formula $\varphi / \Diamond\, \varphi$, there cannot be a $\Box$-trace following the edge.

If there is a $\Diamond$-trace following the edge, then by Definition 3.2.7, $\varphi = EG\psi$. Furthermore, since by our previous invalidity result $\gamma' \nvDash EG\psi$, then by Definition 2.3.2 we know that for all paths π' starting from γ' there exists a configuration $\gamma_\neg \in \pi$ such that $\gamma_\neg \nvDash \psi$. In other words, there is a finite tree κ' with root in γ' whose all leaves $\gamma_\neg \nvDash \psi$. Finally, since by the operational semantics $\gamma \rightsquigarrow \gamma'$, then, by definition, every tree κ' with root in γ' is a subtree of the tree κ with root in γ. Hence $\text{height}(\kappa') \leq \text{height}(\kappa)$ and for all leaves $\gamma_\neg \in \kappa'$ then $\gamma_\neg \in \kappa$.

If there is a precondition trace $(\Psi(\mathbf{E}),\Psi'(\mathbf{E}))$ following the edge, where $(s,h) \vDash \Psi^\alpha(\mathbf{E})$ and $(s',h') \vDash \Psi'^\beta(\mathbf{E})$, then, since the precondition for both conclusion and premise is the same, then the inductive predicates $\Psi(\mathbf{E})$ and $\Psi'(\mathbf{E})$ in both conclusion and premise are interpreted by the same least approximant. Hence $\alpha = \beta$.

Soundness of Wh*◇2

$$\frac{P \vdash C_2 : \varphi}{P \vdash (\textbf{while} * \textbf{do}\, C_1 \textbf{ od} \,; C_2) : \Diamond\varphi}\ (\text{Wh}^*\Diamond 2)$$

Assume the conclusion of the rule is invalid. Pick an arbitrary program state s,h such that the precondition is satisfied $(s,h) \vDash P$ but the program configuration $\langle \textbf{while} * \textbf{do}\, C_1 \textbf{ od} \,; C_2,s,h\rangle$ does not satisfy the temporal property $\Diamond\varphi$. By

Definition 2.3.2, if $\langle \mathbf{while} * \mathbf{do}\, C_1\, \mathbf{od}\, ; C_2, s, h\rangle \not\models \Diamond\varphi$ then for all configurations γ such that $\langle \mathbf{while} * \mathbf{do}\, C_1\, \mathbf{od}\, ; C_2, s, h\rangle \rightsquigarrow \gamma$ it is the case that $\gamma \not\models \varphi$. By the operational semantics of our programming language we know that there are two possible transitions from the current configuration, $\langle \mathbf{while} * \mathbf{do}\, C_1\, \mathbf{od}\, ; C_2, s, h\rangle \rightsquigarrow \langle (C_1; \mathbf{while} * \mathbf{do}\, C_1\, \mathbf{od}; C_2), s, h\rangle$ and $\langle \mathbf{while} * \mathbf{do}\, C_1\, \mathbf{od}; C_2, s, h\rangle \rightsquigarrow \langle C_2, s, h\rangle$. Hence, by our assumption we know that $\langle (C_1\, ; \mathbf{while} * \mathbf{do}\, C_1\, \mathbf{od}\, ; C_2), s, h\rangle \not\models \varphi$ and $\langle C_2, s, h\rangle \not\models \varphi$

Consequently, since by our assumption $(s,h) \models P$ but $\langle C_2, s, h\rangle \not\models \varphi$ then it is the case that the premise is invalid.

Given the structure of the temporal formula $\varphi / \Diamond\, \varphi$, there cannot be a $\Box$-trace following the edge.

If there is a $\Diamond$-trace following the edge, then by Definition 3.2.7, $\varphi = EG\psi$. Furthermore, since by our previous invalidity result $\langle C_2, s, h\rangle \not\models EG\psi$, then by Definition 2.3.2 we know that for all paths π starting from $\langle C_2, s, h\rangle$ there exists a configuration $\gamma_\neg \in \pi$ such that $\gamma_\neg \not\models \psi$. In other words, there is a finite tree κ' with root in $\langle C_2, s, h\rangle$ whose all leaves $\gamma_\neg \not\models \psi$. Finally, since by the operational semantics $\langle \mathbf{while} * \mathbf{do}\, C_1\, \mathbf{od}\, ; C_2, s, h\rangle \rightsquigarrow \langle C_2, s, h\rangle$, then, by definition, every tree with root in $\langle C_2, s, h\rangle$ is a subtree of the tree κ with root in $\langle \mathbf{while} * \mathbf{do}\, C_1\, \mathbf{od}\, ; C_2, s, h\rangle$. Hence $\text{height}(\kappa') \leq \text{height}(\kappa)$ and for all leaves $\gamma_\neg \in \kappa'$ then $\gamma_\neg \in \kappa$.

If there is a precondition trace $(\Psi(\mathbf{E}), \Psi'(\mathbf{E}))$ following the edge, where $(s,h) \models \Psi^\alpha(\mathbf{E})$ and $(s',h') \models \Psi'^\beta(\mathbf{E})$, then, since the precondition for both conclusion and premise is the same, then the inductive predicates $\Psi(\mathbf{E})$ and $\Psi'(\mathbf{E})$ in both conclusion and premise are interpreted by the same least approximant. Hence $\alpha = \beta$.

Soundness of R$\perp$

$$\frac{P * E \mapsto - \not\models \perp}{P \vdash x := [E]\, ; C : \bigcirc\mathsf{error}}\ (\mathrm{R}\perp)$$

Pick an arbitrary program state s,h such that the precondition P is satisfied $(s,h) \models P$. As per the side condition of the rule $P * E \mapsto - \not\models \perp$, it is possible to compose any model of the precondition with a model of $E \mapsto -$ then we know that $[\![E]\!]s \notin \mathsf{dom}(h)$. Therefore, by the operational semantics of our programming language we know that $\langle x := [E] \,;\, C, s, h\rangle \rightsquigarrow \mathsf{fault}$. Consequently, by Definition 2.3.2 we know that $\langle x := [E] \,;\, C, s, h\rangle \vDash \bigcirc\mathsf{error}$.

Soundness of W⊥

$$\frac{P * E \mapsto - \nvDash \bot}{P \vdash [E] := E' \,;\, C : \bigcirc\mathsf{error}} \ (\mathrm{W}\bot)$$

Pick an arbitrary program state s, h such that the precondition P is satisfied $(s,h) \vDash P$. As per the side condition of the rule $P * E \mapsto - \nvDash \bot$, it is possible to compose any model of the precondition with a model of $E \mapsto -$ then we know that $[\![E]\!]s \notin \mathsf{dom}(h)$. Therefore, by the operational semantics of our programming language we know that $\langle [E] := E' \,;\, C, s, h\rangle \rightsquigarrow \mathsf{fault}$. Consequently, by Definition 2.3.2 we know that $\langle [E] := E' \,;\, C, s, h\rangle \vDash \bigcirc\mathsf{error}$.

Soundness of Free ⊥

$$\frac{P * E \mapsto - \nvDash \bot}{P \vdash \mathbf{free}(E) \,;\, C : \bigcirc\mathsf{error}} \ (\mathrm{Free}\bot)$$

Pick an arbitrary program state s, h such that the precondition P is satisfied $(s,h) \vDash P$. As per the side condition of the rule $P * E \mapsto - \nvDash \bot$, it is possible to compose any model of the precondition with a model of $E \mapsto -$ then we know that $[\![E]\!]s \notin \mathsf{dom}(h)$. Therefore, by the operational semantics of our programming language we know that $\langle \mathtt{free}(E) \,;\, C, s, h\rangle \rightsquigarrow \mathsf{fault}$. Consequently, by Definition 2.3.2 we know that $\langle \mathtt{free}(E) \,;\, C, s, h\rangle \vDash \bigcirc\mathsf{error}$.

Soundness of Check

$$\frac{P \vDash Q}{P \vdash C : Q} \ (\mathrm{Check})$$

Pick an arbitrary program state s, h such that the precondition P is satisfied. Formally $(s,h) \vDash P$. We need to show that when an arbitrary program C is executed in the given state, the such configuration would satisfy Q. Formally $\langle C, s, h\rangle \vDash Q$.

Proving that $\langle C, s, h\rangle \vDash Q$ is trivial since by the side condition of the proof rule we have that every model of P is a model of Q (i.e. $P \vDash Q$) and since $(s,h) \vDash P$ then by Definition 2.3.2 we have $\langle C, s, h\rangle \vDash Q$.

Soundness of Ex.Falso

$$\frac{}{\bot \vdash C : \varphi} \text{ (Ex.Falso)}$$

Pick an arbitrary program state s,h such that the precondition P is satisfied $(s,h) \vDash \bot$. As by Definition 2.2.2 $(s,h) \vDash \bot$ never, proving that $\langle C,s,h\rangle \vDash \varphi$ is trivial as anything follows from false.

Soundness of Split

$$\frac{\Omega_1 \vdash C : \varphi \quad \Omega_2 \vdash C : \varphi}{\Omega_1 \vee \Omega_2 \vdash C : \varphi} \text{ (Split)}$$

Assume the conclusion of the rule is invalid. Pick an arbitrary program state s,h such that the precondition is satisfied $(s,h) \vDash \Omega_1 \vee \Omega_2$ but the program configuration $\langle C,s,h\rangle$ does not satisfy the temporal property φ. If $(s,h) \vDash \Omega_1 \vee \Omega_2$ then we know that either $(s,h) \vDash \Omega_1$ or $(s,h) \vDash \Omega_2$.

Case $(s,h) \vDash \Omega_1$. Let $s' = s$ and $h' = h$. Since by our assumption $(s',h') \vDash \Omega_1$ but $\langle C,s',h'\rangle \nvDash \varphi_1$ then it is the case that the left-most premise is invalid.

Case $(s,h) \vDash \Omega_2$. Let $s' = s$ and $h' = h$. Since by our assumption $(s',h') \vDash \Omega_2$ but $\langle C,s',h'\rangle \nvDash \varphi_1$ then it is the case that the right-most premise is invalid.

If there is a $\Box$-trace following the edge, then by Definition 3.2.7, $\varphi = AG\psi$. Furthermore, since by our previous invalidity result $\langle C,s,h\rangle \nvDash AG\psi$, then by Definition 2.3.2 we know that there exists a configuration $\gamma_\neg$ such that $\langle C,s,h\rangle \rightsquigarrow^* \gamma_\neg$ and $\gamma_\neg \nvDash \psi$. Finally, since $\langle C,s,h\rangle = \langle C,s',h'\rangle$, then the length of the path $\langle C,s,h\rangle \rightsquigarrow^* \gamma_\neg$ is the same as the length of the path $\langle C,s',h'\rangle \rightsquigarrow^* \gamma_\neg$.

If there is a $\Diamond$-trace following the edge, then by Definition 3.2.7, $\varphi = EG\psi$. Furthermore, since by our previous invalidity result $\langle C,s,h\rangle \nvDash EG\psi$, then by Definition 2.3.2 we know that for all paths π starting from $\langle C,s,h\rangle$ there exists a configuration $\gamma_\neg \in \pi$ such that $\gamma_\neg \nvDash \psi$. In other words, there is a finite tree κ with root in $\langle C,s,h\rangle$ whose all leaves $\gamma_\neg \nvDash \psi$. Finally, since by construction $\langle C,s,h\rangle = \langle C,s',h'\rangle$, the execution tree κ with root in $\langle C,s,h\rangle$ is the same as κ' and for all leaves $\gamma_\neg \in \kappa$ then $\gamma_\neg \in \kappa'$.

If there is a precondition trace $(\Psi(\mathbf{E}),\Psi'(\mathbf{E}))$ following the edge, where $(s,h) \vDash \Psi^\alpha(\mathbf{E})$ and $(s',h') \vDash \Psi'^\beta(\mathbf{E})$, then, since the precondition for both conclusion and premise is the same, then the inductive predicates $\Psi(\mathbf{E})$ and $\Psi'(\mathbf{E})$ in both conclusion and premise are interpreted by the same least approximant. Hence $\alpha = \beta$.

Soundness of Substitution

$$\frac{x \notin \mathrm{vars}(C) \quad P \vdash C : \varphi}{P[E/x] \vdash C : \varphi[E/x]} \text{ (Subst)}$$

Assume the conclusion of the rule is invalid. Pick an arbitrary program state $s,h[\llbracket E \rrbracket s \mapsto \_]$ such that the precondition is satisfied $(s,h[\llbracket E \rrbracket s \mapsto \_]) \vDash P[E/x]$ but the program configuration $\langle C,s,h[\llbracket E \rrbracket s \mapsto \_]\rangle$ does not satisfy the temporal property $\varphi[E/x]$. By Lemma 2.2.7 we know that $(s,h) \vDash P$ and moreover, since by the side condition of the rule $x \notin \mathrm{vars}(C)$ then we know that whenever $\langle C,s,h[\llbracket E \rrbracket s \mapsto \_]\rangle \vDash \varphi[E/x]$ then $\langle C,s,h\rangle \vDash \varphi$. Alternatively whenever $\langle C,s,h[\llbracket E \rrbracket s \mapsto \_]\rangle \nvDash \varphi[E/x]$ then $\langle C,s,h\rangle \nvDash \varphi$. Finally, since $(s,h) \vDash P$ but $\langle C,s,h\rangle \nvDash \varphi$ then the premise of the rule is invalid.

If there is a $\square$-trace following the edge, then by Definition 3.2.7, $\varphi = AG\psi$. Furthermore, since by our previous invalidity result $\langle C,s,h\rangle \nvDash AG\psi$, then by Definition 2.3.2 we know that there exists a configuration $\gamma_\neg$ such that $\langle C,s,h\rangle \rightsquigarrow^* \gamma_\neg$ and $\gamma_\neg \nvDash \psi$. Finally, since $x \notin \mathrm{vars}(C)$ then we know that $\langle C,s,h\rangle \rightsquigarrow^* \gamma_\neg$ follows the same path as $\langle C,s,h[\llbracket E \rrbracket s \mapsto \_]\rangle \rightsquigarrow^* \gamma_\neg$, then the length of the path $\langle C,s,h\rangle \rightsquigarrow^* \gamma_\neg$ is the same as the length of the path $\langle C,s,h[\llbracket E \rrbracket s \mapsto \_]\rangle \rightsquigarrow^* \gamma_\neg$.

If there is a $\Diamond$-trace following the edge, then by Definition 3.2.7, $\varphi = EG\psi$. Furthermore, since by our previous invalidity result $\langle C,s,h\rangle \nvDash EG\psi$, then by Definition 2.3.2 we know that for all paths π starting from $\langle C,s,h\rangle$ there exists a configuration $\gamma_\neg \in \pi$ such that $\gamma_\neg \nvDash \psi$. In other words, there is a finite tree κ with root in $\langle C,s,h\rangle$ whose all leaves $\gamma_\neg \nvDash \psi$. Finally, since $x \notin \mathrm{vars}(C)$ the execution tree κ with root in $\langle C,s,h\rangle$ follows the same execution paths as κ' and for all leaves $\gamma_\neg \in \kappa$ then $\gamma_\neg \in \kappa'$.

If there is a precondition trace following the edge then since the precondition for both conclusion and premise is the same (modulo the variable substitution

$P[x'/x]$), then the inductive predicates $\Psi(\mathbf{E})$ in P for both conclusion and premise are defined by the same least approximant. Hence $\alpha = \beta$.

Soundness of Conj

$$\frac{P \vdash C : \varphi_1 \quad P \vdash C : \varphi_2}{P \vdash C : \varphi_1 \wedge \varphi_2} \text{ (Conj)}$$

Assume the conclusion of the rule is invalid. Pick an arbitrary program state s,h such that the precondition is satisfied $(s,h) \vDash P$ but the program configuration $\langle C,s,h\rangle$ does not satisfy the temporal property $\varphi_1 \wedge \varphi_2$. By Definition 2.3.2, if $\langle C,s,h\rangle \nvDash \varphi_1 \wedge \varphi_2$ then either $\langle C,s,h\rangle \nvDash \varphi_1$ or $\langle C,s,h\rangle \nvDash \varphi_2$.
Case $\langle C,s,h\rangle \nvDash \varphi_1$. Since by our assumption $(s,h) \vDash P$ but $\langle C,s,h\rangle \nvDash \varphi_1$ then it is the case that the left-most premise is invalid.
Case $\langle C,s,h\rangle \nvDash \varphi_2$. Since by our assumption $(s,h) \vDash P$ but $\langle C,s,h\rangle \nvDash \varphi_2$ then it is the case that the right-most premise is invalid.

Given the temporal property with formula $\varphi_1 \wedge \varphi_2$, there cannot be a $\Box$- or a $\Diamond$-trace following the path.

If there is a precondition trace $(\Psi(\mathbf{E}), \Psi'(\mathbf{E}))$ following the edge, where $(s,h) \vDash \Psi^{\alpha}(\mathbf{E})$ and $(s',h') \vDash \Psi'^{\beta}(\mathbf{E})$, then, since the precondition for both conclusion and premise is the same, then the inductive predicates $\Psi(\mathbf{E})$ and $\Psi'(\mathbf{E})$ in both conclusion and premise are interpreted by the same least approximant. Hence $\alpha = \beta$.

Soundness of $\vee$1

$$\frac{P \vdash C : \varphi_1}{P \vdash C : \varphi_1 \vee \varphi_2} \text{ } (\vee 1)$$

Assume the conclusion of the rule is invalid. Pick an arbitrary program state s,h such that the precondition is satisfied $(s,h) \vDash P$ but the program configuration $\langle C,s,h\rangle$ does not satisfy the temporal property $\varphi_1 \vee \varphi_2$. By Definition 2.3.2, if $\langle C,s,h\rangle \nvDash \varphi_1 \vee \varphi_2$ then $\langle C,s,h\rangle \nvDash \varphi_1$ and $\langle C,s,h\rangle \nvDash \varphi_2$. Consequently, since by our assumption $(s,h) \vDash P$ but $\langle C,s,h\rangle \nvDash \varphi_1$ then it is the case that the premise is invalid.

Given the temporal property with formula $\varphi_1 \vee \varphi_2$, there cannot be a $\Box$- or a $\Diamond$-trace following the path.

If there is a precondition trace $(\Psi(\mathbf{E}), \Psi'(\mathbf{E}))$ following the edge, where

$(s,h) \vDash \Psi^{\alpha}(\mathbf{E})$ and $(s',h') \vDash \Psi'^{\beta}(\mathbf{E})$, then, since the precondition for both conclusion and premise is the same, then the inductive predicates $\Psi(\mathbf{E})$ and $\Psi'(\mathbf{E})$ in both conclusion and premise are interpreted by the same least approximant. Hence $\alpha = \beta$.

Soundness of $\vee 2$

$$\frac{P \vdash C : \varphi_2}{P \vdash C : \varphi_1 \vee \varphi_2}\ (\vee 2)$$

Similar to soundness of $\vee 1$.

Soundness of EF

$$\frac{P \vdash C : \varphi \vee \Diamond EF\varphi}{P \vdash C : EF\varphi}\ (\text{EF})$$

Assume the conclusion of the rule is invalid. Pick an arbitrary program state s,h such that the precondition is satisfied $(s,h) \vDash P$ but the program configuration $\langle C,s,h\rangle$ does not satisfy the temporal property $EF\varphi$. By Definition 2.3.2, if $\langle C,s,h\rangle \nvDash EF\varphi$ then for all γ such that $\langle C,s,h\rangle \leadsto^{*} \gamma$ it is the case that $\gamma \nvDash \varphi$.

By the reflexivity of $\leadsto$ we know that $\langle C,s,h\rangle \leadsto^{0} \langle C,s,h\rangle$. Therefore by our assumption it is the case that $(i)\langle C,s,h\rangle \nvDash \varphi$. Moreover, by the transitivity of $\leadsto$ we know that for all γ if $\langle C,s,h\rangle \leadsto^{+} \gamma$ then $\gamma \nvDash \varphi$. In other words there does not exist a one step transition for which eventually φ will hold. Formally $(ii)\langle C,s,h\rangle \nvDash \Diamond EF\varphi$. Consequently, by our assumption $(s,h) \vDash P$ and results (i) and (ii) the premise of the rule must be invalid.

Given the temporal property with formula $(\Diamond)EF\varphi$, there cannot be a $\Box$- or a $\Diamond$-trace following the path.

If there is a precondition trace $(\Psi(\mathbf{E}),\Psi'(\mathbf{E}))$ following the edge, where $(s,h) \vDash \Psi^{\alpha}(\mathbf{E})$ and $(s',h') \vDash \Psi'^{\beta}(\mathbf{E})$, then, since the precondition for both conclusion and premise is the same, then the inductive predicates $\Psi(\mathbf{E})$ and $\Psi'(\mathbf{E})$ in both conclusion and premise are interpreted by the same least approximant. Hence $\alpha = \beta$.

Soundness of EG

$$\frac{P \vdash C : \varphi \quad P \vdash C : \Diamond EG\varphi}{P \vdash C : EG\varphi}\ (\text{EG})$$

Assume the conclusion of the rule is invalid. Pick an arbitrary program state s, h such that the precondition is satisfied $(s,h) \vDash P$ but the program configuration $\langle C,s,h\rangle$ does not satisfy the temporal property $EG\varphi$. By Definition 2.3.2, if $\langle C,s,h\rangle \nvDash EG\varphi$ then for all paths *pi* starting from $\langle C,s,h\rangle$ there exist $\gamma \in \pi$ such that $\gamma \nvDash \varphi$.

Since $\gamma \in \pi$ then it must be the case that $\langle C,s,h\rangle \leadsto^0 \gamma$ or $\langle C,s,h\rangle \leadsto^+ \gamma$. If $\langle C,s,h\rangle \leadsto^0 \gamma$ then we know that $(i)\langle C,s,h\rangle \nvDash \varphi$. If $\langle C,s,h\rangle \leadsto^+ \gamma$ then we know that $(ii)\langle C,s,h\rangle \nvDash \Diamond EG\varphi$. Letting $s' = s$ and $h' = h$, by our assumption $(s',h') \vDash P$ and results (i) and (ii), then either one of the premises of the rule is invalid.

Given the structure of the temporal formula $\varphi / \Diamond\, \varphi$, there cannot be a $\Box$-trace following the edge.

If there is a $\Diamond$-trace following the edge of the right hand premise, then by Definition 3.2.7, $\varphi = EG\psi$. Furthermore, since by our previous invalidity result $\langle C,s,h\rangle \nvDash EG\psi$, then by Definition 2.3.2 we know that for all paths π starting from $\langle C,s,h\rangle$ there exists a configuration $\gamma_\neg \in \pi$ such that $\gamma_\neg \nvDash \psi$. In other words, there is a finite tree κ' with root in $\langle C,s,h\rangle$ whose all leaves $\gamma_\neg \nvDash \psi$. Finally, since by construction $\langle C,s,h\rangle = \langle C,s',h'\rangle$, then every tree κ with root in $\langle C,s,h\rangle$ is the same tree κ' with root in $\langle C,s',h'\rangle$. Hence height(κ') = height(κ) and for all leaves $\gamma_\neg \in \kappa'$ then $\gamma_\neg \in \kappa$.

If there is a precondition trace $(\Psi(\mathbf{E}), \Psi'(\mathbf{E}))$ following the edge, where $(s,h) \vDash \Psi^\alpha(\mathbf{E})$ and $(s',h') \vDash \Psi'^\beta(\mathbf{E})$, then, since the precondition for both conclusion and premise is the same, then the inductive predicates $\Psi(\mathbf{E})$ and $\Psi'(\mathbf{E})$ in both conclusion and premise are interpreted by the same least approximant. Hence $\alpha = \beta$.

Soundness of EU

$$\frac{P \vdash C : \psi \vee (\varphi \wedge \Diamond E(\varphi U \psi))}{P \vdash C : E(\varphi U \psi)} \text{ (EU))}$$

Assume the conclusion of the rule is invalid. Pick an arbitrary program state s,h such that the precondition is satisfied $(s,h) \vDash P$ but the program configuration $\langle C,s,h\rangle$ does not satisfy the temporal property $E(\varphi U \psi)$. By Definition 2.3.2, if $\langle C,s,h\rangle \nvDash E(\varphi U \psi)$ then for all paths π starting from $\langle C,s,h\rangle$ and for all $i \geq 0$ either

$\pi_i \nvDash \psi$ or there exists $j : 0 \leq j \leq i$ such that $\pi_j \nvDash \varphi$.

In other words, for $i = 0$, since there does not exists $j : 0 \leq j < 0$ then it must be the case that $(i)\langle C,s,h\rangle \nvDash \psi$. For $i > 0$ it must be the case that $(ii)\langle C,s,h\rangle \nvDash \varphi$ or $\langle C,s,h\rangle \rightsquigarrow^{+} \gamma'$ and for all paths π' starting from γ' and for all $i' \geq 0$ either $\pi'_{i'} \nvDash \psi$ or there exists $j' : 0 \leq j' < i'$ such that $\pi_{j'} \nvDash \varphi$. In other words $(iii)\langle C,s,h\rangle \nvDash \Diamond E(\varphi U \psi)$.

Consequently, by our assumption $(s,h) \vDash P$ and results (i), (ii) and (iii) the premise of the rule is invalid.

Given the temporal property, there cannot be a $\Box$- or a $\Diamond$-trace following the path.

If there is a precondition trace $(\Psi(\mathbf{E}), \Psi'(\mathbf{E}))$ following the edge, where $(s,h) \vDash \Psi^{\alpha}(\mathbf{E})$ and $(s',h') \vDash \Psi'^{\beta}(\mathbf{E})$, then, since the precondition for both conclusion and premise is the same, then the inductive predicates $\Psi(\mathbf{E})$ and $\Psi'(\mathbf{E})$ in both conclusion and premise are interpreted by the same least approximant. Hence $\alpha = \beta$.

Soundness of AF

$$\frac{P \vdash C : \varphi \vee \Box AF\varphi}{P \vdash C : AF\varphi}\ \text{(AF)}$$

Assume the conclusion of the rule is invalid. Pick an arbitrary program state s,h such that the precondition is satisfied $(s,h) \vDash P$ but the program configuration $\langle C,s,h\rangle$ does not satisfy the temporal property $AF\varphi$. By Definition 2.3.2, if $\langle C,s,h\rangle \nvDash AF\varphi$ then there exists a path π starting from $\langle C,s,h\rangle$ where for all $\gamma \in \pi$ it is the case that $\gamma \nvDash \varphi$.

Under this assumption, since by definition $\langle C,s,h\rangle \in \pi$ then we know that $(i)\langle C,s,h\rangle \nvDash \varphi$. Moreover, by the existence of a path π where for all $\gamma \in \pi$ it is the case that $\gamma \nvDash \varphi$ then we also know that $(ii)\langle C,s,h\rangle \nvDash \Box AF\varphi$. Consequently, by our assumption $(s,h) \vDash P$ and results (i) and (ii) the premise of the rule is invalid.

Given the temporal property with formula $(\Box)AF\varphi$, there cannot be a $\Box$- or a $\Diamond$-trace following the path.

If there is a precondition trace $(\Psi(\mathbf{E}), \Psi'(\mathbf{E}))$ following the edge, where $(s,h) \vDash \Psi^{\alpha}(\mathbf{E})$ and $(s',h') \vDash \Psi'^{\beta}(\mathbf{E})$, then, since the precondition for both conclusion and premise is the same, then the inductive predicates $\Psi(\mathbf{E})$ and $\Psi'(\mathbf{E})$ in both conclusion and premise are interpreted by the same least approximant. Hence $\alpha = \beta$.

Soundness of AG

$$\frac{P \vdash C : \varphi \quad P \vdash C : \Box AG\varphi}{P \vdash C : AG\varphi}\ (\text{AG})$$

Assume the conclusion of the rule is invalid. Pick an arbitrary program state s, h such that the precondition is satisfied $(s,h) \vDash P$ but the program configuration $\langle C, s, h\rangle$ does not satisfy the temporal property $AG\varphi$. By Definition 2.3.2, if $\langle C, s, h\rangle \nvDash AG\varphi$ then there exist γ such that $\langle C, s, h\rangle \rightsquigarrow^* \gamma$ and $\gamma \nvDash \varphi$.

Since $\langle C, s, h\rangle \rightsquigarrow^* \gamma$ then it must be the case that $\langle C, s, h\rangle \rightsquigarrow^0 \gamma$ or $\langle C, s, h\rangle \rightsquigarrow^+ \gamma$. If $\langle C, s, h\rangle \rightsquigarrow^0 \gamma$ then we know that $(i)\langle C, s, h\rangle \nvDash \varphi$. If $\langle C, s, h\rangle \rightsquigarrow^+ \gamma$ then we know that for all paths π starting from $\langle C, s, h\rangle$ there does not exists a configuration γ such that $\gamma \vDash \varphi$. Formally $(ii)\langle C, s, h\rangle \nvDash \Box AG\varphi$. Letting $s' = s$ and $h' = h$, by our assumption $(s', h') \vDash P$ and results (i) and (ii), then either one of the premises of the rule is invalid.

If there is a $\Box$-trace following the edge, then by Definition 3.2.7, $\varphi = AG\psi$. Furthermore, since by our previous invalidity result $\langle C, s, h\rangle \nvDash AG\psi$, then by Definition 2.3.2 we know that there exists a configuration $\gamma_\neg$ such that $\langle C, s, h\rangle \rightsquigarrow^* \gamma_\neg$ and $\gamma_\neg \nvDash \psi$. Finally, since by construction $\langle C, s, h\rangle = \langle C, s', h'\rangle$, then every path π starting from $\langle C, s, h\rangle$ is the same path as the path π' starting from $\langle C, s', h'\rangle$. Consequently $\text{length}(\pi') = \text{length}(\pi)$.

Given the structure of the temporal formula $\varphi / \Box\, \varphi$, there cannot be a $\Diamond$-trace following the edge.

If there is a precondition trace $(\Psi(\mathbf{E}), \Psi'(\mathbf{E}))$ following the edge, where $(s,h) \vDash \Psi^\alpha(\mathbf{E})$ and $(s', h') \vDash \Psi'^\beta(\mathbf{E})$, then, since the precondition for both conclusion and premise is the same, then the inductive predicates $\Psi(\mathbf{E})$ and $\Psi'(\mathbf{E})$ in both conclusion and premise are interpreted by the same least approximant. Hence $\alpha = \beta$.

Soundness of AU

$$\frac{P \vdash C : \psi \vee (\varphi \wedge \Box A(\varphi U \psi))}{P \vdash C : A(\varphi U \psi)}\ (\text{AU})$$

Assume the conclusion of the rule is invalid. Pick an arbitrary program state s,h such that the precondition is satisfied $(s,h) \vDash P$ but the program configuration $\langle C,s,h \rangle$ does not satisfy the temporal property $A(\varphi U \psi)$. By Definition 2.3.2, if $\langle C,s,h \rangle \nvDash A(\varphi U \psi)$ then there exists a path π starting from $\langle C,s,h \rangle$ where for all $i \geq 0$ either $\pi_i \nvDash \psi$ or there exists $j : 0 \leq j \leq i$ such that $\pi_j \nvDash \varphi$.

In other words, for $i = 0$, since there does not exists $j : 0 \leq j < 0$ then it must be the case that $(i)\langle C,s,h \rangle \nvDash \psi$. For $i > 0$ it must be the case that $(ii)\langle C,s,h \rangle \nvDash \varphi$ or $\langle C,s,h \rangle \rightsquigarrow^+ \gamma'$ and there exists a path π' starting from γ' where for all $i' \geq 0$ either $\pi'_{i'} \nvDash \psi$ or there exists $j' : 0 \leq j' < i'$ such that $\pi_{j'} \nvDash \varphi$. In other words $(iii)\langle C,s,h \rangle \nvDash \Box A(\varphi U \psi)$.

Consequently, by our assumption $(s,h) \vDash P$ and results (i), (ii) and (iii) the premise of the rule is invalid.

Given the temporal property, there cannot be a $\Box$- or a $\Diamond$-trace following the path.

If there is a precondition trace $(\Psi(\mathbf{E}), \Psi'(\mathbf{E}))$ following the edge, where $(s,h) \vDash \Psi^{\alpha}(\mathbf{E})$ and $(s',h') \vDash \Psi'^{\beta}(\mathbf{E})$, then, since the precondition for both conclusion and premise is the same, then the inductive predicates $\Psi(\mathbf{E})$ and $\Psi'(\mathbf{E})$ in both conclusion and premise are interpreted by the same least approximant. Hence $\alpha = \beta$.

Soundness of EG-Finite

$$\frac{P \vdash \varepsilon : \varphi}{P \vdash \varepsilon : EG\varphi} \text{ (EG-Finite)}$$

Assume the conclusion of the rule is invalid. Pick an arbitrary program state s,h such that the precondition is satisfied $(s,h) \vDash P$ but the program configuration $\langle \varepsilon,s,h \rangle$ does not satisfy the temporal property $EG\varphi$. By Definition 2.3.2, if $\langle \varepsilon,s,h \rangle \nvDash EG\varphi$ then for all paths pi starting from $\langle C,s,h \rangle$ there exist $\gamma \in \pi$ such that $\gamma \nvDash \varphi$.

Since by the operational semantics there are no possible transitions from the current configuration, then it must be the case that all paths are comprised of the single configuration $\langle\varepsilon,s,h\rangle$ (i.e. for all paths $\pi, \pi = \langle\varepsilon,s,h\rangle \rightsquigarrow^0 \langle\varepsilon,s,h\rangle$). Therefore, as a consequence of our argument, if $\langle\varepsilon,s,h\rangle \nvDash EG\varphi$ then it must be the case that $\langle\varepsilon,s,h\rangle \nvDash \varphi$. Finally, since by our assumption $(s,h) \vDash P$ but $\langle\varepsilon,s,h\rangle \nvDash \varphi$ then the premise of the rule is invalid.

Given the temporal property, there cannot be a $\Box$- or a $\Diamond$-trace following the path.

If there is a precondition trace $(\Psi(\mathbf{E}),\Psi'(\mathbf{E}))$ following the edge, where $(s,h) \vDash \Psi^{\alpha}(\mathbf{E})$ and $(s',h') \vDash \Psi'^{\beta}(\mathbf{E})$, then, since the precondition for both conclusion and premise is the same, then the inductive predicates $\Psi(\mathbf{E})$ and $\Psi'(\mathbf{E})$ in both conclusion and premise are interpreted by the same least approximant. Hence $\alpha = \beta$.

Soundness of Cons

$$\frac{P \vdash Q \quad Q \vdash C : \psi \quad \psi \vdash \varphi}{P \vdash C : \varphi} \text{ (Cons)}$$

Assume the conclusion of the rule is invalid. Pick an arbitrary program state s,h such that the precondition is satisfied $(s,h) \vDash P$ but the program configuration $\langle C,s,h\rangle$ does not satisfy the temporal property φ. By the side condition of the rule, since $P \vdash Q$, we have $(s,h) \vDash Q$. Moreover, since by the side condition of the rule $\psi \vdash \varphi$ then if a configuration $\gamma \vDash \psi$ then it should be the case that $\gamma \vDash \varphi$. Alternatively, if $\gamma \nvDash \varphi$ then is must be the case that $\gamma \nvDash \psi$. Consequently, following the reasoning of our argument $\langle C,s,h\rangle \nvDash \varphi$. Finally, since by our assumption $(s,h) \vDash Q$ but $\langle C,s,h\rangle \nvDash \psi$ then the premise of the rule is invalid.

If there is a $\Diamond$-trace following the edge, then by Definition 3.2.7, $\varphi = \psi = EG\varphi'$. Furthermore, since by our previous invalidity result $\langle C,s,h\rangle \nvDash EG\varphi'$, then by Definition 2.3.2 we know that for all paths π starting from $\langle C,s,h\rangle$ there exists a configuration $\gamma_{\neg} \in \pi$ such that $\gamma_{\neg} \nvDash \varphi'$. In other words, there is a finite tree κ with root in $\langle C,s,h\rangle$ whose all leaves $\gamma_{\neg} \nvDash \varphi'$. Finally, since $P \vdash Q$ then every tree κ' with root in $\langle C,s',h'\rangle$ is the same tree as κ with root in $\langle C,s,h\rangle$. Hence $\text{height}(\kappa') = \text{height}(\kappa)$ and for all leaves $\gamma_{\neg} \in \kappa'$ then $\gamma_{\neg} \in \kappa$.

If there is a $\square$-trace following the edge, then by Definition 3.2.7, $\varphi = \psi = AG\varphi'$. Furthermore, since by our previous invalidity result $\langle C,s,h\rangle \nvDash AG\varphi'$, then by Definition 2.3.2 we know that there exists a configuration $\gamma_\neg$ such that $\langle C,s,h\rangle \rightsquigarrow^* \gamma_\neg$ and $\gamma_\neg \nvDash \varphi'$. Finally, since $P \vdash Q$ then every path π starting from $\langle C,s,h\rangle$ is the same path as the path π' starting from $\langle C,s',h'\rangle$. Consequently length(π') = length(π).

If there is a precondition trace $(\Psi(\mathbf{E}),\Psi'(\mathbf{E}))$ following the edge, where $(s,h) \vDash \Psi^\alpha(\mathbf{E})$ and $(s',h') \vDash \Psi'^\beta(\mathbf{E})$, then, by Definition 3.2.8 the precondition for both conclusion and premise is the same, then the inductive predicates $\Psi(\mathbf{E})$ and $\Psi'(\mathbf{E})$ in both conclusion and premise are interpreted by the same least approximant. Hence $\alpha = \beta$.

Soundness of Unfold-Pre

$$\frac{(\Pi \cup \Pi_i' : \Sigma * \Sigma_i' \vdash C : \varphi)_{1\le i\le k}}{\Pi : \Psi(\mathbf{E}) * \Sigma \vdash C : \varphi} \text{ (Unfold-Pre)}$$

Assume the conclusion of the rule is invalid. Pick an arbitrary program state s,h such that the precondition is satisfied $(s,h) \vDash \Pi : \Psi(\mathbf{E}) * \Sigma$ but the program configuration $\langle C,s,h\rangle$ does not satisfy the temporal property φ. By Definition 2.2.2 we can split h into two disjoint subheaps $h = h' \uplus h''$ so that $(s,h') \vDash \Pi : \Psi(\mathbf{E})$ and $(s,h'') \vDash \Pi : \Sigma$. Since $(s,h') \vDash \Pi : \Psi(\mathbf{E})$ then by Definition 2.2.2 we know that $([\![\mathbf{E}]\!]s,h') \in [\![\Psi]\!]$. Moreover, by Definition 2.2.4 we know that the program state (s,h') is in the semantic definition of Ψ for some approximant α (i.e. $(s,h') \in [\![\Psi]\!]^\alpha$). Let $h_1' \dots h_n'$ be disjoint heaps so that $h' = h_1' \uplus \dots \uplus h_n'$ and $(s,h_1' \uplus h'') \vDash (\Pi \cup \Pi_i' : \Sigma * \Sigma_i')_{1\le i\le k}$ (where k is the size of the set of inductive rules as per Definition 2.2.3). Since $h = h' \uplus h''$ and $h' = h_1' \uplus \dots \uplus h_n'$ then, any path starting from $\langle C,s,h' \uplus h''\rangle$ is a path starting from $\langle C,s,h_i' \uplus h''\rangle$ for some $1 \le i \le k$. Consequently, since $(s,h_i' \uplus h'') \vDash \Pi \cup \Pi_i' : \Sigma * \Sigma_i'$ but $\langle C,s,h_i' \uplus h''\rangle \nvDash \varphi$ then a premise of the rule $(\Pi \cup \Pi_i' : \Sigma * \Sigma_i' \vdash C : \varphi)_{1\le i\le k}$ is invalid for some i.

If there is a $\Diamond$-trace following the edge, then by Definition 3.2.7, $\varphi = EG\psi$. Furthermore, since by our previous invalidity result $\langle C,s,h\rangle \nvDash EG\psi$, then by Definition 2.3.2 we know that for all paths π starting from $\langle C,s,h\rangle$ there exists a configuration $\gamma_\neg \in \pi$ such that $\gamma_\neg \not\models \psi$. In other words, there is a finite tree κ of height, say n, with root in $\langle C,s,h\rangle$ whose all leaves $\gamma_\neg \not\models \psi$. Finally, since $h = h' \uplus h''$ and $h' = h'_1 \uplus \ldots \uplus h'_n$ then every tree κ'_i with root in $\langle C,s,h'_i \uplus h''\rangle$ is a subtree of κ, hence the height of κ'_i is at most n. Consequently $\text{height}(\kappa'_i) \le \text{height}(\kappa)$.

If there is a $\square$-trace following the edge, then by Definition 3.2.7, $\varphi = AG\psi$. Furthermore, since by our previous invalidity result $\langle C,s,h\rangle \not\models AG\psi$, then by Definition 2.3.2 we know that there exists a configuration $\gamma_\neg$ such that $\langle C,s,h\rangle \rightsquigarrow^* \gamma_\neg$ and $\gamma_\neg \not\models \psi$. Finally, since $h = h' \uplus h''$ and $h' = h'_1 \uplus \ldots \uplus h'_n$ then every path π starting from $\langle C,s,h\rangle$ is the same path as a path π' starting from $\langle C,s,h'_i \uplus h''\rangle$ for some i. Consequently $\text{length}(\pi') = \text{length}(\pi)$.

If there is a precondition trace following the edge then by our previous, since $(s,h') \models \Pi : \Psi(\mathbf{E})$ and $h' = h'_1 \uplus \ldots \uplus h'_n$ and $(s,h'_i \uplus h'') \models \Pi \cup \Pi'_i : \Sigma * \Sigma'_i$ for inductive rule $\Pi'_i : \Sigma'_i \Rightarrow \Psi(\mathbf{E})$, where $\Psi(\mathbf{E})$ is the predicate being unfolded, then $(s,h'_i) \in [\![\Psi]\!]^{\beta<\alpha}$. Hence the condition holds. □

Finally, global soundness is obtained by extending the properties established in Lemma 3.3.1 to paths in a pre-proof, as follows.

Theorem 3.3.2 (Soundness). *If $P \vdash C : \varphi$ is provable, then it is valid.*

Proof. Suppose for contradiction that there is a cyclic proof $\mathcal{P}$ of $J = P \vdash C : \varphi$ but J is invalid. That is, for some stack s and heap h, we have $(s,h) \models P$ but $\langle C,s,h\rangle \not\models \varphi$. Then, by local soundness of the proof rules, we can construct an infinite path $(P_i \vdash C_i : \varphi_i)_{i\ge 0}$ in $\mathcal{P}$ of invalid sequents. Since $\mathcal{P}$ is a cyclic proof, by Definition 3.2.10 there exists an infinitely progressing trace following some tail $(P_i \vdash C_i : \varphi_i)_{i\ge n}$ of the path.

If this trace is a $\square$-trace, by Definition 3.2.7 we know that for each pair of conclusion and premise (J_k, J_{k+1}), for some ψ, J_k is of the form $AG\psi$ and J_{k+1} is of the form $\square AG\psi$, or viceversa. Moreover by condition 1 of Lemma 3.3.1 we know there is a well-defined suffix of the path π' starting from $\langle J_{k+1_C}, J_{k+1_s}, J_{k+1_h}\rangle$ such that $\pi'_m \not\models \psi$ and π' is a subpath of a path π starting from $\langle J_{k_C}, J_{k_s}, J_{k_h}\rangle$. Since an infinite numer of symbolic execution rules are applied along the path (as this is the only way to discharge a sequents of the form $\_ \vdash \_ : \Box\_$), then, the length of the path decreases infinitely often. This contradicts the well-foundedness of the natural numbers, which invalidates our assumption. Consequently, J must indeed be valid.

If this trace is a $\Diamond$-trace, by Definition 3.2.7 we know that for each pair of conclusion and premise (J_k, J_{k+1}), for some ψ, J_k is of the form $EG\psi$ and J_{k+1} is of the form $\Diamond EG\psi$, or viceversa. Moreover by condition 2 of Lemma 3.3.1 we know there is a smallest finite execution subtree κ' with root in $\langle J_{k+1_C}, J_{k+1_s}, J_{k+1_h}\rangle$, each of whose leaves γ satisfies $\gamma \not\models \psi$ and κ' is a subtree of κ with root in $\langle J_{k_C}, J_{k_s}, J_{k_h}\rangle$. Since an infinite numer of symbolic execution rules are applied along the path (as this is the only way to discharge a sequents of the form $\_ \vdash \_ : \Diamond\_$), then, the heigth of the subtree decreases infintely often. This contradicts the well-foundedness of the natural numbers, which invalidates our assumption. Consequently, J must indeed be valid.

If this trace is a precondition trace, by Definition 3.2.8 we know that for each sequent J_k, there exists an inductive predicate formula $\Psi_k(\mathbf{E})$ which is a subformula of J_{k_P}. Moreover, by condition 3 of Lemma 4.2.1 we know that for each pair of conclusion and premise (J_k, J_{k+1}) the inductive predicates $\Psi_k(\mathbf{E})$ and $\Psi_{k+1}(\mathbf{E})$ have a least-fixed point interpretation which can be constructed as the union of a chain of ordinal-indexed approximations α and β respectively, where $\alpha \leq \beta$. Since by Definition 3.2.8 the trace following the path progresses infinitely often, we know that this chain of approximants decreases infinitely often, which contradicts the well-foundedness of the ordinals, in turn invalidating our assumption. Consequently J must indeed be valid.

$\Box$

Readers familiar with Hoare-style proof systems might wonder about *relative completeness* of our system, i.e., whether all valid judgements are derivable if all valid entailments between formulas are derivable. Typically, such a result might be established by showing that for any program C and temporal property φ, we can (a) express the logically weakest precondition for C to satisfy φ, say $wp(C, \varphi)$, and (b) derive $wp(C, \varphi) \vdash C : \varphi$ in our system. Relative completeness then follows from the rule of consequence, (Cons). Unfortunately, it seems certain that such weakest preconditions are not expressible in our language. For example, in [19], the multiplicative implication of separation logic, $-\!\!*$, is needed to express weakest preconditions, whereas it is not present in our language due to the problems it poses for automation (a compromise typical of most separation logic analyses). Indeed, it seems likely that we would need to extend our precondition language well beyond this, since [19] only treats termination, whereas we treat arbitrary temporal properties. Since our focus in this dissertation is on automation, we leave such an analysis to future work.

3.4 Related work

Automated verification of temporal properties of programs can be classified into two main schools: model checking and deductive verification, of which model checking has gathered more attention in recent years. Although finite-state transition systems were the focus of earlier works (e.g. [34, 86]), recent advances such as restrictions on the explored state-space [12], precondition synthesis [38], counterexample-guided refinement [41], bounded model checking [32] and automata-theoretic approaches [39] have enabled the treatment of infinite transition systems.

Here, we instead take the deductive verification approach, and therefore the main differences between our work and model checking are mainly inherited ones. On the one hand, we do not rely on program transformations or overapproximations, which could result in potentially unsound procedures; indeed, the formal soundness of our system(s) eliminates the possibility of false positives. On the other hand, we might fail to terminate, and we do not produce counterexamples in case of failure. (However, it is possible in principle that counterexamples *could* be produced from failed proofs.)

A common limitation of early proof systems for different (fragments of) temporal logics is their focus on finite state transition systems [56, 61, 13]. Contrary to these systems, our proof system can handle infinite state, non-terminating programs. In the realm of infinite state systems, previous proof systems for verifying temporal properties of arbitrary transition systems [75, 93] have shed some light on the soundness and relative completeness of deductive verification. Unfortunately, these early systems have typically relied upon complex verification conditions that raise the question of whether full automation is achievable, arguably the most cited argument against deductive verification.

Of particular relevance here are those proof systems for temporal properties based on cyclic proof. Our work can be seen as an extension of the cyclic termination proofs in [19] to arbitrary temporal properties. In [13], a procedure for the verification of CTL* properties is developed that employs a cyclic proof system for LTL as a sub-procedure. A subtle but important difference when compared to our work is the lack of cut/consequence rule (used e.g. to generalise precondition formulas or to apply intermediary lemmas). A side benefit of such restriction is the greatly simplified global soundness condition required to check the validity of their proofs.

A cyclic proof system for the verification of CTL* properties of infinite-state transition systems is presented in [93]. Focusing on generality, this system avoids considering details of state formulas and their evolution throughout program execution by assuming an oracle for a general transition system. The system relies on a soundness condition that is similar to Defn. 3.2.10, but does not track progress in the same way, imposing extra conditions on the order in which rules are applied. The success criterion for validity of a proof also presents some differences; it relies on finding ranking functions, intermediate assertions and checking for the validity of Hoare triples, and it is far from clear that such checks can be fully automated. In contrast, we rely on a relatively simple ω-regular condition, which is decidable and can be automatically checked by CYCLIST [91, 16, 22].

Chapter 4

Adaptation to LTL

In this chapter we reconfigure our cyclic proof system for CTL, described previously, to handle linear time (LTL) temporal properties of programs. Contrary to CTL, where computation is viewed as a tree of executions, LTL treats the execution of programs as a collection of traces.

The subtle difference between CTL and LTL semantics has been previously discussed in Section 2.3. A similar example of these differences can be seen in the following program.

Example 4.0.1. *Assume the following program starts its execution from an initial program state where* $y = false$.

```
while(*) {
  y:=true;
}
y:=false;
while(true) {
  y:=true;
}
```

Under these circumstances, we can demonstrate the LTL property $F(y = true \wedge XX(y = true))$ *stating that in every execution path, we will eventually reach a point where* $y = true$ *and* $y = true$ *will also hold after two computation steps. On the other hand, it is not possible to show that the analogous CTL property* $AF(y = true \wedge \Box\Box (y = true))$ *given that at every starting point of the execution of the loop at the top,* *there exists a branch (the one that exits the loop) on the execution tree on which* $\Box\Box(y = true)$ *does not hold. Proving the property* $AF(y = true \wedge \Diamond\Diamond(true))$ *can be done, but this property is too weak to convey the same meaning as the analogous LTL property.*

Even if these previous examples are handcrafted to demonstrate the difference between CTL and LTL, the reality is the view of time as linear, where each state has a single possible successor, requires a slightly different handling of the constructs of our programming language that induce the possibility of different futures. To this effect, and following the common approach of *determinising* the execution of nondeterministic while programs, we introduce *prophecy variables* (cf. [40]) that predict the outcome of nondeterministic choices in the program. Such determinisation of branching commands induces a linearisation of time, in the sense that, at any point during a program execution, there is a single possible successive state (future). Such semantics form the basis of our cyclic proof system for LTL, whose details are described in this chapter.

The rest of this chapter is structured as follows. In Section 4.1 we review the concept of prophecy variables for determinising the execution of nondeterministic programs. We then present the proof rules of our system, emphasizing the introduction of new rules to handle nondeterminism. We then introduce the concept of LTL traces and present the global soundness condition of our system based on these traces. We finish the section by presenting some examples to demonstrate the apli-cability of our proof system. Finally, in Section 4.2 we demonstrate soundness of our LTL cyclic proof system.

4.1 LTL cyclic proofs

To account for prophecy variables, we define a special set Proph of prophecy variables (disjoint from Var), and extend our memory states so that the stack component maps variables to values and prophecy variables to prophecy values defined as $s : (\mathsf{Var} \to \mathsf{Val}) \times (\mathsf{Proph} \to \mathbb{N} \cup \{\bot, ?\})$. Roughly speaking, each prophecy variable is associated with a nondeterministic branching command, where initially, all

$$\frac{[\![*_i]\!]s = ? \quad n \in \mathbb{N} \cup \{\bot\}}{\langle \textbf{while } *_i \textbf{ do } C \textbf{ od} \;;\, C', s, h\rangle \rightsquigarrow \langle C \;;\, \textbf{while } *_i \textbf{ do } C \textbf{ od}, s[*_i \mapsto n], h\rangle}$$

$$\frac{[\![*_i]\!]s = \bot}{\langle \textbf{while } *_i \textbf{ do } C \textbf{ od} \;;\, C', s, h\rangle \rightsquigarrow \langle C \;;\, \textbf{while } *_i \textbf{ do } C \textbf{ od}, s, h\rangle}$$

$$\frac{[\![*_i]\!]s = 0}{\langle \textbf{while } *_i \textbf{ do } C \textbf{ od} \;;\, C', s, h\rangle \rightsquigarrow \langle C', s[*_i \mapsto ?], h\rangle}$$

$$\frac{[\![*_i]\!]s > 0}{\langle \textbf{while } *_i \textbf{ do } C \textbf{ od}, s, h\rangle \rightsquigarrow \langle C \;;\, \textbf{while } *_i \textbf{ do } C \textbf{ od}, s[*_i \mapsto [\![*_i]\!]s - 1], h\rangle}$$

$$\frac{[\![*_i]\!]s \in \{?, 0\}}{\langle \textbf{if } *_i \textbf{ then } C \textbf{ else } C' \textbf{ fi} \;;\, C'', s, h\rangle \rightsquigarrow \langle C \;;\, C'', s[*_i \mapsto 1], h\rangle}$$

$$\frac{[\![*_i]\!]s = 1}{\langle \textbf{if } *_i \textbf{ then } C \textbf{ else } C' \textbf{ fi} \;;\, C'', s, h\rangle \rightsquigarrow \langle C' \;;\, C'', s[*_i \mapsto 0], h\rangle}$$

Figure 4.1: Small-step operational semantics for nondeterminism in LTL.

prophecy variables hold an uninitialised value (?). For `while` commands, the value of the prophecy variable determines the number of times the loop is executed; either an infinite ($\bot$) or a finite ($\mathbb{N}$) number of times. In case of `if` commands, the value of the prophecy value alternates between 0 and 1, which in turn causes the execution to follow the first or second branch. The corresponding operational semantics of our language with prophecy variables is shown in Figure 4.1.

Judgements in the LTL system are of the form $P \vdash C : \varphi$, where: P is a symbolic heap formula as per Definition 2.2.1; C is a sequence of commands as per Definition 2.1.1 (i.e. a computer program); and φ is a temporal assertion written in the LTL language described in Definition 2.3.3.

The interpretation of judgements for LTL is as follows:

Definition 4.1.1 (LTL validity)**.** An LTL judgement $P \vdash C : \varphi$ is *valid* if and only if, for all memory states (s, h) such that $s, h \vDash P$ and for all paths π starting from $\langle C, s, h\rangle$, we have $\pi \vDash \varphi$, according to Definition 2.3.4.

The proof rules for LTL judgements are shown in Figure 4.2. Roughly speaking, these are obtained from the CTL proof rules by:

1. removing the symbolic execution rules for $\Diamond$ formulas and all rules for the CTL temporal operators;
2. replacing $\Box$ by the "next" operator X in all remaining rules;
3. adding specific proof rules for determinised symbolic execution rules; and
4. adding specific proof rules for LTL temporal operators.

Similarly to Chapter 3, proofs in our LTL system are cyclic proofs, where every leaf node of the derivation tree are matched to an interior node of the proof graph. The global soundness condition for LTL, needed to qualify pre-proofs as proofs, requires modified auxiliary definitions, as described next.

Definition 4.1.2 (LTL Trace)**.** Let $(J_i = P_i \vdash C_i\colon \varphi_i)_{i\geq 0}$ be a path in a pre-proof $\mathcal{P}$. The sequence of temporal formulas $(\varphi_i)_{i\geq 0}$ is an *LTL trace* following $(J_i)_{i\geq 0}$ if there exists a ψ such that for all $i \geq 0$ the following holds:

- the formula φ_i is of the form $G\psi$ and the formula φ_{i+1} is of the form $XG\psi$, or vice versa;
- $\varphi_i = \varphi_{i+1}$ whenever J_i is the conclusion of either the consequence rule (Cons) or the (Unfold-Pre) rule.

We say that an LTL trace progresses whenever $(\varphi_i, \varphi_{i+1})$ is the temporal component of a pair of sequents which are, respectively, the conclusion and premise of a symbolic execution rule. An LTL trace is *infinitely progressing* if it progresses at infinitely many points.

As in the case of the CTL cyclic proof system, we account for precondition traces, which are the same as in Definition 3.2.8 with some minor modifications:

Definition 4.1.3 (Precondition trace)**.** Let $(J_i = P_i \vdash C_i\colon \varphi_i)_{i\geq 0}$ be a path in a pre-proof $\mathcal{P}$. The sequence of symbolic heap formulas along the path, $(\Psi_i)_{i\geq 0}$, is a *precondition trace* following that path $(J_i)_{i\geq 0}$ if:

Symbolic execution rules:

$$\frac{}{P \vdash C : \mathsf{final}}\ (\text{Final}) \qquad \frac{P \vdash C : \varphi}{P \vdash (\mathbf{skip}\ ;\ C) : X\varphi}\ (\text{Skip})$$

$$\frac{x = E[x'/x], P[x'/x] \vdash C : \varphi}{P \vdash (x := E\ ;\ C) : X\varphi}\ (\text{Assign}) \qquad \frac{x = E'[x'/x], (P * E \mapsto E')[x'/x] \vdash C : \varphi}{P * E \mapsto E' \vdash (x := [E]\ ;\ C) : X\varphi}\ (\text{Read})$$

$$\frac{P * E \mapsto E' \vdash C : \varphi}{P * E \mapsto - \vdash ([E] := E'\ ;\ C) : X\varphi}\ (\text{Write}) \qquad \frac{P \vdash C : \varphi}{P * E \mapsto - \vdash (\text{free}(E)\ ;\ C) : X\varphi}\ (\text{Free})$$

$$\frac{P[x'/x] * x \mapsto v \vdash C : \varphi}{P \vdash (x := alloc()\ ;\ C) : X\varphi}\ v \text{ fresh (Alloc□)} \qquad \frac{B, P \vdash C_1\ ;\ C_3 : \varphi \quad \neg B, P \vdash C_2\ ;\ C_3 : \varphi}{P \vdash (\mathbf{if}\ B\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}\ ;\ C_3) : \bigcirc\varphi}\ (\text{If})$$

$$\frac{B, P \vdash (C_1\ ;\ \mathbf{while}\ B\ \mathbf{do}\ C_1\ \mathbf{od}\ ;\ C_2) : \varphi \quad \neg B, P \vdash C_2 : \varphi}{P \vdash (\mathbf{while}\ B\ \mathbf{do}\ C_1\ \mathbf{od}\ ;\ C_2) : \bigcirc\varphi}\ (\text{Wh}) \qquad \frac{i = 1, P \vdash C_1\ ;\ C_3 : \varphi}{i = 0, P \vdash (\mathbf{if} * \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}\ ;\ C_3) : X\varphi}\ (\text{If-Z})$$

$$\frac{i = 0, P \vdash C_2\ ;\ C_3 : \varphi}{i = 1, P \vdash (\mathbf{if} * \mathbf{then}\ C_1\ \mathbf{else}\ C_2\ \mathbf{fi}\ ;\ C_3) : X\varphi}\ (\text{If-N}) \qquad \frac{i = \bot, P \vdash (C_1\ ;\ \mathbf{while} *_i \mathbf{do}\ C_1\ \mathbf{od}\ ;\ C_2) : \varphi}{i = \bot, P \vdash (\mathbf{while} *_i \mathbf{do}\ C_1\ \mathbf{od}\ ;\ C_2) : X\varphi}\ (\text{Wh-}\bot)$$

$$\frac{i = ?, P \vdash C_2 : \varphi}{i = 0, P \vdash (\mathbf{while} *_i \mathbf{do}\ C_1\ \mathbf{od}\ ;\ C_2) : X\varphi}\ (\text{Wh-Z}) \qquad \frac{i = n-1, P \vdash (C_1\ ;\ \mathbf{while} *_i \mathbf{do}\ C_1\ \mathbf{od}\ ;\ C_2) : \varphi \quad n > 0}{i = n, P \vdash (\mathbf{while} *_i \mathbf{do}\ C_1\ \mathbf{od}\ ;\ C_2) : X\varphi}\ (\text{Wh-N})$$

$$\frac{i = \bot, P \vdash (C_1\ ;\ \mathbf{while} *_i \mathbf{do}\ C_1\ \mathbf{od}) : \varphi \quad i = n, P \vdash (C_1\ ;\ \mathbf{while} *_i \mathbf{do}\ C_1\ \mathbf{od}) : \varphi}{i = ?, P \vdash (\mathbf{while} *_i \mathbf{do}\ C_1\ \mathbf{od}\ ;\ C_2) : X\varphi}\ (\text{Det.})$$

Faulting execution rules:

$$\frac{P * E \mapsto \mathsf{nil} \not\models \bot}{P \vdash (x := [E]\ ;\ C) : X\mathsf{error}}\ (\text{R}\bot) \qquad \frac{P * E \mapsto \mathsf{nil} \not\models \bot}{P \vdash ([E] := E'\ ;\ C) : X\mathsf{error}}\ (\text{W}\bot) \qquad \frac{P * E \mapsto \mathsf{nil} \not\models \bot}{P \vdash (\text{free}(E)\ ;\ C) : X\mathsf{error}}\ (\text{Free}\bot)$$

Logical rules:

$$\frac{P \models Q}{P \vdash C : Q}\ (\text{Check}) \qquad \frac{}{\bot \vdash C : \varphi}\ (\text{Ex.Falso}) \qquad \frac{\Omega_1 \vdash C : \varphi \quad \Omega_2 \vdash C : \varphi}{\Omega_1 \vee \Omega_2 \vdash C : \varphi}\ (\text{Split})$$

$$\frac{P \vdash C : \varphi \quad x \notin \text{vars}(C)}{P[E/x] \vdash C : \varphi[E/x]}\ (\text{Subst}) \qquad \frac{P \vdash C : \varphi_1 \quad P \vdash C : \varphi_2}{P \vdash C : \varphi_1 \wedge \varphi_2}\ (\text{Conj}) \qquad \frac{P \vdash C : \varphi_i \quad i \in \{1,2\}}{P \vdash C : \varphi_1 \vee \varphi_2}\ (\vee)$$

$$\frac{P \vdash C : \varphi \vee XF\varphi}{P \vdash C : F\varphi}\ (\text{F}) \qquad \frac{P \vdash C : \varphi \quad P \vdash C : XG\varphi}{P \vdash C : G\varphi}\ (\text{G}) \qquad \frac{P \vdash C : \psi \vee (\varphi \wedge X(\varphi U \psi))}{P \vdash C : (\varphi U \psi)}\ (\text{U})$$

$$\frac{P \vdash Q \quad Q \vdash C : \psi \quad \psi \vdash \varphi}{P \vdash C : \varphi}\ (\text{Cons})$$

$$\frac{(\Pi \cup \Pi'_i : \Sigma * \Sigma'_i \vdash C : \varphi)_{1 \le i \le k}}{\Pi : \Psi(\mathbf{E}) * \Sigma \vdash C : \varphi} \left(\begin{array}{c} \Pi_1 : \Sigma_1 \Rightarrow \Psi(\mathbf{E}_1), \ldots, \Pi_k : \Sigma_k \Rightarrow \Psi(\mathbf{E}_k) \\ \Pi'_i : \Sigma'_i = \Pi_i : \Sigma_i \text{ with existential variables freshened and} \\ \text{arguments } \mathbf{E} \text{ substituted for parameters } \mathbf{E}_i \end{array}\right)\ (\text{Unfold-Pre})$$

Figure 4.2: LTL proof rules.

(i) Whenever J_i is the conclusion of the (Unfold-Pre) rule, the predicate $\Psi_i(\mathbf{E})$ is the predicate in the spatial formula of P_i being unfolded and $\Psi_{i+1} = \Psi'(\mathbf{E})$, where $\Psi'(\mathbf{E})$ is obtained in the premise J_{i+1} by unfolding $\Psi(\mathbf{E})$; and

(ii) $\Psi_i = \Psi_{i+1}$ (modulo any rewriting done by rules (Assign), (Read), (Alloc), (Subst)) for all other rules.

We say that a precondition trace *progresses* whenever (Unfold-Pre) is applied. A precondition trace is *infinitely progressing* if it progresses at infinitely many points.

Definition 4.1.4 (LTL proof)**.** A pre-proof $\mathcal{P}$ is an *LTL proof* if, for every infinite path $(J_i)_{i\geq 0}$ in $\mathcal{P}$, there is an infinitely progressing LTL or precondition trace following some tail $(J_i)_{i\geq n}$ of the path.

We revisit Example 2.3.5 to illustrate the formation of backlinks and the global soundness condition for the LTL system.

Example 4.1.5. *Consider the following (labelled) program:*

```
1: while(*i) {
2:   x:=true;
   }
3: x:=false;
4: x:=true;
5: while (x=x) {
6:   skip;
   }
```

Based on the semantics of nondeterminism described in Figure 4.1, there are three possible program behaviours:

1. *The program never leaves the nondeterministic loop. Such program behaviour is the result of assigning the prophecy value $\bot$ to the prophecy variable that is associated to the nondeterministic loop.*

2. *The program executes the nondeterministic loop a number of times (say n) before exiting. This program behaviour arises from assigning an arbitrary prophecy value $n \in \mathbb{N} \setminus \{0\}$ to the prophecy variable; and*

3. *The program never enters the loop, as the result of assigning* 0 *to the prophecy variable associated with the nondeterministic loop.*

Under these conditions, it can be seen that all program executions satisfy the LTL property $FG(x = true)$, that is, for every program execution, x will eventually become true and remain unchanged from that point onwards. Figure 4.1.5 shows the proof of this property using the cyclic proof system described in this chapter.

The proof graph at the bottom of the figure is the result of the derivations where the prophecy variable i is assigned a prophecy value $\perp$ by the (Det) proof rule; as such, this proof graph corresponds to those program executions that never enter the loop. Note that along the infinite path induced by the backlink, all sequents have a temporal formula of the form $G\psi$ or $XG\psi$, where $\psi = x = true$, forming a path ($G\psi$,$XG\psi$,$G\psi$,$XG\psi$,$G\psi$,...) along the path. Moreover, due to the application of symbolic execution rules (Wh-$\perp$) and (Assign) along the path, the trace progresses infinitely often.

The proof graph at the top of the figure is the result of the derivations where the prophecy variable i is assigned a prophecy value $n \in \mathbb{N}$ by the (Det) proof rule; as such, this proof graph corresponds to those program executions that execute the body of the loop and then exit the loop after a number of iterations.

Taking advantage of the use of cyclic proofs, we can denote the prophecy variable as an inductive predicate $N(i)$ that characterises the natural numbers. We can then apply a case-split rule on the inductive predicate $N(i)$. The case where $n \neq 0$ is handled on the right hand side of the proof graph whereas the $n = 0$ case is handled on the left hand side of the proof graph.

Note that along the infinite path induced by the backlink on the right hand side, all sequents have a precondition formula that contains the inductive predicate $N(i)$. Moreover, along the path there are infinitely many applications of the proof rule (Unfold-Pre), and, therefore there is an infinite progressing precondition trace along the path as per Definition 3.2.8. Similarly, along the infinite path induced by the backlink on the left hand side proof graph, all sequents have a temporal formula of the form $G\psi$ or $XG\psi$, where $\psi = x = true$, forming an LTL trace of the form

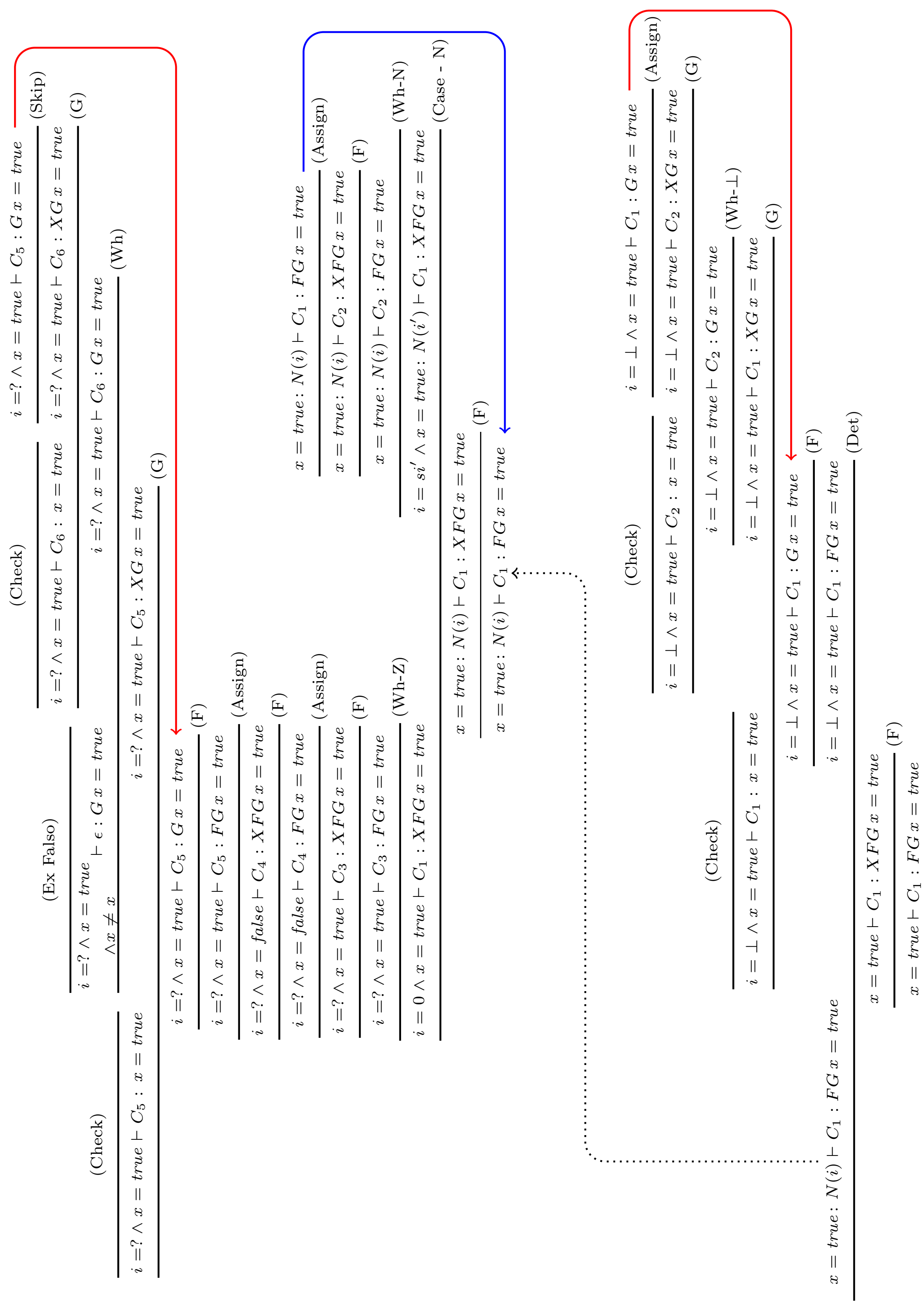


Figure 4.3: LTL example. The proof tree at the bottom corresponds to those program executions that never exit the first loop. The proof tree at the top (connected to the left-most sequent of the bottom tree) corresponds to those program executions that execute the first loop a finite number of times and eventually exit it.

($G\psi$,$XG\psi$,$G\psi$,$XG\psi$,$G\psi$,...) along the path. Moreover, due to the application of rules (Wh) and $(Skip)$ along the path, the LTL trace progresses infinitely often.

As established in Definition 4.1.4, since along every infinite path in the proof graph there is an infinitely progressing LTL or precondition trace following some tail of the path, then our pre-proof is a valid proof. Strictly speaking, there is yet another infinite path in our proof graph. This infinite path results from the combination of a finite number of "iterations" along the cycle with a precondition trace, followed by an infinite number of "iterations" along the cycle with an LTL trace in the top proof graph. This infinite path is nevertheless handled by our previous cases since its tail corresponds to an infinite path along which there is an infinitely progressing LTL trace.

4.2 Soundness of LTL system

In this section we show that our LTL proof system is sound. We first show local soundness of the proof rules along with the trace properties that are maintained by all derivation rules, as established in Definitions 3.2.7 and 3.2.8. For each proof rule, we show: (1) soundness from conclusion to premises by assuming that the conclusion is invalid (by Definition 4.1.1) and proceeding to establish the invalidity (of at least one) of the premise(s); and (2) the trace property regarding the length of paths is preserved by each rule by either maintaining or reducing the length of the path. In the case of the axioms, we show that the conclusion is a tautology. Finally we show the global soundness of our system, essentially, by extending the properties established for local soundness to paths in a pre-proof.

Lemma 4.2.1. *Let $J = (P \vdash C\colon \varphi)$ be the conclusion of a proof rule R. If J is invalid under program state (s,h), then there exists a premise of the rule $J' = (P' \vdash C' : \varphi')$ and a memory state (s',h') such that*

1. *the sequent J' is not valid under (s',h')*

2. *if there is an LTL trace (φ,φ') following the edge (J,J') then, letting ψ be the unique formula given by Definition 4.1.2, there exists a k such that $\pi_k \nvDash \psi$,*

and the finite path $\pi' \overset{\text{def}}{=} \langle C', s', h' \rangle \ldots \pi[k]$ *is a subpath of* $\pi \overset{\text{def}}{=} \langle C, s, h \rangle \ldots \pi[k]$. *Therefore* $\text{length}(\pi') \leq \text{length}(\pi)$. *Moreover,* $\text{length}(\pi') < \text{length}(\pi)$ *when* R *is a symbolic execution rule.*

3. *if there is a precondition trace* $(\Psi(\mathbf{E}), \Psi'(\mathbf{E}))$ *following the edge* (J, J') *then letting* $\alpha(\beta)$ *be the least approximant for which the inductive predicate* $\Psi(\mathbf{E})(\Psi'(\mathbf{E}))$ *is interpreted (i.e.* $(s,h) \models \Psi^{\alpha}(\Psi'^{\beta})$*), then the following relation holds and it is well-defined:* $\beta \leq \alpha$. *Moreover* $\beta < \alpha$ *when* R *is the* $($*Unfold-Pre*$)$ *rule.*

Proof. We proceed by case analysis of the proof rule R, only showing the proof of the rules specific to LTL; all the remaining cases are similar to the local soundness proof of Chapter 3.

Soundness of F

$$\frac{P \vdash C : \varphi \vee XF\varphi}{P \vdash C : F\varphi} \text{ (F)}$$

Assume the conclusion of the rule is invalid. Pick an arbitrary program state s,h such that $(s,h) \models P$ but for some execution path π starting from $\langle C, s, h \rangle$, the path π does not satisfy the temporal property $F\varphi$. By Definition 2.3.4, if $\pi \not\models F\varphi$ then for all $k \geq 0, \pi_k \not\models \varphi$.

Under this assumption, since there exists a path π starting from $\langle C, s, h \rangle$ where for all $k \geq 0, \pi_k \not\models \varphi$, then we know that, in particular $(i)\pi_0 \not\models \varphi$. Moreover, it is also the case that, in particular, $\pi_1 \not\models F\varphi$. Hence $(ii)\pi_0 \not\models XF\varphi$. Consequently, by our assumption $(s,h) \models P$ and results (i) and (ii), the premise of the rule is invalid.

Given the temporal property with formula $(X)F\varphi$, there cannot be an LTL-trace following the path.

If there is a precondition trace $(\Psi(\mathbf{E}), \Psi'(\mathbf{E}))$ following the edge, where $(s,h) \models \Psi^{\alpha}(\mathbf{E})$ and $(s', h') \models \Psi'^{\beta}(\mathbf{E})$, then, since the precondition for both conclusion and premise is the same, then the inductive predicates $\Psi(\mathbf{E})$ and $\Psi'(\mathbf{E})$ in both conclusion and premise are interpreted by the same least approximant. Hence $\alpha = \beta$.

Soundness of G

$$\frac{P \vdash C : \varphi \quad P \vdash C : XG\varphi}{P \vdash C : G\varphi} \text{ (G)}$$

Assume the conclusion of the rule is invalid. Pick an arbitrary program state s,h such that $(s,h) \vDash P$ but for some execution path π starting from $\langle C,s,h\rangle$, the path π does not satisfy the temporal property $G\varphi$. By Definition 2.3.4, if $\pi \nvDash G\varphi$ then there exists $k \geq 0$ such that $\pi_k \nvDash \varphi$.

Case $k = 0$: It follows from our assumptions that $(i)\pi \nvDash \varphi$, where $\pi[0] = \langle C,s,h\rangle$. Hence the left hand premise is invalid.

Case $k > 0$: Let $\gamma = \pi[1]$ so that $\langle C,s,h\rangle \rightsquigarrow \gamma \rightsquigarrow \ldots$. By our assumption we know that there exists $k > 0$ such that $\pi_k \nvDash \varphi$, then we know that for all paths π' starting from γ, $\pi' \nvDash G\varphi$. Consequently, $(ii)\pi \nvDash XG\varphi$. Hence the right hand premise is invalid.

Therefore, letting $s' = s$ and $h' = h$, by our assumption $(s',h') \vDash P$ and results (i) and (ii), then either one of the premises of the rule is invalid.

If there is an LTL-trace following the edge, then by Definition 4.1.2, said trace must be following the right hand edge and $\psi = \varphi$. Furthermore, since by our previous invalidity result $\pi \nvDash G\psi$, then by Definition 2.3.4 we know that there exists a $k > 0$ such that $\pi_k \nvDash \psi$. Finally, since by construction $\langle C,s,h\rangle = \langle C,s',h'\rangle$, then every path τ starting from $\langle C,s,h\rangle$ is the same path τ' as that starting from $\langle C,s',h'\rangle$. Consequently $\text{length}(\tau') = \text{length}(\tau)$.

If there is a precondition trace $(\Psi(\mathbf{E}),\Psi'(\mathbf{E}))$ following the edge, where $(s,h) \vDash \Psi^{\alpha}(\mathbf{E})$ and $(s',h') \vDash \Psi'^{\beta}(\mathbf{E})$, then, since the precondition for both conclusion and premise is the same, then the inductive predicates $\Psi(\mathbf{E})$ and $\Psi'(\mathbf{E})$ in both conclusion and premise are interpreted by the same least approximant. Hence $\alpha = \beta$.

Soundness of U

$$\frac{P \vdash C : \psi \vee (\varphi \wedge X(\varphi U \psi))}{P \vdash C : (\varphi U \psi)} \text{ (U)}$$

Assume the conclusion of the rule is invalid. Pick an arbitrary program state s,h such that $(s,h) \vDash P$ but for some execution path π starting from $\langle C,s,h\rangle$, the path π does not satisfy the temporal property $(\varphi U \psi)$. By Definition 2.3.4, if $\pi \nvDash (\varphi U \psi)$

then either for all $i \geq 0, \pi_i \nvDash \psi$ (i.e. the property ψ is never satisfied along the path) or there exists $j : 0 \leq j \leq i$ such that $\pi_j \nvDash \varphi$ (i.e. the property ψ is eventually satisfied, but there exists a configuration in the path that does not satisfy φ).

Case for all $i \geq 0, \pi_i \nvDash \psi$: By construction $\langle C, s, h \rangle = \pi[0]$. Then it follows from our assumption that there exists a path π starting from $\langle C, s, h \rangle$ such that $(i)\pi \nvDash \psi$.

Case for all $i \geq 0$, there exists $j : 0 \leq j \leq i$ such that $\pi_j \nvDash \varphi$ Assume the property ψ is eventually satisfied in suffix $i > 0$ of π such that $\pi_i \vDash \psi$. Then, since $\pi \nvDash (\varphi U \psi)$, then it must be the case that there exists $j : 0 \leq j \leq i$ such that $\pi_j \nvDash \varphi$.

Subcase $j = 0$: since by construction $\langle C, s, h \rangle = \pi[0]$ then it must be the case that there is path π starting from $\langle C, s, h \rangle$ such that $(ii)\pi \nvDash \varphi$.

Subcase $j > 0$: Let $\gamma = \pi[1]$ so that $\langle C, s, h \rangle \rightsquigarrow \gamma \rightsquigarrow \ldots$. Then we know that there exists a path π' starting from γ where for all $i' \geq 0$ either $\pi'_{i'} \nvDash \psi$ or there exists $j' : 0 \leq j' < i'$ such that $\pi_{j'} \nvDash \varphi$. In other words $(iii)\pi \nvDash X(\varphi U \psi)$. Consequently, by our assumption $(s, h) \vDash P$ and results (i), (ii) and (iii) the premise of the rule is invalid.

Given the temporal property of the sequents, there cannot be an LTL-trace following the path.

If there is a precondition trace $(\Psi(\mathbf{E}), \Psi'(\mathbf{E}))$ following the edge, where $(s, h) \vDash \Psi^{\alpha}(\mathbf{E})$ and $(s', h') \vDash \Psi'^{\beta}(\mathbf{E})$, then, since the precondition for both conclusion and premise is the same, then the inductive predicates $\Psi(\mathbf{E})$ and $\Psi'(\mathbf{E})$ in both conclusion and premise are interpreted by the same least approximant. Hence $\alpha = \beta$.

Soundness of Det.

$$\frac{i = \bot, P \vdash (C_1 \,;\, \textbf{while} *_i \textbf{do}\, C_1\, \textbf{od}) : \varphi \qquad i = n, P \vdash (C_1 \,;\, \textbf{while} *_i \textbf{do}\, C_1\, \textbf{od}) : \varphi}{i = ?, P \vdash (\textbf{while} *_i \textbf{do}\, C_1\, \textbf{od} \,;\, C_2) : X\varphi} \text{(Det.)}$$

Assume the conclusion of the rule is invalid. Pick an arbitrary program state s, h so that $(s, h) \vDash i = ?, P$ but for some execution path π starting from $\langle \textbf{while} *_i \textbf{do}\, C_1\, \textbf{od} \,;\, C_2 \,;\, C_2, s, h \rangle$, the path π does not satisfy the temporal property $X\varphi$. By Definition 2.3.4 if $\pi \nvDash X\varphi$ then $\pi_1 \nvDash \varphi$. By the operational semantics we know there are two possible execution paths: $\pi[1] = \langle C_1 \,;\, \textbf{while} *_i \textbf{do} \, C_1 \, \textbf{od}, s[*_i \mapsto \bot], h\rangle$ or $\pi[1] = \langle C_1 \,;\, \textbf{while} *_i \textbf{do} \, C_1 \, \textbf{od}, s[*_i \mapsto n], h\rangle$. We show the details of the foremost while omitting the latter due to their similarity.

Case $\gamma' = \langle C_1 \,;\, \textbf{while} *_i \textbf{do} \, C_1 \, \textbf{od} \,;\, C_2, s[*_i \mapsto \bot], h\rangle$: By construction we know that $[\![*_i]\!]s[*_i \mapsto \bot] = \bot$, moreover, by our assumption that $(s,h) \vDash i = ?, P$ we know that $(s[*_i \mapsto \bot], h), \vDash i = \bot, P$. Since by our previous finding, there exists a path $\pi' = \pi_1$ starting from γ' such that $\pi' \nvDash \varphi$ then the left-most premise of the rule is invalid.

If there is an LTL trace following the edge, then by Definition 4.1.2, $\varphi = G\psi$. Furthermore, since by our previous result $\pi_1 \nvDash G\psi$, then by Definition 2.3.4 we know that there exists $k \geq 1$ such that $\pi_k \nvDash \psi$. Finally, since by the operational semantics $\langle \textbf{while} *_i \textbf{do} \, C_1 \, \textbf{od} \,;\, C_2 \,;\, C_2, s, h\rangle \rightsquigarrow \langle C_1 \,;\, \textbf{while} *_i \textbf{do} \, C_1 \, \textbf{od}, s[*_i \mapsto \bot], h\rangle$, then every execution path τ' starting from $\langle C_1 \,;\, \textbf{while} *_i \textbf{do} \, C_1 \, \textbf{od}, s[*_i \mapsto \bot], h\rangle$ is a subpath of an execution path τ starting from $\langle \textbf{while} *_i \textbf{do} \, C_1 \, \textbf{od} \,;\, C_2 \,;\, C_2, s, h\rangle$. Consequently $\text{length}(\tau') < \text{length}(\tau)$.

If there is a precondition trace $(\Psi(\mathbf{E}), \Psi'(\mathbf{E}))$ following the edge, where $(s,h) \vDash \Psi^{\alpha}(\mathbf{E})$ and $(s',h') \vDash \Psi'^{\beta}(\mathbf{E})$, then, since the precondition for both conclusion and premise is the same (modulo the subformula concerning prophecy variables, which are not part of inductive predicate definitions), then the inductive predicates $\Psi(\mathbf{E})$ and $\Psi'(\mathbf{E})$ in both conclusion and premise are interpreted by the same least approximant. Hence $\alpha = \beta$.

Soundness of Wh-$\bot$

$$\frac{i = \bot, P \vdash (C_1 \,;\, \textbf{while} *_i \textbf{do} \, C_1 \, \textbf{od} \,;\, C_2) : \varphi}{i = \bot, P \vdash (\textbf{while} *_i \textbf{do} \, C_1 \, \textbf{od} \,;\, C_2) : X\varphi} \text{Wh-}\bot$$

Assume the conclusion of the rule is invalid. Pick an arbitrary program state s,h such that $(s,h) \vDash i = \bot, P$ but there exists a path π starting from the program configuration $\gamma = \langle \textbf{while} *_i \textbf{do} \, C_1 \, \textbf{od} \,;\, C_2, s, h\rangle$ such that the path π does not satisfy the temporal property $X\varphi$. By Definition 2.3.4 if $\pi \nvDash X\varphi$ then $\pi_1 \nvDash \varphi$. By the operational semantics we know there is a single possible path with root in γ where $\pi[1] = \langle C_1 \,;\, \textbf{while} *_i \textbf{do} \, C_1 \, \textbf{od} \,;\, C_2, s, h\rangle$. Since by our assumption $(s,h) \vDash i = \bot, P$

but there exists a path $\pi' = \pi_1$ starting from $\langle C_1 \text{ ; } \mathbf{while} *_i \mathbf{do}\, C_1 \,\mathbf{od}, s, h\rangle$ such that $\pi' \nvDash \varphi$ then the premise of the rule must be invalid.

If there is an LTL trace following the edge, then by Definition 4.1.2, $\varphi = G\psi$. Furthermore, since by our previous result $\pi_1 \nvDash G\psi$, then by Definition 2.3.4 we know that there exists $k \geq 1$ such that $\pi_k \nvDash \psi$. Finally, since by the operational semantics $\langle \mathbf{while} *_i \mathbf{do}\, C_1 \,\mathbf{od} \text{ ; } C_2, s, h\rangle \leadsto \langle C_1 \text{ ; } \mathbf{while} *_i \mathbf{do}\, C_1 \,\mathbf{od}, s, h\rangle$, then every execution path τ' starting from $\langle C_1 \text{ ; } \mathbf{while} *_i \mathbf{do}\, C_1 \,\mathbf{od}, s, h\rangle$ is a subpath of an execution path τ starting from $\langle \mathbf{while} *_i \mathbf{do}\, C_1 \,\mathbf{od} \text{ ; } C_2, s, h\rangle$. Consequently length$(\tau') <$ length(τ).

If there is a precondition trace $(\Psi(\mathbf{E}), \Psi'(\mathbf{E}))$ following the edge, where $(s,h) \vDash \Psi^{\alpha}(\mathbf{E})$ and $(s',h') \vDash \Psi'^{\beta}(\mathbf{E})$, then, since the precondition for both conclusion and premise is the same, then the inductive predicates $\Psi(\mathbf{E})$ and $\Psi'(\mathbf{E})$ in both conclusion and premise are interpreted by the same least approximant. Hence $\alpha = \beta$.

Soundness of Wh-N

$$\frac{i = n-1, P \vdash (C_1 \text{ ; } \mathbf{while} *_i \mathbf{do}\, C_1 \,\mathbf{od} \text{ ; } C_2) : \varphi \quad n > 0}{i = n, P \vdash (\mathbf{while} *_i \mathbf{do}\, C_1 \,\mathbf{od} \text{ ; } C_2) : X\varphi} \text{ (Wh-N)}$$

Assume the conclusion of the rule is invalid. Pick an arbitrary program state s,h such that $(s,h) \vDash i = n, P$ but there exists a path π starting from the program configuration $\gamma = \langle \mathbf{while} *_i \mathbf{do}\, C_1 \,\mathbf{od} \text{ ; } C_2, s, h\rangle$, such that the path π does not satisfy the temporal property $X\varphi$. By Definition 2.3.4 if $\pi \nvDash X\varphi$ then $\pi_1 \nvDash \varphi$. By the operational semantics we know there is a single possible path with root in γ where $\pi[1] = \langle C_1 \text{ ; } \mathbf{while} *_i \mathbf{do}\, C_1 \,\mathbf{od} \text{ ; } C_2, s[*_i \mapsto n-1], h\rangle$. By construction, we know that $(s[*_i \mapsto n-1], h) \vDash i = n-1$. Moreover, by our assumption that $(s,h) \vDash i = n, P$ then $(s[*_i \mapsto n-1], h) \vDash i = n-1, P$. On the other hand, since $\pi_1 \nvDash \varphi$ then the premise of the rule must be invalid.

If there is an LTL trace following the edge, then by Definition 4.1.2, $\varphi = G\psi$. Furthermore, since by our previous result $\pi_1 \nvDash G\psi$, then by Definition 2.3.4 we know that there exists $k \geq 1$ such that $\pi_k \nvDash \psi$. Finally, since by the operational semantics $\langle \textbf{while} *_i \textbf{do} \, C_1 \, \textbf{od} \, ; C_2, s, h\rangle \rightsquigarrow \langle C_1 \, ; \textbf{while} *_i \textbf{do} \, C_1 \, \textbf{od} \, ; C_2, s[*_i \mapsto n-1], h\rangle$, then every path τ' starting from $\langle C_1 \, ; \textbf{while} *_i \textbf{do} \, C_1 \, \textbf{od} \, ; C_2, s[*_i \mapsto n-1], h\rangle$ is a subpath of an execution path τ starting from $\langle \textbf{while} *_i \textbf{do} \, C_1 \, \textbf{od} \, ; C_2, s, h\rangle$. Consequently length$(\tau')$ < length(τ).

If there is a precondition trace $(\Psi(\mathbf{E}), \Psi'(\mathbf{E}))$ following the edge, where $(s,h) \vDash \Psi^{\alpha}(\mathbf{E})$ and $(s',h') \vDash \Psi'^{\beta}(\mathbf{E})$, then, since the precondition for both conclusion and premise is the same (modulo the subformula concerning prophecy variables, which are not part of inductive predicate definitions), then the inductive predicates $\Psi(\mathbf{E})$ and $\Psi'(\mathbf{E})$ in both conclusion and premise are interpreted by the same least approximant. Hence $\alpha = \beta$.

Soundness of Wh-Z

$$\frac{i = ?, P \vdash C_2 : \varphi}{i = 0, P \vdash (\textbf{while} *_i \textbf{do} \, C_1 \, \textbf{od} \, ; C_2) : X\varphi} \text{(Wh-Z)}$$

Assume the conclusion of the rule is invalid. Pick an arbitrary program state s,h such that $(s,h) \vDash i = 0, P$ but there exists an execution path π starting from the program configuration $\gamma = \langle \textbf{while} *_i \textbf{do} \, C_1 \, \textbf{od} \, ; C_2, s, h\rangle$ such that the path π does not satisfy the temporal property $X\varphi$. By Definition 2.3.4 if $\pi \nvDash X\varphi$ then $\pi_1 \nvDash \varphi$. By the operational semantics we know there is a single possible path with root γ where $\pi[1] = \langle C_2, s[*_i \mapsto ?], h\rangle$. By construction $(s[*_i \mapsto ?], h) \vDash i = ?$. Moreover, by our assumption that $(s,h) \vDash i = 0, P$ then $(s[*_i \mapsto ?], h) \vDash i = ?, P$. On the other hand, since $\pi_1 \nvDash \varphi$ then the premise of the rule must be invalid.

If there is an LTL trace following the edge, then by Definition 4.1.2, $\varphi = G\psi$. Furthermore, since by our previous result $\pi_1 \nvDash G\psi$, then by Definition 2.3.4 we know that there exists $k \geq 1$ such that $\pi_k \nvDash \psi$. Finally, since by the operational semantics $\langle \textbf{while} *_i \textbf{do} \, C_1 \, \textbf{od} \, ; C_2, s, h\rangle \rightsquigarrow \langle C_2, s[*_i \mapsto ?], h\rangle$, then every execution path τ' starting from $\langle C_2, s[*_i \mapsto ?], h\rangle$ is a subpath of an execution path τ starting from $\langle \textbf{while} *_i \textbf{do} \, C_1 \, \textbf{od} \, ; C_2, s, h\rangle$. Consequently length$(\tau')$ < length(τ).

If there is a precondition trace $(\Psi(\mathbf{E}), \Psi'(\mathbf{E}))$ following the edge, where $(s,h) \vDash \Psi^{\alpha}(\mathbf{E})$ and $(s',h') \vDash \Psi'^{\beta}(\mathbf{E})$, then, since the precondition for both conclusion and premise is the same (modulo the subformula concerning prophecy variables, which are not part of inductive predicate definitions), then the inductive predicates $\Psi(\mathbf{E})$ and $\Psi'(\mathbf{E})$ in both conclusion and premise are interpreted by the same least approximant. Hence $\alpha = \beta$.

Soundness of If-Z

$$\frac{i = 1, P \vdash C_1 \ ; C_3 : \varphi}{i = 0, P \vdash (\textbf{if} * \textbf{then}\, C_1 \,\textbf{else}\, C_2 \,\textbf{fi} \ ; C_3) : X\varphi} \ \text{(If-Z)}$$

Assume the conclusion of the rule is invalid. Pick an arbitrary program state s,h such that $(s,h) \vDash i = 0, P$ but there exists a path π starting from the program configuration $\gamma = \langle \textbf{if} * \textbf{then}\, C_1 \,\textbf{else}\, C_2 \,\textbf{fi} \ ; C_3, s, h\rangle$, such that the path π does not satisfy the temporal property $X\varphi$. By Definition 2.3.4 if $\pi \nvDash X\varphi$ then $\pi_1 \nvDash \varphi$. By the operational semantics we know there is a single possible path with root in γ where $\pi[1] = \langle C_1 \ ; C_3, s[*_i \mapsto 1], h\rangle$. By construction, we know that $(s[*_i \mapsto 1], h) \vDash i = 1$. Moreover, by our assumption that $(s,h) \vDash i = 0, P$ then $(s[*_i \mapsto 1], h) \vDash i = 1, P$. On the other hand, since $\pi_1 \nvDash \varphi$ then the premise of the rule must be invalid.

If there is an LTL trace following the edge, then by Definition 4.1.2, $\varphi = G\psi$. Furthermore, since by our previous result $\pi_1 \nvDash G\psi$, then by Definition 2.3.4 we know that there exists $k \geq 1$ such that $\pi_k \nvDash \psi$. Finally, since by the operational semantics $\langle \textbf{if} * \textbf{then}\, C_1 \,\textbf{else}\, C_2 \,\textbf{fi} \ ; C_3, s, h\rangle \rightsquigarrow \langle C_1 \ ; C_3, s[*_i \mapsto 1], h\rangle$, then every execution path τ' starting from $\langle C_1 \ ; C_3, s[*_i \mapsto 1], h\rangle$ is a subpath of an execution path τ starting from $\langle \textbf{if} * \textbf{then}\, C_1 \,\textbf{else}\, C_2 \,\textbf{fi} \ ; C_3, s, h\rangle$. Consequently $\text{length}(\tau') < \text{length}(\tau)$.

If there is a precondition trace $(\Psi(\mathbf{E}), \Psi'(\mathbf{E}))$ following the edge, where $(s,h) \vDash \Psi^{\alpha}(\mathbf{E})$ and $(s',h') \vDash \Psi'^{\beta}(\mathbf{E})$, then, since the precondition for both conclusion and premise is the same (modulo the subformula concerning prophecy variables, which are not part of inductive predicate definitions), then the inductive predicates $\Psi(\mathbf{E})$ and $\Psi'(\mathbf{E})$ in both conclusion and premise are interpreted by the same least approximant. Hence $\alpha = \beta$.

Soundness of If-N

$$\frac{i = 0, P \vdash C_2\ ;\ C_3 : \varphi}{i = 1, P \vdash (\mathbf{if} * \mathbf{then}\, C_1\ \mathbf{else}\, C_2\ \mathbf{fi}\ ;\ C_3) : X\varphi}\ \text{(If-N)}$$

Assume the conclusion of the rule is invalid. Pick an arbitrary program state s,h such that $(s,h) \vDash i = 1, P$ but there exists a path π starting from the program configuration $\gamma = \langle \mathbf{if} * \mathbf{then}\, C_1\ \mathbf{else}\, C_2\ \mathbf{fi}\ ;\ C_3, s, h\rangle$, such that the path π does not satisfy the temporal property $X\varphi$. By Definition 2.3.4 if $\pi \nvDash X\varphi$ then $\pi_1 \nvDash \varphi$. By the operational semantics we know there is a single possible path with root in γ where $\pi[1] = \langle C_2\ ;\ C_3, s[*_i \mapsto 0], h\rangle$. By construction, we know that $(s[*_i \mapsto 0], h) \vDash i = 0$. Moreover, by our assumption that $(s,h) \vDash i = 1, P$ then $(s[*_i \mapsto 0], h) \vDash i = 0, P$. On the other hand, since $\pi_1 \nvDash \varphi$ then the premise of the rule must be invalid.

If there is an LTL trace following the edge, then by Definition 4.1.2, $\varphi = G\psi$. Furthermore, since by our previous result $\pi_1 \nvDash G\psi$, then by Definition 2.3.4 we know that there exists $k \geq 1$ such that $\pi_k \nvDash \psi$. Finally, since by the operational semantics $\langle \mathbf{if} * \mathbf{then}\, C_1\ \mathbf{else}\, C_2\ \mathbf{fi}\ ;\ C_3, s, h\rangle \rightsquigarrow \langle C_2\ ;\ C_3, s[*_i \mapsto 0], h\rangle$, then every execution path τ' starting from $\langle C_2\ ;\ C_3, s[*_i \mapsto 0], h\rangle$ is a subpath of an execution path τ starting from $\langle \mathbf{if} * \mathbf{then}\, C_1\ \mathbf{else}\, C_2\ \mathbf{fi}\ ;\ C_3, s, h\rangle$. Consequently $\text{length}(\tau') < \text{length}(\tau)$.

If there is a precondition trace $(\Psi(\mathbf{E}), \Psi'(\mathbf{E}))$ following the edge, where $(s,h) \vDash \Psi^{\alpha}(\mathbf{E})$ and $(s',h') \vDash \Psi'^{\beta}(\mathbf{E})$, then, since the precondition for both conclusion and premise is the same (modulo the subformula concerning prophecy variables, which are not part of inductive predicate definitions), then the inductive predicates $\Psi(\mathbf{E})$ and $\Psi'(\mathbf{E})$ in both conclusion and premise are interpreted by the same least approximant. Hence $\alpha = \beta$. □

Theorem 4.2.2. *If $P \vdash C : \varphi$ is provable in the LTL system then it is valid.*

Proof. Suppose for contradiction that there is a cyclic proof $\mathcal{P}$ of $J = P \vdash C : \varphi$ but J is invalid. That is, for some stack s and heap h, we have $(s,h) \vDash P$ but there exists a path π starting from $\langle C, s, h\rangle$ such that $\pi \nvDash \varphi$. Then, by local soundness of the proof rules, we can construct an infinite path $(J_i = P_i \vdash C_i : \varphi_i)_{i \geq 0}$ of invalid sequents, so that for each J_i there exists a memory state (s_i, h_i) such that $(s_i, h_i) \vDash J_{i_P}$ but there exists an execution path π starting from $\langle J_{i_C}, s_i, h_i\rangle$ such that $\pi \nvDash \varphi$. Since $\mathcal{P}$ is a cyclic proof, by Definition 4.1.4 there must be an infinitely progressing LTL or precondition trace following some tail $(P_i \vdash C_i\colon \varphi_i)_{i\geq n}$ of the path.

If this trace is an LTL trace, by Definition 4.1.2 we know that for each pair of conclusion and premise (J_k, J_{k+1}), for some ψ, J_{k_φ} is of the form $G\psi$ and J_{k+1_φ} is of the form $XG\psi$, or vice versa. Moreover, by condition 2 of Lemma 4.2.1 we know there is a well-defined suffix of the path π' starting from $\langle J_{k+1_C}, J_{k+1_s}, J_{k+1_h}\rangle$ such that $\pi'_m \nvDash \psi$ and π' is a subpath of a path π starting from $\langle J_{k_C}, s_k, h_k\rangle$. Since by Definition 4.1.2 the trace following the path progresses infinitely often, we know that the length of the suffix path decreases infinitely often. This contradicts the well-foundedness of the natural numbers, which invalidates our assumptions. Consequently J must indeed be valid.

If this trace is a precondition trace, by Definition 3.2.8 we know that for each sequent J_k, there exists an inductive predicate formula $\Psi_k(\mathbf{E})$ which is a subformula of J_{k_P}. Moreover, by condition 3 of Lemma 4.2.1 we know that for each pair of conclusion and premise (J_k, J_{k+1}) the inductive predicates $\Psi_k(\mathbf{E})$ and $\Psi_{k+1}(\mathbf{E})$ have a least-fixed point interpretation which can be constructed as the union of a chain of ordinal-indexed approximations α and β respectively, where $\alpha \leq \beta$. Since by Definition 3.2.8 the trace following the path progresses infinitely often, we know that this chain of approximants decreases infinitely often, which contradicts the well-foundedness of the ordinals, in turn invalidating our assumption. Consequently J must indeed be valid. $\square$

Previously in this chapter we have shown a practical example of the application of our LTL cyclic proof system to demonstrate that all possible executions of a program exhibit a behaviour described by a temporal property. While the relatively low complexity of the example program resulted in a proof graph of manageable size, it is easy to estimate that larger, more complex programs would result in larger proofs that could be considered challenging to construct by hand and, more importantly, hard to manually check the global soundness condition that validates its soundness. Overcoming this problem is the focus of our attention in Chapters 6 and 7, where we present an implementation of our CTL and LTL cyclic proof systems presented so far, capable of automatically discovering temporal proofs of programs. We run this automated tool on a range of practical examples to demonstrate the viability of our approach to the verification of temporal properties.

Chapter 5

Fairness

An important component in the verification of reactive systems is a set of *fairness constraints* to guarantee that no computation is neglected forever. These fairness constraints are usually categorised as *weak* and *strong* fairness [68]. Since weak fairness requirements are usually related to parallel composition of processes, a property that our programming language lacks, we limit ourselves to the treatment of strong fairness. In this chapter we describe how our cyclic proof systems can be modified to treat (strong) fairness constraints.

In Section 5.1 we introduce our fairness constraint mechanism that guarantees that no computation is neglected forever. Then, in Section 5.2 we modify our CTL cyclic proof system to account only for fair paths; we establish the *fair global soundness condition* and prove its decidability. Finally, in Section 5.3 we show that a similar approach can be used to adapt our LTL cyclic proof system to introduce fairness constraints.

5.1 Fair program executions

The use of nondeterminism in our programming language in particular, and in static analysis in general, allows us to model program behaviour under unknown information introduced by external agents, such as user input, process schedulers and external procedures. Using nondeterministic loops we can, say, model the execution of a task while the user does not input a stop signal; or model a simple process scheduler that will alternate between two tasks by using nondeterministic branching commands. But the semantics of nondeterminism could be too relaxed, introducing program behaviour that was not intended by always choosing to execute one branch of a nondeterministic command while never executing the other possible branches.

Fairness constraints are, intuitively, a condition imposed on the execution of nondeterministic programs. They are commonly used as a mechanism to strengthen the semantics of nondeterminism by requiring that no computation is neglected forever. Hence, at every decision point where multiple possible executions are realisable, one must be *fair* in the selection of the subsequent execution.

Fairness constraints are commonly stated as a finite set of pairs of program points (C_1, C_2). Each of these pairs require that program location C_1 is executed infinitely often, if and only if program location C_2 is executed infinitely often. Whereas it is possible to have arbitrary program points for each fairness constraint, we limit their use to pairs of commands (C_i, C_j) for each nondeterministic command of the form **if** $*$ **then** C_i **else** C_j **fi** or **while** $*$ **do** C_i **od** C_j. For this purpose, we consider program commands to be uniquely labelled, to avoid confusion between different instances of the same command.

Definition 5.1.1 (Fair Execution)**.** Let C be a program command and $\pi = (\pi_i)_{i\geq 0}$ a program execution. We say that π *visits C infinitely often* if there are infinitely many distinct $i \geq 0$ such that $\pi_i = \langle C, \_, \_\rangle$.

An execution π is fair under fairness constraint (C_i, C_j) if it is the case that π visits C_i infinitely often if and only if π visits C_j infinitely often.

Furthermore, π is fair for a program C if it is fair for all fairness constraints (C_i, C_j) such that C contains a command of the form **if** $*$ **then** C_i **else** C_j **fi** or **while** $*$ **do** C_i **od** C_j.

Note that, according to Definition 5.1.1, every finite execution is trivially fair.

Fairness constraints allow us to identify fair and unfair program executions, but they do not prevent the *existence* of unfair execution paths. To this effect, we restrict the satisfaction definition of temporal properties to consider only the set of fair executions as follows:

Definition 5.1.2 (Fair CTL Satisfaction Relation)**.** A program execution π is a *model* of a CTL temporal formula ψ under fairness constraints if the relation $\pi \vDash_f \psi$ holds, defined by structural induction on ψ:

$$\begin{array}{rcl}
\gamma \vDash_f P & \Leftrightarrow & \gamma_s, \gamma_h \vDash P \\
\gamma \vDash_f \mathsf{error} & \Leftrightarrow & \gamma = \mathsf{fault} \\
\gamma \vDash_f \mathsf{final} & \Leftrightarrow & \gamma_C = \varepsilon \\
\gamma \vDash_f \varphi_1 \wedge \varphi_2 & \Leftrightarrow & \gamma \vDash_f \varphi_1 \text{ and } \gamma \vDash_f \varphi_2 \\
\gamma \vDash_f \varphi_1 \vee \varphi_2 & \Leftrightarrow & \gamma \vDash_f \varphi_1 \text{ or } \gamma \vDash_f \varphi_2 \\
\gamma \vDash_f \Diamond\varphi & \Leftrightarrow & \exists\gamma'.\ \gamma \rightsquigarrow \gamma' \text{ and } \gamma' \vDash_f \varphi \\
\gamma \vDash_f \Box\varphi & \Leftrightarrow & \forall\gamma'.\ \gamma \rightsquigarrow \gamma' \text{ implies } \gamma' \vDash_f \varphi \\
\gamma \vDash_f EF\varphi & \Leftrightarrow & \exists \text{ fair } \pi \text{ starting from } \gamma.\ \exists\gamma' \in \pi.\gamma' \vDash_f \varphi \\
\gamma \vDash_f AF\varphi & \Leftrightarrow & \forall \text{ fair } \pi \text{ starting from } \gamma.\ \exists\gamma' \in \pi.\gamma' \vDash_f \varphi \\
\gamma \vDash_f EG\varphi & \Leftrightarrow & \exists \text{ fair } \pi \text{ starting from } \gamma.\ \forall\gamma' \in \pi.\gamma' \vDash_f \varphi \\
\gamma \vDash_f AG\varphi & \Leftrightarrow & \forall \text{ fair } \pi \text{ starting from } \gamma.\ \forall\gamma' \in \pi.\gamma' \vDash_f \varphi \\
\gamma \vDash_f E(\varphi_1 U \varphi_2) & \Leftrightarrow & \exists \text{ fair } \pi \text{ starting from } \gamma.\ \exists i \geq 0.\ \pi_i \vDash_f \varphi_2 \\
 & & \text{and } \forall j{:}0 \leq j < i.\pi_j \vDash_f \varphi_1 \\
\gamma \vDash_f A(\varphi_1 U \varphi_2) & \Leftrightarrow & \forall \text{ fair } \pi \text{ starting from } \gamma.\ \exists i \geq 0.\ \pi_i \vDash_f \varphi_2 \\
 & & \text{and } \forall j{:}0 \leq j < i.\pi_j \vDash_f \varphi_1
\end{array}$$

Example 5.1.3. *To demonstrate the effect of fairness constraints on program analysis, consider a nonterminating program C that executes a given task (say setting a flag program variable x to true) a nondeterministic number of times while resetting the flag to $false$ after each execution of the loop:*

```
1: while(true) {
2:   while(*) {
3:      x:=true;
   }
4:   x:=false;
}
```

From the operational semantics of our language (Figure 2.1) we know there *exists an execution path π on which the body of the nondeterministic loop is executed forever, hence, never setting the value of program variable x to $false$. Given such an execution path, it is not true that for every execution path of our program we will eventually reach a program state (s,h) where $x = false$ (i.e. for all $(s,h), \langle C,s,h \rangle \nvDash AF(x = false)$). On the other hand, it is easy to see that π is not a fair execution path according to Definition 5.1.1.*

Suppose we know that, regardless of the number of iterations, the nondeterministic loop will always terminate (i.e. the loop guard depends on unknown but finite information). In this case, we can impose a condition on our program as a fairness constraint (C_3,C_4) to guarantee that if we execute the body of the loop infinitely often, we will also exit the loop infinitely often, effectively limiting the execution of the inner loop to a finite number of iterations on each iteration of the outer loop. Under this condition, we can easily see that for every path that meets our imposed condition (i.e. fair paths), we will eventually reach a program state (s,h) where $x = false$. In conclusion, under fairness constraint (C_3,C_4), program C satisfies the property $AF(x = false)$ (i.e. for all $(s,h), \langle C,s,h \rangle \vDash_f AF(x = false)$).

In the following section, we will show the cyclic proof of this example in an adaptation to our cyclic CTL proof system which restricts its analysis to fair execution paths.

5.2 Fair CTL cyclic proof system

Considering program executions under fairness constraints, we can adapt our CTL cyclic proof system to consider only fair executions by lifting the definition of fairness to proof graphs, and modifying our notion of validity and global soundness condition to consider only fair executions. Under these considerations, the interpretation of judgements is adapted as follows:

Definition 5.2.1 (Fair CTL Judgement)**.** A fair CTL judgement $P \vdash_f C : \varphi$ is *valid* if and only if, for all memory states (s,h) such that $s,h \vDash P$, we have $\langle C,s,h \rangle \vDash_f \varphi$.

We lift the definition of fairness from program executions to paths in a pre-proof in the expected way.

Definition 5.2.2 (Fair Path)**.** A path in a pre-proof $(J_i = P_i \vdash_f C_i : \varphi_i)_{i \geq 0}$ is said to *visit C infinitely often* if there are many distinct $i \geq 0$ such that $J_{i_C} = C$. A path in a pre-proof is fair under fairness constraints (C_i, C_j) if it is the case that $(J_i)_{i \geq 0}$ visits C_i infinitely often if and only if $(J_i)_{i \geq 0}$ visits C_j infinitely often. Furthermore, $(J_i)_{i \geq 0}$ is fair for a program C if it is fair for fairness constraints (C_i, C_j) such that C contains a command of the form **if** $*$ **then** C_i **else** C_j **fi** or **while** $*$ **do** C_i **od** C_j.

Intuitively, to account for fairness constraints, we simply need to restrict the global soundness condition of our CTL cyclic proof system so that it quantifies over all *fair* infinite paths in a pre-proof, ignoring unfair paths. However, as it stands, this intuition is not quite correct. Consider the program

```
while(true) {
  if(*) {
    x:=1;
  } else
    x:=2;
} }
```

This program has the CTL property $EG(x = 1)$ owing precisely to the unfair execution that always favours the first branch of the nondeterministic `if`. We can witness this using a cyclic proof with a single loop that invokes the rule (If* $\Diamond$1) infinitely often. The infinite path created by this loop is unfair and thus such a proof should not count as a fair cyclic proof. However, if we simply ignore this infinite path, the only one in the pre-proof, then the global soundness condition is trivially satisfied. Our answer is to take a more subtle view of the roles played by existential $(EG/EF/\Diamond)$ and universal $(AG/AF/\Box)$ properties; unfair paths created by the former must be disallowed, whereas unfair paths created by the latter can simply be disregarded.

Definition 5.2.3 (Bad Pre-proof)**.** A pre-proof $\mathcal{P}$ is *bad* if there is an infinite path $(J_i = P_i \vdash_f C_i : \varphi_i)_{i \geq 0}$ in $\mathcal{P}$ such that, given a program point C, the rule (Wh* $\Diamond$ 1)/(If* $\Diamond$1) is applied to infinitely many distinct J_i such that $J_{i_C} = C$ and (Wh* $\Diamond$ 2)/(If* $\Diamond$2) is applied to finitely many distinct J_i such that $J_{i_C} = C$, or vice versa.

Definition 5.2.4 (Fair Cyclic CTL Proof)**.** A pre-proof $\mathcal{P}$ is a *fair cyclic CTL proof* if

1. it is not bad, according to Definition 5.2.3 above, and
2. for every infinite *fair* path $(P_i \vdash_f C_i : \varphi_i)_{i \geq 0}$ in $\mathcal{P}$, there is an infinitely progressing $\Box$-trace, $\Diamond$-trace or precondition trace following some tail $(P_i \vdash_f C_i : \varphi_i)_{i \geq n}$ of the path.

To exemplify the concepts introduced in this section, we show a fair cyclic CTL proof of Example 5.1.3.

Example 5.2.5. *Assume the following labelled program C starts its execution from in initial program state (s,h), such that $(s,h) \vDash x = true$.*

```
1: while(true) {
2:   while(*) {
3:     x:=true;
     }
4:   x:=false;
   }
```

Under fairness constraint (C_3, C_4), we can verify that along every fair program execution in C there will always eventually be a program state in which $x = false$. Figure 5.1 shows a reduced version of the proof of this property in our fair cyclic CTL proof system, where the premise of each (AF) rule has been replaced by the corresponding subsequent application of a $(\vee)$ rule (omitted for brevity). Note the formation of a cycle on the leftmost branch of the tree, along which there is no trace following the path. Whereas such cycle would result in an invalid proof in our CTL cyclic proof system, in the case of a fair *CTL cyclic proof, such a cycle does not affect the validity of the proof as the path in question visits C_3 infinitely often but it does not visit C_4 infinitely often; hence the proof path is unfair as per Definition 5.2.2. Hence, the global soundness condition is trivially satisfied as there are only finite fair paths in the pre-proof. Consequently, the pre-proof qualifies as a fair CTL cyclic proof.*

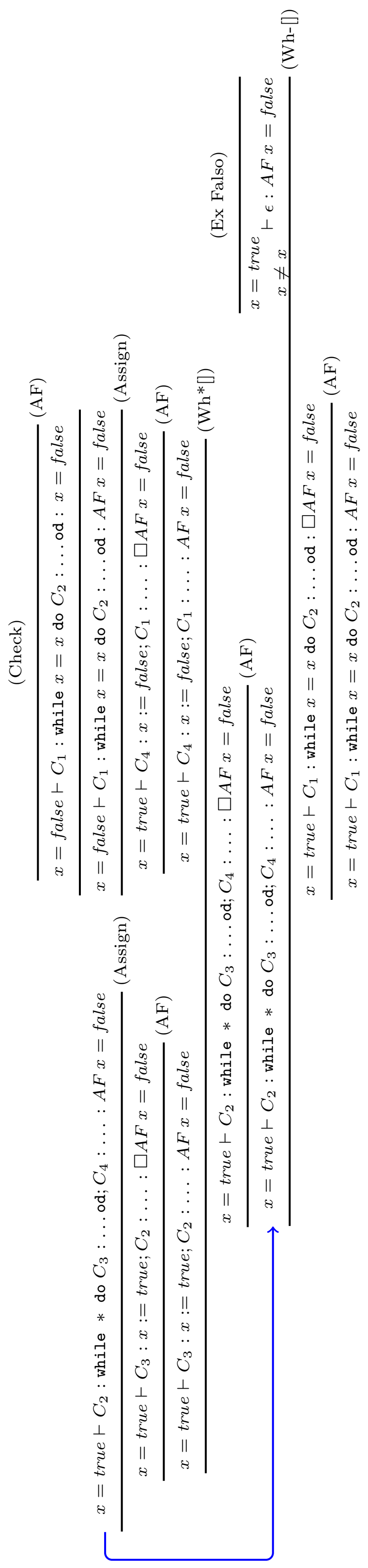


Figure 5.1: Fair CTL cyclic proof example

On a more realistic example, we return to our server program from Example 1.1.1. In Chapter 3, we have used our CTL cyclic proof system to prove that it is always *possible* for the heap to become empty, i.e. $AGEF(\mathsf{emp})$ (see Example 3.2.12 for full details). Whereas the *possibility* of freeing the heap is undoubtedly preferred over *never* freeing the heap, this property implicitly tells us that there might exist a possible program execution where the heap will never be empty, which is undesirable. Ideally, we would like to prove a stronger property stating that the heap will *always eventually* become empty, i.e. $AGAF(\mathsf{emp})$. Our server program in fact does not satisfy this property, because (i) the program can always choose to execute the second branch of the nondeterministic **if** command, always choosing to accept more job requests; and (ii) the program can always choose to execute the second inner loop infinitely often, adding job requests to the list forever. But suppose we have more information about our server; suppose we know that

1. our server is guaranteed to always alternate between accepting job requests and processing the list of accepted requests, never choosing the same task every time; and
2. our server only accepts a finite number of job requests at the time, in other words, the nondeterministic loop always terminates.

Under this assumptions, we could impose a fairness constraint in our server program of the form $\{(C_3,C_7),(C_8,C_1)\}$ to prove that the heap will *always eventually* become empty, i.e. $AGAF(\mathsf{emp})$.

Example 5.2.6. *Consider our server program listed in previous examples.*

```
1: while(true){
2:   if(*) {
3:     while(x!=nil) {
4:       temp:=x.next;
5:       free(x);
6:       x:=temp;
       }
```

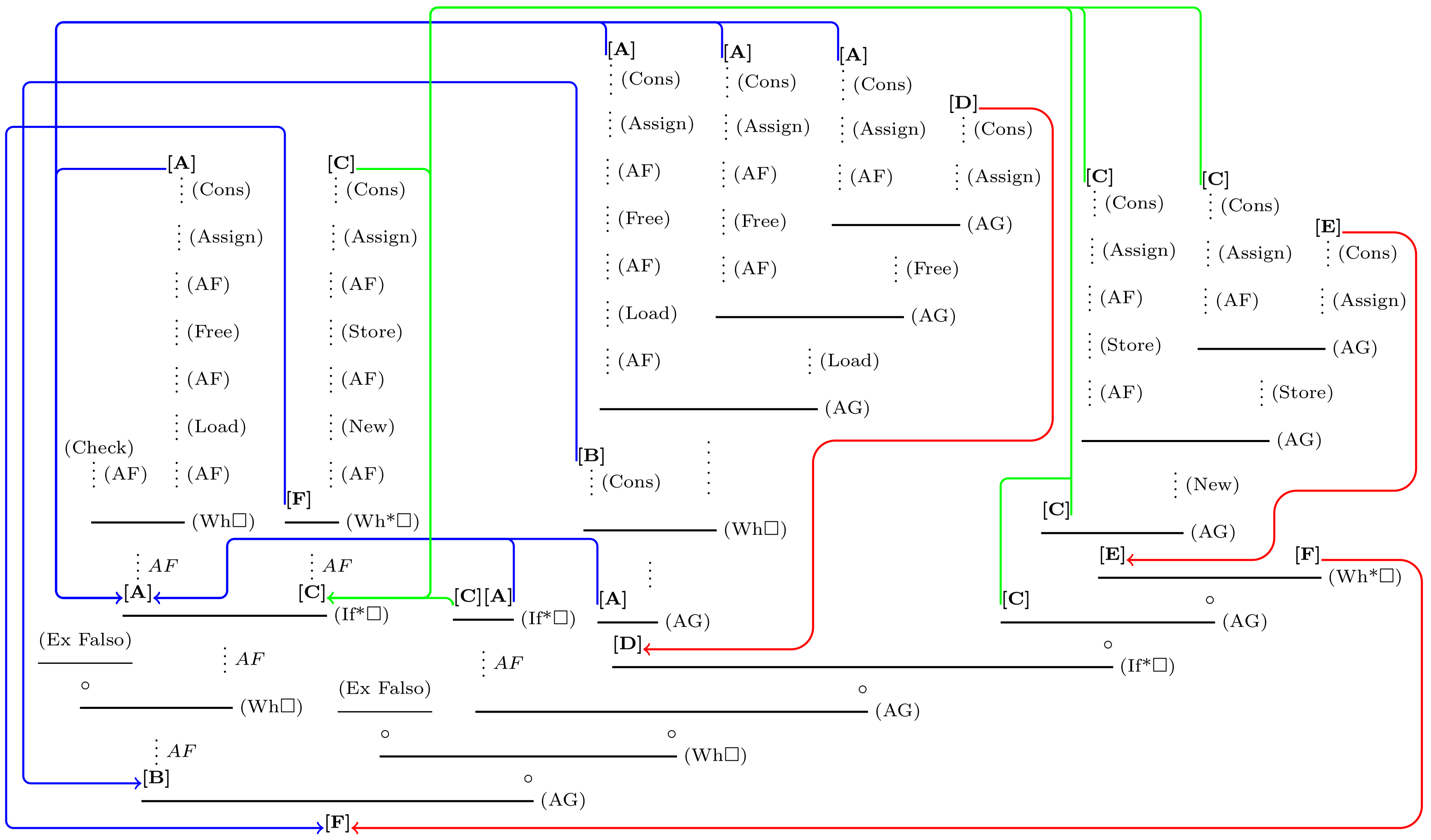
```
        } else {
7:        while(*) {
8:          y:=new();
9:          y.next:=x;
10:         x:=y;
      } } }
```

Moreover, assume the fairness constraint $\{(C_3,C_7),(C_8,C_1)\}$ *imposed on the analysis of this program under the assumptions listed above.*

Figure 5.2 shows a fair CTL cyclic proof of the property $AGAF(\mathsf{emp})$*. Note that the imposition of fairness constraints relaxes the conditions under which back-links can be formed. In particular, this relaxed condition can be seen in back-links on sequents labelled with* $[C]$ *as they yield an infinite path with no valid trace. Yet, because these infinite paths are* unfair*, they are not considered in the global soundness condition. Another case worth mentioning are those paths on which there is a valid precondition trace following (i.e. those formed by back-links on sequents labelled with* $[A]$*) as these, too, are unfair; this is because they visit only one branch of the nondeterministic* ***if*** *command infinitely often but they only visit the other branch finitely many times. A similar case happens with those infinite paths on which there is a valid* $\Box$*-trace following the path; every infinite iteration of these cycles (and their respective combinations) are unfair. Hence, they are not considered in the global soundness condition.*

The only infinite fair *path arises from following a combination of the previously discussed back-links. Following this fair path, we are guaranteed that the memory will always eventually be emptied, satisfying the property* $AGAF(\mathsf{emp})$*. Hence, this pre-proof qualifies as a valid cyclic proof since along every infinite fair path, there is a* $\Box$*-* and *precondition trace following the path.*

The demonstration of soundness of our fair CTL cyclic proof system follows closely the approach presented in Section 3.3. We first show local soundness of the proof rules along with the trace properties that are maintained by all derivation rules. This lemma is very similar to that of the CTL cyclic proof system, with slight



$[A] = ls(x, nil) \vdash \texttt{while}\, x \neq nil\, \texttt{do} \ldots \texttt{od} : AF(\mathsf{emp})$

$[B] = ls(x, nil) \vdash \texttt{while}\, x = x\, \texttt{do} \ldots \texttt{od} : AF(\mathsf{emp})$

$[C] = ls(x, nil) \vdash \texttt{while} * \texttt{do} \ldots \texttt{od} : AF(\mathsf{emp})$

$[D] = ls(x, nil) \vdash \texttt{while}\, x \neq nil\, \texttt{do} \ldots \texttt{od} : AGAF(\mathsf{emp})$

$[E] = ls(x, nil) \vdash \texttt{while} * \texttt{do} \ldots \texttt{od} : AGAF(\mathsf{emp})$

$[F] = ls(x, nil) \vdash \texttt{while}\, x = x\, \texttt{do} \ldots \texttt{od} : AGAF(\mathsf{emp})$

Figure 5.2: Single threaded monolithic server example

adaptations to consider only fair paths. Intuitively, the major difference on the proof of this lemma, in comparison to Lemma 3.3.1, is to demonstrate that the execution paths (trees) used in the proof are, indeed, fair. This is nonetheless trivial as any finite path (tree) is fair by Definition 5.1.1 As such, we list the adapted lemma for completeness, but omit its proof due to its similarity to Lemma 3.3.1.

Lemma 5.2.7 (Local Soundness)**.** *Let $J = (P \vdash_f C : \varphi)$ be the conclusion of a proof rule R. If J is invalid under (s,h), then there exists a premise of the rule $J' = P' \vdash_f C' : \varphi'$ and a model (s',h') such that J' is not valid under (s',h') and, furthermore,*

1. *if there is a box trace (φ, φ') following the edge (J,J') then, letting ψ be the unique formula given by Definition 3.2.7, there is a configuration γ such that $\gamma \nvDash_f \psi$, and the finite path $\pi' = \langle C', s', h' \rangle \ldots \gamma$ is well-defined and a subpath of $\pi = \langle C, s, h \rangle \ldots \gamma$. Therefore* $\text{length}(\pi') \leq \text{length}(\pi)$. *Moreover,* $\text{length}(\pi') < \text{length}(\pi)$ *when R is a symbolic execution rule.*

2. *if there is a diamond trace (φ, φ') following the edge (J,J') then, letting ψ be the unique formula given by Definition 3.2.7, there is a smallest finite tree κ with root $\langle C, s, h \rangle$, each of whose leaves γ satisfies $\gamma \nvDash_f \psi$. Moreover, κ has a subtree κ' with root $\langle C', s', h' \rangle$ and whose leaves are all leaves of κ. Therefore* $\text{height}(\kappa') \leq \text{height}(\kappa)$. *Moreover,* $\text{height}(\kappa') < \text{height}(\kappa)$ *when R is a symbolic execution rule.*

3. *if there is a precondition trace $(\Psi(\mathbf{E}), \Psi'(\mathbf{E}))$ following the edge (J,J') then letting α (β) be the least approximant for which the inductive predicate $\Psi(\mathbf{E})$ ($\Psi'(\mathbf{E})$) is interpreted, then the following relation holds and it is well-defined: $\beta \leq \alpha$. Moreover $\beta < \alpha$ when R is the (Unfold-Pre) rule.*

As for our previous systems, global soundness is obtained by extending the properties established in Lemma 5.2.7 to paths in a pre-proof, as follows.

Theorem 5.2.8 (Soundness)**.** *If $P \vdash_f C : \varphi$ is provable, then it is valid.*

Proof. Suppose for contradiction that there is a cyclic proof $\mathcal{P}$ of $J = P \vdash_f C : \varphi$ but J is invalid. That is, for some stack s and heap h, we have $(s,h) \vDash P$ but $\langle C, s, h \rangle \nvDash_f \varphi$.

By local soundness of the proof rules, we can construct an infinite path $\Pi = (P_i \vdash_f C_i : \varphi_i)_{i\geq 0}$ in $\mathcal{P}$ of invalid sequents. We first show that Π is a fair path (we limit ourselves to the treatment of nondeterministic branching commands as all other commands are naturally fair).

Suppose therefore that Π is unfair under fairness constraint (C_i, C_j), say. We consider the case in which C contains command **if** $*$ **then** C_i **else** C_j **fi** and Π visits C_i infinitely often and C_j only finitely often; the case for command **while** $*$ **do** C_i **od** ; C_j is similar. Using Definition 5.2.2 we know that **if** $*$ **then** C_i **else** C_j **fi** itself is symbolically executed infinitely often on Π. It cannot be the case that (If* $\Diamond$1) is applied infinitely often and (If* $\Diamond$2) only finitely often on Π, otherwise Π would be a *bad* path, which is specifically excluded by Definition 5.2.4. Nor it can be that both rules are applied finitely often, since in that case Π would be fair under constraint (C_i, C_j), contrary to our assumption.

The only remaining possibility is that (If*$\Box$) is applied infinitely often on Π. In that case, it must be the case that Π contains infinitely many occurrences of the left premise of the rule and only finitely many instances of the right premise of the rule (or vice versa). Hence the program execution underlying the pre-proof path Π is also unfair. Since the satisfaction relation $\vDash_f$ is restricted to fair program executions, this contradicts the assumption that $\langle C, s, h\rangle \nvDash_f \varphi$ (The full justification of this last step requires the observation that, in order to produce a $\Box$ infinitely often, φ must be of the form $AF\psi/AG\psi$). Consequently Π must be a fair path.

By Definition 5.2.4 we know that for every infinite fair path there exists an infinitely progressing trace following some tail $(P_i \vdash_f C_i : \varphi_i)_{i\geq n}$ of the path. If this trace is a $\Box$-trace, by condition 1 of Lemma 5.2.7 we can construct an infinite sequence of finite paths to a fixed configuration γ of infinitely decreasing length, contradiction. A similar argument related to the height of computation trees applies in the case of a $\Diamond$-trace. A precondition trace yields an infinitely decreasing sequence of ordinal approximations of some inductive predicate, also a contradiction.

$\Box$

5.2.1 Decidable soundness condition

The decidability problem of the global soundness conditions for cyclic proof systems has been well studied. [91, 16]. The usual approach consists on building two Büchi automata $\mathcal{B}_1, \mathcal{B}_2$ such that $\mathcal{B}_1$ accepts strings that corresponds to infinite paths in a pre-proof graph, and $\mathcal{B}_2$ accepts strings over which infinitely progressing traces following the path are found. Checking the global soundness condition amounts to check that the language inclusion $\mathcal{L}(\mathcal{B}_1) \subseteq \mathcal{L}(\mathcal{B}_2)$ holds. Nevertheless, as our fair CTL cyclic proof system, in comparison to previous cyclic proof systems, strengthens the global soundness condition by admitting only fair paths, we require a slightly more involved approach to demonstrate the decidability of our global soundness condition.

Our approach is as follows: we first check that a given pre-proof graph $\mathcal{P}$ is not bad according to Definition 5.2.3. We do so by building two Büchi automata $\mathcal{A}_{f1}$ and $\mathcal{A}_{f2}$ for each fairness constraint (C_i, C_j) such that $\mathcal{L}(\mathcal{A}_{f1})$ is the set of strings of vertices of $\mathcal{P}$ such that the rule (Wh* $\diamond$1)/(If* $\diamond$1) is applied infinitely often to a sequent of the form $P \vdash_f C : \varphi$ along the path, where $C = \textbf{if} * \textbf{then}\, C_i \,\textbf{else}\, C_j \,\textbf{fi}/\textbf{while} * \textbf{do}\, C_i \,\textbf{od}\, ; C_j$; similarly, $\mathcal{L}(\mathcal{A}_{f2})$ is the set of strings of vertices of $\mathcal{P}$ such that the rule (Wh* $\diamond$2)/(If* $\diamond$2) is applied infinitely often to a sequent of the form $P \vdash_f C : \varphi$ along the path, where $C = \textbf{if} * \textbf{then}\, C_i \,\textbf{else}\, C_j \,\textbf{fi}/\textbf{while} * \textbf{do}\, C_i \,\textbf{od}\, ; C_j$. We can check that $\mathcal{P}$ is not bad, according to Definition 5.2.3, by checking that $\mathcal{L}(\mathcal{A}_{f1}) \subseteq \mathcal{L}(\mathcal{A}_{f2})$ and $\mathcal{L}(\mathcal{A}_{f2}) \subseteq \mathcal{L}(\mathcal{A}_{f1})$. Secondly, we construct a Streett automata $\mathcal{A}_S$ such that $\mathcal{L}(\mathcal{A}_S)$ is the set of strings of vertices of $\mathcal{P}$ such that the string is fair for all pairs of fairness constraints (C_i, C_j) according to Definition 5.2.2. We then build a Büchi automaton $\mathcal{A}_T$ such that $\mathcal{L}(\mathcal{A}_T)$ is the set of strings of vertices of $\mathcal{P}$ such that an infinitely progressing trace can be found on a suffix of the string. We then check that the language inclusion relation $\mathcal{L}(\mathcal{A}_S) \subseteq \mathcal{L}(\mathcal{A}_T)$ holds; this, along with language inclusion relations $\mathcal{L}(\mathcal{A}_{f1}) \subseteq \mathcal{L}(\mathcal{A}_{f2})$ and $\mathcal{L}(\mathcal{A}_{f2}) \subseteq \mathcal{L}(\mathcal{A}_{f1})$ guarantee that $\mathcal{P}$ is a valid proof.

Following the approach described above, we first check that a pre-proof $\mathcal{P}$ is not bad. Recalling the definition of a Büchi automaton:

Definition 5.2.9 (Büchi automaton)**.** A nondeterministic *Büchi automaton* is a tuple $\mathcal{A} = (\Sigma, Q, q_0, \Delta, F)$, where:

- Σ is a finite alphabet;
- Q is a set of states;
- $q_0 \in Q$ is the initial state;
- $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation;
- $F \subseteq Q$ is the set of accepting states

Given an infinite word $\alpha = \alpha_0\alpha_1\alpha_2\ldots \in \Sigma^\omega$, where Σ^ω is the set of all infinite words over alphabet Σ, a *run* of $\mathcal{A}$ on α is a sequence of states $\sigma = \sigma_0\sigma_1\sigma_2$ such that $\sigma_0 = q_0$ and $\Delta(\sigma_i, \alpha_i, \sigma_{i+1})$ for all $i \geq 0$. Letting $\inf(\sigma)$ be the set of states occurring infinitely often in σ (i.e. $\inf(\sigma) = \{q \mid \exists$ infinitely many $i.\sigma_i = q\}$), then a run σ of $\mathcal{A}$ is said to be *accepting* if some accepting state occurs infinitely often on the run (i.e. $\inf(\sigma) \cap F \neq \emptyset$). The *language accepted by* $\mathcal{A}$ is defined by

$$\mathcal{L}(\mathcal{A}) = \{\alpha \in \Sigma^\omega \mid \text{ there is an accepting run of } \mathcal{A} \text{ on } \alpha\}$$

Our first Büchi automaton $\mathcal{A}_{f1}$, whose language accepts all strings over vertices of our pre-proof graph on which the rule (Wh* $\Diamond$1)/(If* $\Diamond$1) is applied infinitely often to a sequent of the form $P \vdash_f C : \varphi$ along the path, where $C =$ **if** $*$ **then** C_i **else** C_j **fi**/**while** $*$ **do** C_i **od** ; C_j, is built according to the following definition.

Definition 5.2.10 (Fair automaton 1)**.** Let $\mathcal{P} = (\mathcal{D}, \mathcal{L})$ be a fair CTL cyclic pre-proof. Then the *fair automaton* corresponding to P for command C is defined by *Fair1* $= (V, Q, q_0, \Delta, F)$, where

- V is the finite set of vertices in the proof graph $\mathcal{G}_\mathcal{P}$.
- $q_0 = (\mathrm{root}(\mathcal{D}), 0)$
- $Q = \{(v, f \in \{0,1\}) \mid v \in V\}$

- $F = \{(v,1) \mid v \in V\}$
- Δ is defined by

$$\Delta((v,f),v',(v',1)) \quad \text{where } r(v) = (\text{Wh}^* \Diamond 1)/(\text{If}^* \Diamond 1) \text{ and } s(v)_C = C$$

$$\Delta((v,f),v',(v',0)) \quad \text{for any other rule}$$

Proposition 5.2.11. *Given a program command C, for any $w \in V^\omega$, the fair automaton 1 accepts w if and only if the rule $(Wh^* \Diamond 1)/(If^* \Diamond 1)$ is applied infinitely often to a sequent of the form $P \vdash_f C : \varphi$ along some suffix of w.*

Proof. Case $\Leftarrow$. Suppose there is a suffix $v_i v_{i+1} v_{i+2} v_{i+3} \ldots$ of w such that the rule $(\text{Wh}^* \Diamond 1)/(\text{If}^* \Diamond 1)$ is applied infinitely along the suffix. Now define a sequence $(\sigma_j)_{j \geq 0}$ by:

$$\sigma_j = \begin{cases} (v_j,0) & \text{if } 0 \leq j \leq i-1 \\ (v_j,1) & \text{if } j \geq i \text{ and } r(v_j) = (\text{Wh}^* \Diamond 1)/(\text{If}^* \Diamond 1) \text{ and } s(v_j)_C = C \\ (v_j,0) & \text{if } j \geq i \text{ and } s(v_j)_C \neq C \text{ or } r(v_j) \text{ is any other rule} \end{cases}$$

It is easy to see that $(\sigma_j)_{j \geq 0}$ is a run of the trace automaton on w, and moreover, since the rule $(\text{Wh}^* \Diamond 1)/(\text{If}^* \Diamond 1)$ is applied infinitely, some final state $(v,1)$ must occur infinitely often in σ. Hence, σ is an accepting run of trace automaton on w.

Case $\Rightarrow$. Let σ be an accepting run of the trace automaton on $w = v_0 v_1 v_2 v_3 \ldots$. By definition, some state $(v,\tau,2)$ must occur infinitely often in σ. By construction there must be a suffix $\sigma_j \sigma_{j+1} \sigma_{j+2} \ldots$, where for all $j \geq i$, we have $\sigma_j = (v_i, s_j)$, where $s_j \in \{1,2\}$. Then, since by our assumption some state $(v,\tau,2)$ must occur infinitely often and for each of those states $v, r(v) = (\text{Wh}^* \Diamond 1)/(\text{If}^* \Diamond 1)$ and $s(v_j)_C = C$ it is easy to see that the rule $(\text{Wh}^* \Diamond 1)/(\text{If}^* \Diamond 1)$ is applied infinitely often. □

Similarly, our second Büchi automaton $\mathcal{A}_{f2}$, which language accepts all strings over vertices of our proof graph on which the rule $(\text{Wh}^* \Diamond 2)/(\text{If}^* \Diamond 2)$ is applied infinitely often to a sequent of the form $P \vdash_f C : \varphi$ along the path, where

$C = \textbf{if} * \textbf{then}\, C_i\, \textbf{else}\, C_j\, \textbf{fi}/\textbf{while} * \textbf{do}\, C_i\, \textbf{od}\, ; C_j$, is built according to the following definition.

Definition 5.2.12 (Fair automaton 2)**.** Same as Definition 5.2.10 except for the transition relation Δ, defined by

$$\Delta((v,f),v',(v',1)) \quad \text{where } r(v) = (\text{Wh}^* \diamond 2)/(\text{If}^* \diamond 2) \text{ and } s(v)_C = C$$

$$\Delta((v,f),v',(v',0)) \quad \text{for any other rule}$$

Proposition 5.2.13. *Given a program command C, for any $w \in V^\omega$, the fair automaton 2 accepts w if and only if the rule $(Wh^* \diamond 2)/(If^* \diamond 2)$ is applied infinitely often to a sequent of the form $P \vdash_f C : \varphi$ along some suffix of w.*

Proof. Similar to proof of Proposition 5.2.11 □

Given $\mathcal{A}_{f1}$ and $\mathcal{A}_{f2}$, we can check that along every infinite path of a fair CTL pre-proof $\mathcal{P}$, the rule $(\text{Wh}^* \diamond 1)/(\text{If}^* \diamond 1)$ is applied infinitely often if and only if the rule $(\text{Wh}^* \diamond 2)/(\text{If}^* \diamond 2)$ is applied infinitely often by checking that $\mathcal{L}(\mathcal{A}_{f1}) \subseteq \mathcal{L}(\mathcal{A}_{f2})$ and $\mathcal{L}(\mathcal{A}_{f2}) \subseteq \mathcal{L}(\mathcal{A}_{f1})$. If this condition is satisfied, then we guarantee that $\mathcal{P}$ is not bad.

For the second requirement of the global soundness condition we build an automaton $\mathcal{A}_S$ where $\mathcal{L}(\mathcal{A}_S)$ is the set of strings of vertices of a fair CTL pre-proof $\mathcal{P}$ such that the string is fair for all pairs of fairness constraints (C_i,C_j) according to Definition 5.2.2. Recalling the definition of a Streett automaton:

Definition 5.2.14 (Streett automaton)**.** A nondeterministic *Streett automaton* is a tuple $\mathcal{A} = (\Sigma, Q, q_0, \Delta, F)$, where:

- Σ is a finite alphabet;
- Q is a set of states;
- $q_0 \in Q$ is the initial state;
- $\Delta \subseteq Q \times \Sigma \times Q$ is the transition relation;

- $\{(R_1,G_1),(R_2,G_2),\dots,(R_k,G_k)\}$ is a finite set of pairs of states, where $R_i,G_i \in Q$

A run σ of $\mathcal{A}$ on an infinite word α is said to be accepting if for all $i : 0 \leq i \leq k$ we have $\inf(\sigma) \cap \{R_i\} \neq \emptyset \Rightarrow \inf(\sigma) \cap \{G_i\} \neq \emptyset$.

Our Street automaton $\mathcal{A}_s$, which language accepts all strings over vertices of our pre-proof graph that are fair with respect to the set of fairness constraints $\{(C_i,C_j),(C_{i+1},C_{j+1}),\dots,(C_{i+k},C_{j+k}))$ of program C is built according to the following definition.

Definition 5.2.15 (Fair Streett automaton)**.** Let $\mathcal{P} = (\mathcal{D},\mathcal{L})$ be a fair CTL cyclic pre-proof and let $\mathrm{FC} = \{(R_1,G_1),(R_2,G_2),\dots,(R_k,G_k)\}$ be the set of fairness constraints of program C. Then the *Fair Streett automaton* corresponding to $\mathcal{P}$ is defined by $\mathit{Street} = (V,Q,q_0,\Delta,F)$, where

- V is the finite set of vertices in the proof graph $\mathcal{P}$.
- $Q = \{(v) \mid v \in V\}$
- $q_0 = (\mathrm{root}(\mathcal{D}))$
- $F = \{(v_1,v_2) \mid v,v_1,v_2 \in V$ and $r(v) = (\mathrm{Wh}{*}\square)/(\mathrm{If}{*}\square)$ and $v_1 = p(1,v)$ and $v_2 = p(2,v)$ and for some $k, s(v_1)_C = R_k$ and $s(v_2)_C = G_k$
- Δ is defined by

$$\Delta((v),v',(v')) \quad \text{where } s(v') = p(k,v) \text{ for some } k$$

Proposition 5.2.16. *Given a program C and its set of fairness constraints* $\mathrm{FC} = \{(R_1,G_1),(R_2,G_2),\dots,(R_k,G_k)\}$*, for any $w \in V^\omega$, the fair Sttreet automaton accepts w if and only if w is fair under* FC.

Proof. Case $\Leftarrow$. Suppose there is a suffix $v_i v_{i+1} v_{i+2} v_{i+3} \dots$ of w which is fair under the set of fairness constraints $\mathrm{FC} = \{(R_1,G_1),(R_2,G_2),\dots,(R_k,G_k)\}$. That is, for all $0 \leq i \leq k$, if there exists an infinite number of states v_1 such that $s(v_1)_C = R_i$,

then there exists an infinite number of states $v2$ such that $s(v_2)_C = G_i$. Now define a sequence $(\sigma_j)_{j\geq 0}$ by $\sigma_j = v_j$. It is easy to see that $(\sigma_j)_{j\geq 0}$ is a run of the trace automaton on w, and moreover, since for all $0 \leq i \leq k$, if there exists an infinite number of states v_1 such that $s(v_1)_C = R_i$ (i.e. $\inf(\sigma) \cap R_i \neq \emptyset$), then there exists an infinite number of states $v2$ such that $s(v_2)_C = G_i$ (i.e. $\inf(\sigma) \cap G_i \neq \emptyset$). Consequently, σ is an accepting run of the fair Sttreet automaton with acceptance condition $\{(R_1, G_1), (R_2, G_2), \dots, (R_k, G_k)\}$ on w.
Case $\Rightarrow$. Let σ be an accepting run of the trace automaton on $w = v_0 v_1 v_2 v_3 \dots$. By definition, for all fairness constraints (C_i, C_j) in the acceptance condition F of the automaton, $\inf(\sigma) \cap C_i \neq \emptyset \Rightarrow \inf(\sigma) \cap C_j \neq \emptyset$ (i.e if there exists an infinite number of states v_1 such that $s(v_1)_C = C_i$, then there must be an infinite number of states v_2 such that $s(v_2)_C = C_j$). Then, since for all pair of commands in the acceptance condition $(C_1, C_j) \in$ F, the pair (C_i, C_j) is in the set of fairness constraints FC (i.e. $(C_i, C_j) \in$ FC), it is easy to see that the run w is fair under fairness constraints FC. □

Before introducing our trace automaton, we first introduce some auxiliary definitions that aid its construction.

Definition 5.2.17 (Trace value relation)**.** Let $\mathcal{T}$ be a finite set, and let $\text{TVal} \subseteq \mathcal{T} \times \mathsf{Seqs}$ be the relation that maps each sequent $S \in \mathsf{Seqs}$ to its (finitely many) trace values $\tau \in \mathcal{T}$ such that $\text{TVal}(S, \tau)$.

Definition 5.2.18 (Trace pair function)**.** A trace pair function $\text{TPair} : (\mathcal{T}, \mathcal{T}) \to (\mathsf{Seqs} \times \mathsf{Rules} \times \mathsf{Seqs}) \to \{0, 1, 2\}$ for any pre-proof $\mathcal{P} = (\mathcal{D} = \{V, s, r, p\}, \mathcal{L})$ is defined as:

$$\text{TPair}(\tau, \tau')(s, r, s') = \begin{cases} 1 & \text{if } \text{TVal}(\tau, s) \text{ and } \text{TVal}(\tau', s') \text{ and} \\ & (\tau, \tau') \text{ is a trace following } (s, s') \\ 2 & \text{if } \text{TVal}(\tau, s) \text{ and } \text{TVal}(\tau', s') \text{ and} \\ & (\tau, \tau') \text{ is a progressing trace following } (s, s') \\ 0 & \text{otherwise} \end{cases}$$

Our last Büchi automaton $\mathcal{A}_T$, whose language is the set of strings of vertices of $\mathcal{P}$ such that an infinitely progressing trace can be found on a suffix of the string is built according to the following definition.

Definition 5.2.19 (Trace Automaton)**.** Let $\mathcal{P} = (\mathcal{D}, \mathcal{L})$ be a fair CTL cyclic pre-proof. Then the *trace automaton* corresponding to $\mathcal{P}$ is defined by *Trace* = (V, Q, q_0, Δ, F), where

- V is the finite set of vertices in the proof graph $\mathcal{P}$.
- $q_0 = (\mathrm{root}(\mathcal{D}))$
- $Q = \{q_0\} \cup \{(v, \tau, p) \mid v \in V, \mathrm{TVal}(s(v), \tau), p \in \{1, 2\}\}$
- $F = \{(v, \tau, 2) \mid v \in V, \mathrm{TVal}(s(v), \tau)\}$
- Δ is defined by

$$\begin{array}{ll} \Delta(q_0, v, (v, \tau, 1)) & \text{where } \mathrm{TVal}(s(v), \tau) \\ \Delta((v, \tau, p), v', (v', \tau', 1)) & \text{if } \mathrm{TPair}(\tau, \tau')(s(v), r(v), s(v')) = 1 \\ \Delta((v, \tau, p), v', (v', \tau', 2)) & \text{if } \mathrm{TPair}(\tau, \tau')(s(v), r(v), s(v')) = 2 \end{array}$$

Proposition 5.2.20. *For any $w \in V^\omega$, the trace automaton accepts w if and only if there is an infinitely progressing trace following some suffix of w.*

Proof. Case $\Leftarrow$. Suppose there is a suffix $v_i v_{i+1} v_{i+2} v_{i+3} \ldots$ of w such that there is an infinitely progressing trace $\tau = \tau_i \tau_{i+1} \tau_{i+2} \tau_{i+3} \ldots$ following the suffix. Now define a sequence $(\sigma_j)_{j \geq 0}$ by:

$$\sigma_j = \begin{cases} q_0 & \text{if } 0 \leq j \leq i - 1 \\ (v_j, \tau_j, 1) & \text{if } j \geq i \text{ and } j \text{ is not a progressing point of } \tau \\ (v_j, \tau_j, 2) & \text{if } j \geq i \text{ and } j \text{ is a progressing point of } \tau \end{cases}$$

It is easy to see that $(\sigma_j)_{j\geq 0}$ is a run of the trace automaton on w, and moreover, as τ has infinitely many progressing points, some final state $(v,\tau,2)$ must occur infinitely often in σ. Hence, σ is an accepting run of trace automaton on w.

Case $\Rightarrow$. Let σ be an accepting run of the trace automaton on $w = v_0v_1v_2v_3\dots$. By definition, some state $(v,\tau,2)$ must occur infinitely often in σ. As state q_0 is not reachable from such state, there must be a suffix $\sigma_j\sigma_{j+1}\sigma_{j+2}\dots$, where for all $j \geq i$, we have $\sigma_j = (v_i,\tau_i,s_j)$, where $s_j \in \{1,2\}$. Then, it is easy to see that the trace $\tau = \tau_i\tau_{i+1}\tau_{i+2}\dots$ is a trace following the suffix $v_iv_{i+1}v_{i+2}\dots$. Moreover, as some state $(v,\tau,2)$ occurs infinitely often, this trace progresses infinitely often. $\square$

Given $\mathcal{A}_S$ and $\mathcal{A}_T$, checking that along every infinite fair path of a fair CTL pre-proof $\mathcal{P}$ there is an infinitely progressing trace along the path, according to Definition 5.2.4, amounts to checking that $\mathcal{L}(\mathcal{A}_S) \subseteq \mathcal{L}(\mathcal{A}_T)$.

Proposition 5.2.21 (Decidable Soundness Condition)**.** *It is decidable whether a fair pre-proof is a valid fair cyclic proof.*

Proof. For the first requirement (i.e. the pre-proof is not bad), let $\mathcal{A}_{f1}$ be a Büchi automaton built according to Definition 5.2.10. Moreover, let $\mathcal{A}_{f2}$ be a Büchi automaton built according to Definition 5.2.12. Checking that the rule $(\text{Wh}^* \diamond 1)/(\text{If}^* \diamond 1)$ is applied infinitely often if and only if the rule $(\text{Wh}^* \diamond 2)/(\text{If}^* \diamond 2)$ is applied infinitely often along all paths in $\mathcal{P}$ is reduced to checking that the following relation holds of the languages accepted by both automata: $\mathcal{L}(\mathcal{A}_{B2}) \subseteq \mathcal{L}(\mathcal{A}_{B1})$ and $\mathcal{L}(\mathcal{A}_{B1}) \subseteq \mathcal{L}(\mathcal{A}_{B2})$. Since language inclusion of Büchi automata is decidable, then our first requirement for the soundness condition is decidable.

For the second requirement, let $\mathcal{A}_S$ be a Fair Streett automaton with acceptance condition formed of conjuncts of the form $(\text{Fin}(i) \vee \text{Inf}(j)) \wedge (\text{Fin}(j) \vee \text{Inf}(i))$ for each pair of fairness constraints (i,j) according to Definition 5.2.15. Moreover, let $\mathcal{A}_T$ be a Büchi automata that accepts all infinite paths in $\mathcal{P}$ such that an infinitely progressing trace exists along the path, as per Definition 5.2.19. $\mathcal{P}$ is valid if and only if $\mathcal{A}_S$ accepts all strings accepted by $\mathcal{A}_T$ (i.e. $\mathcal{L}(\mathcal{A}_S) \subseteq \mathcal{L}(\mathcal{A}_T)$). Since Streett automata can be transformed into Büchi automata [69] and inclusion between Büchi automata is decidable, then our second requirement for the soundness condition is decidable.

Consequently, since both requirements are decidable, the global soundness condition for fair cyclic proofs is decidable. □

5.3 Fair LTL cyclic proof system

In this section, we show how to adapt our LTL cyclic proof system introduced in Chapter 4 to account only for fair paths. Following the approach introduced in the previous section, we first restrict the satisfaction relation of Linear Temporal Logic to consider only the set of fair execution paths as follows:

Definition 5.3.1 (Fair LTL Satisfaction Relation)**.** A program execution π is a model of a LTL temporal formula ψ under fairness constraints if the relation $\pi \vDash_f \psi$ holds, defined by structural induction on ψ:

$$\begin{array}{rcl}
\pi \vDash_f Q & \Leftrightarrow & (\pi[0]_s, \pi[0]_h) \vDash Q \\
\pi \vDash_f \mathsf{error} & \Leftrightarrow & \pi[0] = \mathsf{fault} \\
\pi \vDash_f \mathsf{final} & \Leftrightarrow & \pi[0]_C = \epsilon \\
\pi \vDash_f \psi_1 \wedge \psi_2 & \Leftrightarrow & \pi \vDash_f \psi_1 \text{ and } \pi \vDash_f \psi_2 \\
\pi \vDash_f \psi_1 \vee \psi_2 & \Leftrightarrow & \pi \vDash_f \psi_1 \text{ or } \pi \vDash_f \psi_2 \\
\pi \vDash_f X\psi & \Leftrightarrow & \pi_1 \vDash_f \psi \\
\pi \vDash_f F\psi & \Leftrightarrow & \exists k \geq 0.\, \pi_k \vDash_f \psi \\
\pi \vDash_f G\psi & \Leftrightarrow & \pi \text{ is fair and } \forall k \geq 0.\, \pi_k \vDash_f \psi \\
\pi \vDash_f \psi_1 U \psi_2 & \Leftrightarrow & \exists k \geq 0.\, \pi_k \vDash_f \psi_2 \text{ and } \forall 0 \leq j \leq k.\, \pi_j \vDash_f \psi_1
\end{array}$$

Note that only G properties explicitly require the programming execution path to be fair. Models of all other formulas are implicitly finite hence they are fair.

We then can adapt our LTL cyclic proof system to consider only fair executions paths by lifting the definition of fairness to proof graphs and by modifying our notion of validity and global soundness condition to consider only fair executions. Under this considerations, the interpretation of judgements is adapted as follows:

Definition 5.3.2 (Fair LTL Judgement)**.** A fair LTL judgement $P \vdash_f C : \varphi$ is *valid* if and only if, for all memory states (s,h) and for all fair execution paths π starting from $\langle C,s,h\rangle$ we have $s,h \vDash P$ implies $\pi \vDash_f \varphi$.

Since we are only concerned with proofs about fair executions, we modify the definition of a valid cyclic LTL proof.

Definition 5.3.3 (Fair Cyclic LTL Proof)**.** A pre-proof $\mathcal{P}$ is a *fair cyclic LTL proof* if, for every infinite fair path $(J_i)_{i\geq 0}$ in $\mathcal{P}$, there is an infinitely progressing LTL or precondition trace following some tail $(J_i)_{i\geq n}$ of the path.

To exemplify the concepts introduced in this section, we show a fair cyclic LTL proof of the following example.

Example 5.3.4. *Assume the following labelled program C starts its execution from in initial program state (s,h), such that $(s,h) \vDash x = true$.*

```
1: while(x!=nil) {
2:   if(*) {
3:     skip;
     } else {
4:     free(x);
     }
   }
5: y:=true
6: while(true) {
7:   skip
   }
```

Under fairness constraint (C_3,C_4), we can verify that along every fair program execution in C there will always eventually be a program state from which $y = true$ holds onwards. Figure 5.3 shows a reduced version of the proof of this property in our fair cyclic LTL proof system, where the premise of each (F) rule has been replaced by the corresponding subsequent application of a $(\vee)$ rule, which in turn has been omitted from the proof for brevity.

Note the formation of a cycle on the leftmost branch of the three. Infinite iterations of this path visit C_3 infinitely often, but they do not visit C_4 at all. Given that this path is unfair, it is in fact not considered in the validity of the proof. On the other hand, the cycle on the top of the pre-proof is, in fact, fair, as it does not visit C_3 or C_4 infinitely often. Given that along this path there is an LTL trace following the path, our global soundness condition is satisfied. Consequently, the pre-proof qualifies as a fair LTL cyclic proof.

The demonstration of soundness of our fair LTL cyclic proof system follows closely the approach presented in Section 4.2, with slight adaptations to consider only fair paths, similarly done in Section 5.2. As such, we list the adapted lemma for completeness, but omit its proof due to its similarity to Lemma 4.2.1 and to Lemma 5.2.7.

Lemma 5.3.5. *Let $J = (P \vdash_f C : \varphi)$ be the conclusion of a proof rule R. If J is invalid under (s,h), then there exists a premise of the rule $J' = (P' \vdash_f C' : \varphi')$ and a model (s',h') such that J' is not valid under (s',h') and, furthermore,*

1. *if there is an LTL trace (φ, φ') following the edge (J,J') then, letting ψ be the unique formula given by Definition 4.1.2, there exists a k such that $\pi_k \not\models \psi$, and the finite path $\pi' \overset{\text{def}}{=} \langle C', s', h'\rangle \dots \pi_k$ is a subpath of $\pi \overset{\text{def}}{=} \langle C, s, h\rangle \dots \pi_k$. Therefore* $\text{length}(\pi') \leq \text{length}(\pi)$. *Moreover,* $\text{length}(\pi') < \text{length}(\pi)$ *when R is a symbolic execution rule.*

2. *if there is a precondition trace $(\Psi(\mathbf{E}), \Psi'(\mathbf{E}))$ following the edge (J,J') then letting α (β) be the least approximant for which the inductive predicate $\Psi(\mathbf{E})$ ($\Psi'(\mathbf{E})$) is interpreted, then the following relation holds and it is well-defined: $\beta \leq \alpha$. Moreover $\beta < \alpha$ when R is the (Unfold-Pre) rule.*

As for our previous systems, global soundness is obtained by extending the properties established in Lemma 5.3.5 to paths in a pre-proof, as follows.

Theorem 5.3.6 (Soundness). *If $P \vdash_f C : \varphi$ is provable, then it is valid.*

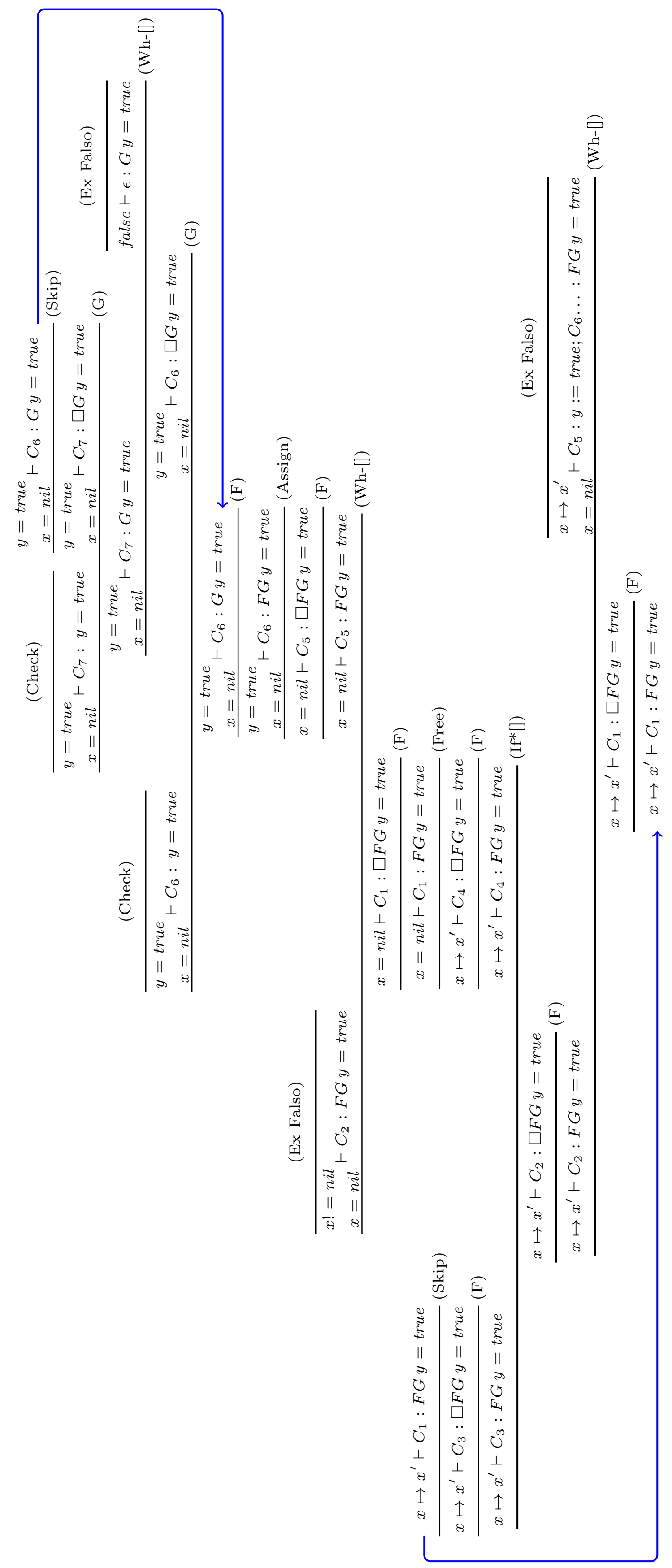


Figure 5.3: Fair LTL cyclic proof example

Proof. Suppose for contradiction that there is a fair cyclic LTL proof $\mathcal{P}$ of $J = P \vdash_f C : \varphi$ but J is invalid. That is, for some stack s and heap h, we have $(s,h) \models P$ but for some execution path π starting from $\langle C,s,h \rangle$, $\pi \not\models_f \varphi$. By local soundness of the proof rules, we can construct an infinite path $(P_i \vdash_f C_i : \varphi_i)_{i \geq 0}$ in $\mathcal{P}$ of invalid sequents. We first show that such an infinite path is a fair path.

Suppose, for contradiction, that such an infinite unfair path is the only invalid path. Since the path is unfair, then by Definition 5.2.2 the underlying execution path is also unfair. Since by our assumption $\pi \not\models_f \varphi$, then there must be a fair path along which φ is not satisfied. This contradicts our assumption that the unfair path is the only invalid path. Consequently there must be another invalid infinite path in the pre-proof which is fair.

By Definition 5.3.3 we know that for every infinite fair path there exists an infinitely progressing LTL trace following some tail $(P_i \vdash_f C_i : \varphi_i)_{i \geq n}$ of the path. By condition 1 of Lemma 5.3.5 we can construct an infinite sequence of finite paths to a fixed configuration γ of infinitely decreasing length, contradiction. A precondition trace yields an infinitely decreasing sequence of ordinal approximations of some inductive predicate, also a contradiction.

□

In this chapter we have discussed how to adapt our cyclic proof systems to handle fairness constraints for both our CTL and LTL cyclic proof systems. Along the discussion we have presented proof figures of simple example programs which were mainly designed to exemplify the concepts introduced in the chapter, as similarly done in previous Chapter 3 and Chapter 4. Despite the relative simplicity of the examples, the proofs are quite large, and would require great care when produced by hand. On the other hand, due to the nature of our approach, we have the advantage of a natural way to automate the elaboration of these proofs by means of a proof search algorithm. An automated implementation of the cyclic proof systems presented so far is the subject of the following chapter.

Chapter 6

Implementation

In this chapter we discuss the implementation details of the cyclic proof systems presented earlier in this thesis. Throughout this chapter, we focus on the CTL cyclic proof system as the basis of the discussion and point out the LTL and fairness adaptations only when the specifics deviate from the main implementation.

We implement our proof systems on top of the CYCLIST theorem prover [22]. As a generic cyclic theorem prover, CYCLIST provides an interface to instantiate user-defined cyclic proof systems as OCaml functors parameterised over user-defined datatypes that describe the desired logic and its set of axioms and rules of inference. Once the logic has been instantiated, CYCLIST provides the capability to define and/or extend the proof strategy with user-defined tactics. Finally, CYCLIST also provides an interface to a custom model checker based for the automated verification of the global soundness condition. With these user-defined modules in place, cyclist manipulates the cyclic proof structure and performs the proof search algorithm, resulting in a valid cyclic-proof structure in case one is found.

The rest of this chapter is structured as follows: in Section 6.1 we introduce the code structures and concepts used throughout the implementation as the basis for our proof structures. We then introduce our proof rules as functions that transform a conclusion sequent to a list of premise sequents and keeps track of the traces following the path. These structures constitute the instantiation of our logic and its axioms and rules of inference that are feed to CYCLIST for its manipulation. Later on, in Section 6.2, we discuss the details of tactics defined to extend CYCLIST proof search algorithm. We conclude this chapter with the details of the extension for the automated soundness check of our *fair* cyclic proofs. The implementation's source code and benchmarks are publicly available at [45].

6.1 Fundamentals

The entry point of our implementation is a command line program that receives an input file as its parameter. This file is required to have the following structure

- `fields`: <list_of_fields>;
- `precondition` : <symbolic_heap_formula>;
- `property`: <temporal_logic_formula>;
- <list_of_commands>

where *list_of_fields* is a list of the record names to which data structures point. *symbolic_heap_formula* is a precondition formula as per Defn. 2.2.1 that describes the initial state of the program memory before its execution. *temporal_logic_formula* is a CTL (LTL) temporal logic formula as per Defn. 2.3.1 (Defn. 2.3.3) that expresses the temporal behaviour to prove of the given program. Finally, *list_of_commands* is a program written in the language defined in Section 2.1.

The implementation parses the input and generates a *SEQUENT* structure: an OCaml module that defines a tuple with type `(SymbolicHeap * CommandList * TemporalFormula)`, along with utility functions that operate on this type. When generating a SEQUENT, a pre-processing step is triggered where the inductive predicates in the `SymbolicHeap` formula are individually annotated with (distinct) identifiers to distinguish the traces that arise from them; a similar annotation pre-processing is done for the temporal subformulas that give rise to traces (i.e. *AG* and *EG* subformulas for CTL and *G* for LTL formulas). Once the first pre-processing step is done, a proof *NODE* is created by extending the SEQUENT structure with two extra types to facilitate the tracking of traces, resulting in a tuple of type `(SymbolicHeap * CommandList * TemporalFormula`

`* TagsList * TagPairsList)`, where `TagsList` is simply a list of the identifiers used to annotate distinct inductive predicates and temporal subformulas during the first pre-processing step, and `TagPairList` is a list of pairs of tags (i.e. each element of the list has type `(Tag * Tag)`) that represent the progressing step of the traces that arise from the annotated formulas. These annotated sequents are the basis of our proof rules, which in turn transform one NODE goal to a list of NODE premises according to the definitions of the rules in our proof systems described earlier.

6.1.1 Proof Rules

Proof rules are functions that operate on proof NODE structures. In the case of axioms, the functions receive a proof NODE as an argument and return an `Option` monad, depending on whether the sequent is an axiom of our proof system. As per Figure 3.1, the axioms in our CTL system are implemented as follows:

```
let symex_check_axiom entails =
Rule.mk_axiom (fun (sf,_,tf) -> Option.mk
    (Tl_form.is_checkable tf &&
    Option.is_some (entails sf (Tl_form.extract_slformula tf))) "Check")

let ex_falso_axiom =
Rule.mk_axiom (fun (sf,_,_) -> Option.mk
    (Sl_form.inconsistent sf) "Ex_Falso")

let symex_empty_axiom =
Rule.mk_axiom (fun (_,cmd,tf) -> Option.mk
    (Cmd.is_empty cmd && Tl_form.is_box tf) "Empty")
```

In the case of inference rules, the functions return a list of tuples, each of type NODE (along with a descriptor that identifies each rule name). Each entry in the list represents a sequent premise of the inference rule along with bookkeeping information about the tagpairs of the sequent and the progressing tagpairs that result of the application of the rule. We abstract this bookkeeping in the auxiliary function `fix_tps` which receives a SEQUENT as a parameter and returns the corresponding proof NODE by computing the tags of the sequent and the progressing tagpairs from conclusion to premise.

```
  let fix_tps premise_list =
  Blist.map
    (fun (goals,description) -> Blist.map (fun sequent ->
    (sequent, tagpairs sequent, progpairs sequent)) goals, description) list
```

Then each rule simply produces its corresponding list of premise sequents and passes its result to `fix_tps` to do the tags and traces bookkeeping.

The following code snippet illustrates the implementation of our inference rules, showing the details of our (AG) rule source code. The rule first obtains the precondition (`sf`), program command (`cmd`) and temporal formula (`tf`) components of the sequent `seq`. Then, the temporal formulas `tf1` and `tf2` for the corresponding premise sequents are computed. Finally a list of two SEQUENTS (each corresponding to a premise of the rule) is passed to the auxiliary function `fix_tps` for tagpair bookkeeping. The body of the function is surrounded by a `try` ...`catch` statement in case the temporal formula in question is not of the correct form, in which case an empty list is returned.

```
let ag_rule =
  let rl seq =
    try
      let (sf,cmd,tf) = Seq.dest seq in
      let (tf1,tf2) = Tl_form.unfold_ag tf in
        fix_tps
          [[([sf],cmd,tf1); ([sf],cmd,tf2)], "AG"]
    with WrongTf -> [] in
  wrap rl
```

As for the symbolic execution rules, their logic embeds the symbolic execution engine needed in our system. Roughly speaking, all symbolic execution rules follow a similar pattern in their operation by performing the following steps:

1. Modify the symbolic heap formula in the precondition according to the operational semantics of the command being executed;

2. Replace the command list component of the sequent with the next command to be executed (its continuation);

3. Modify the temporal formula postcondition, removing the prepending $\Diamond$ or *Box* subformula.

We abstract steps 2 and 3 in an auxiliary function `mk_symex` that computes the continuation `cont` of the program command and the corresponding temporal formula `tf'` of the premises as follows:

```
let mk_symex f =
  let rl seq =
  try
    let (_,cmd,tf) = Seq.dest seq in
    let cont = Cmd.get_cont cmd in
    let tf' = Tl_form.step tf in
      fix_tps (Blist.map
              (fun (g,d) -> Blist.map (fun h' -> ([h'], cont, tf')) g, d)
              (f seq))
  with WrongCmd, WrongTf -> []
  in wrap rl
```

Where the auxiliary function *wrap* attempts to simplify the resulting sequent by removing unnecessary equalities in the precondition symbolic heap formula.

```
  let wrap r =
  Rule.mk_infrule
    (Seqtactics.compose r (Seqtactics.attempt simplify_seq_rl))
```

Then each symbolic execution rule is simply a function from a SEQUENT to a list of symbolic heap formulas, which are the result of symbolically executing a program. This list of symbolic execution formulas is then passed to the auxiliary `mk_symex` function to finish the construction of the premise sequents.

Modifying the symbolic heap formula in the precondition has a similar pattern for most rules: they first obtain the precondition and the command components from the SEQUENT argument, disregarding the temporal formula component. Then, the expressions and variables involved in the command are obtained. Finally, the new symbolic formula that reflects the change in the program state as result of the symbolic execution is computed.

The next code snippet illustrates these steps, showing the details of the (Free) rule. In this rule, the precondition heap formula and the program command are stored in variables `sf` and `cmd` respectively. Then, the expression representing the memory location to be freed is obtained and stored in local variable `e`. A call to an auxiliary function is made to obtain the symbolic heap subformula pertaining to the memory location to be freed; this subformula is stored in variable `pto`. The function returns the symbolic heap formula that results from removing the subformula `pto` from the original sequent precondition `sf` or an empty list in case any of the previous operations raises an exception.

```
let symex_free_rule =
  let rl seq =
    try
      let (sf,cmd,_) = dest_sh_seq seq in
      let e = Cmd.dest_free cmd in
      let pto = find_pto_on sf e in
      [[ SH.del_pto sf pto ], "Free"]
    with Not_symheap | WrongCmd | Not_found -> [] in
  mk_symex rl
```

All other symbolic execution rules are implemented in a similar way to this example, with the exception of the branching commands, whose implementation is slightly more complex as the result depends on whether a universally or existentially quantified path formula is present and whether the branching condition holds of one execution path or the other.

The following code snippet shows the implementation of the (While) rule. In this rule, the precondition heap formula `sf`, the program command `cmd` and the temporal property `tf` of the sequent are stored in local variables. The guard condition `cond` and the body of the loop `cmd'` are computed, followed by the continuation `cont` of the while loop. Then, the two symbolic heaps resulting from extending the precondition formula `sf` with a symbolic heap that satisfies(invalidates) the condition `cond` are computed and stored in variables `sf'`(`sf''`). Then, the corresponding temporal property formula `tf'` for the premise(s) is calculated. Next, we proceed to compute the appropriate premise sequent(s) depending on whether we are exploring a $\Box$ or a $\Diamond$ formula.

In case of a $\Box$ formula, we return a list of two SEQUENT structures. The first one is composed of (1) a precondition formula `sf'` that satisfies the condition of the loop; (2) a program component comprised of the sequence of the body of the loop followed by the execution of the loop (3) a temporal property `tf'`. The second SEQUENT structure is composed of (1) a precondition formula `sf''` that invalidates the condition of the loop; (2) a program component `cont` that is the continuation of the loop (the next command in the original sequence of commands) (3) a temporal property `tf'`.

In case of a $\Diamond$ formula, we always return a single premise SEQUENT depending on whether the precondition formula validates the condition of the loop or not. In case it does, we return a sequent corresponding to the execution of the body of the loop. On the contrary, we return a sequent corresponding to the execution of the next command in the original sequence of commands.

```
let symex_while_rule =
  let rl seq =
    try
      let (sf,cmd,tf) = dest_sh_seq seq in
      let (cond,cmd') = Cmd.dest_while cmd in
      let cont = Cmd.get_cont cmd in
      let (sf',sf'') = Cond.fork sf cond in
      let tf' = Tl_form.step tf in
      if Tl_form.is_box tf then
        fix_tps
          [[([sf'],Cmd.mk_seq cmd' cmd,tf');([sf''],cont,tf')], "While-Box"]
      else if Tl_form.is_diamond tf then
        if Cond.validated_by sf cond then
          fix_tps [[ ([sf'], Cmd.mk_seq cmd' cmd, tf')], "While-<>1"]
        else
          fix_tps [[ ([sf''], cont, tf')], "While-<>2"]
      else
        []
    with Not_symheap | WrongCmd -> [] in
  wrap rl
```

Even though not technically a proof rule in our systems, backlinks are encoded as a proof rule in the implementation, in the sense that they are functions from proof NODES to previously discovered NODES in our proof search. The following code snipet shows the details of the backlink function. Backlinking differs from all other proof rules in that not only receives a proof node as its parameter, but also the whole pre-proof structure explored so far, `prf`. From these parameters, we first obtain the SEQUENT structure `src_seq` of the open NODE and the list of all target NODES, `targets`, from the proof object. We then compute a list of those nodes that syntactically match our `src_seq`, either directly, by substitution of existentially quantified variables, or by the result of the (Cons) rule. These nodes are stored in `apps`. Depending on whether (Cons) and variable substitution are needed, we return a sequence of these rules preceding the application of `Rule.mk_backrule` which will trigger the soundness procedure to verify that the resulting proof is a valid cyclic proof.

```
let dobackl node prf =
  let src_seq = Proof.get_seq node prf in
  let targets = Rule.all_nodes node prf in
  let apps =
    Blist.bind
      (fun node' ->
       Blist.map
         (fun res -> (node',res))
         (matches src_seq (Proof.get_seq node' prf))
      )
      targets in
  let f (targ_node, (theta,tagpairs)) =
      let targ_seq = Proof.get_seq targ_node prf in
      let (sf_targ_seq,cmd_targ_seq,tf_targ_seq) = targ_seq in
      let targ_seq' = (Sl_form.subst_tags
                       tagpairs sf_targ_seq, cmd_targ_seq, tf_targ_seq) in
      let subst_seq = Seq.subst theta targ_seq' in
      Rule.sequence [
          if Seq.equal src_seq subst_seq
          then Rule.identity
          else Rule.mk_infrule (cons subst_seq);

          if Sl_term.Map.for_all Sl_term.equal theta
          then Rule.identity
          else Rule.mk_infrule (subst_rule theta targ_seq');

          Rule.mk_backrule
            false
            (fun _ _ -> [targ_node])
            (fun _ _ ->
             (TagPairs.reflect tagpairs), "Backl"])
        ] in
  let rule_list = (Blist.map f apps) in
  Rule.first rule_list node prf
```

6.2 Proof Search Algorithm

The proof rules are the building blocks of the proof *tactics* used in our proof search algorithm. These tactics are esentially a strategy that define the order in which the proof rules should be applied. In implementation terms, these tactics are functions from a list of proof rules to a proof rule. For example, `Rule.first` tries each rule in its list of arguments one at the time, in the order they appear, until a rule succeeds. Once a rule succeeds, no other rules down the list will be applied. Using this rule, we build a generic `unfold_gs` tactic to group (AG) and (EG) rules as they induce a common pattern in our proof search.

```
let unfold_gs =
Rule.first [
  ag_rule ;
  eg_rule ;
]
```

A slightly more involved tactic is `symex`, which not only groups all the symbolic execution rules but also makes use of `Rule.compose` and `Rule.attempt` tactics to optimise the application of branching rules. `Rule.compose` applies the rule in the first argument and immediately applies the tactic in the second argument to (all of) the premise(s) that result from the application of the first tactic. `Rule.attempt`, as its name implies, attempts to apply a rule; in case the application is unsuccessful, the proof search is able to backtrack instead of failing. Using these two tactics, we optimise the application of branching commands as one of their premises usually results in an unsatisfiable state, which is easily discharged by our (Ex.Falso) axiom.

```
let symex =
Rule.first [
  symex_skip_rule ;
  symex_assign_rule;
  symex_load_rule ;
  symex_store_rule ;
  symex_free_rule ;
  symex_new_rule ;
  (Rule.compose symex_ifelse_rule (Rule.attempt ex_falso_axiom));
  (Rule.compose symex_while_rule (Rule.attempt ex_falso_axiom));
]
```

Tactics like these form the basis of the core of the automated verification tool: a proof search algorithm whose main goal is to close all open nodes in the proof graph. This is done by first checking if the open node is an instance of an axiom, in which case the node is marked as closed. Otherwise we attempt to find a successful application of an inference rule using the following tactic:

```
rules := Rule.first [
  split;
  simplify;
  Rule.choice [
    dobackl;
    Rule.compose_pairwise unfold_gs
      [Rule.attempt !axioms; (Rule.first [symex;symex_empty_axiom])];
    unfold_fs;
    symex;
    (Rule.compose (unfold_pre) (Rule.attempt ex_falso_axiom));
    disjunction_rule;
    conjunction_rule;
  ];
]
```

We first attempt to apply the (Split) rule to break left-hand side disjunction formulas. We then attempt to simplify the precondition to remove any redundant equalities. We then choose between

- Attempting to form a backlink by matching the sequent to a previously discovered syntactically identical sequent;
- Applying EG/AG unfolding rules;
- Applying EF/AF unfolding rules;
- Applying a symbolic execution rule;
- Unfold an inductive predicate on the symbolic heap formula;
- Applying ($\vee$) rule;
- Applying (Conj) rule.

As a small optimisation of the proof search algorithm, we speed up the proof search by exploiting common patterns that arise from the structure of the rules. For example, in the application of rules (AG) and (EG), the sequent in the left-hand premise strips the preceding AG/EG temporal operator in the CTL formula. This usually leads to a sequent whose temporal property is simply a symbolic heap formula, which we usually discharge by the use of (Check) axiom. On the other hand, the sequent on the right-hand premise prepends a $\Box$ or $\Diamond$ operator to the temporal formula. Sequents like this require the use of a symbolic execution rules to discharge them. Such optimisation is reflected in our proof tactic listed in the previous code snippet.

A second optimisation arises in the application of unfolding an inductive predicate in the symbolic heap formula. Since such unfolding can lead to inconsistencies in the precondition formula, we attempt to discharge the premise of the (Unfold-Pre) rule via the (Ex. Falso) axiom.

Other common patterns that could lead to optimisations arise on specific scenarios, specifically in the formation of backlinks, which tend to occur at the start of `while` loops. As forming backlinks is an expensive operation, one can be tempted to limit their application to parts of the the proof search that involve a `while` loop. Nevertheless, in doing so, we would also lose the capability of performing lemma discovery in other parts of the program. Consequently, adding specific optimisations could be beneficial in speeding up some proof searches but would hinder performance in some other cases. Allowing the user to choose between different proof search heuristics, or better yet define their own, is a potential idea for future work, but falls out of the scope of this work.

6.2.1 Automated Soundness Check

When cycles are formed in the proof search the implementation automatically checks for the soundness of the resulting proof graph. In performing this operations, CYCLIST first simplifies the proof graph by stripping each node of its sequent structure, keeping only the information related to the tagpairs in each sequent along with a unique identifier for each NODE in the graph. From this graph, two Büchi automata are built to check for the global soundness of the proof graph: the first one accepts all infinite strings over node ids. The second one accepts all infinite strings over node ids such that a progressing trace exists along the path. For the details of this construction and its soundness proof we refer the reader to Section 5.2.1. CYCLIST then relies on the SPOT [52] model checker tool to verify that the language accepted by the first Büchi automata is a sublanguage of the language accepted by the second Büchi automata.

Both the CTL and LTL implementations of our proof systems were able to make use of the CYCLIST infrastructure to automatically check the global soundness of the proof graph, but the implementation of the systems aware of fairness constraints required to make some changes to properly check for soundness.

Recall from Definition 5.2.4 that a fair cyclic proof meets two conditions: (i) not to be bad (according to Definition 5.2.3) and (ii) for every infinite *fair* path $(P_i \vdash_f C_i : \varphi_i)_{i\geq 0}$ in $\mathcal{P}$, there is an infinitely progressing $\Box$-trace, $\Diamond$-trace or precondition trace following some tail $(P_i \vdash_f C_i : \varphi_i)_{i\geq n}$ of the path.

Checking the validity of condition (i) requires a very similar approach to that of a standard cyclic proof system, in that it requires to check for a relation of languages between two Büchi automata. As such, we omit the details of its implementation.

Checking for the validity of condition (ii) requires a more involved approach. To check for it, we construct a Streett automaton that accepts all infinite strings over node ids such that whenever the rule $((\text{Wh}^* \Diamond 1))/(\text{If}^* \Diamond 1)$ is applied infinitely often to some command occurrence, then the rule $(\text{Wh}^* \Diamond 2)/(\text{If}^* \Diamond 2)$ is applied infinitely often, or viceversa. To this effect, we keep a set of fairness constraints (tuples of node Ids) for each pair of rule applications $(\text{Wh}^* \Diamond 1, \text{Wh}^* \Diamond 2)$ or $(\text{If}^* \Diamond 1, \text{If}^* \Diamond 2)$ respectively as follows:

```
void FairProof::set_fairness_constraint(const Vertex & v1,
                                        const Vertex & v2,
                                        int c1, int c2) {
  assert( vertices.find(v1) != vertices.end() );
  assert( vertices.find(v2) != vertices.end() );
  acc_set_map[v1].insert(c1);
  acc_set_map[v2].insert(c2);
  fairness_constraints.insert(FairnessConstraint(c1, c2));
}
```

Using this set, we establish the accepting condition of the Street automata by conjuncts of the form $(\mathrm{Fin}(i) \vee \mathrm{Inf}(j)) \wedge (\mathrm{Fin}(j) \vee \mathrm{Inf}(i))$ for each pair of fairness constraints (i, j).

```
void FairProofAutomaton::set_acceptance_condition() {
  std::stringstream acceptance_condition;
  std::unordered_set<FairnessConstraint> fairness_constraints =
      get_fairness_constraints();
  for(auto elem = fairness_constraints.begin();
      elem != fairness_constraints.end(); ++elem) {
        if(elem != fairness_constraints.begin()) {
          acceptance_condition << "␣&␣";
        }
        acceptance_condition << "(Fin(" << (std::get< 0 >(*elem))
        << ") | Inf(" << (std::get< 1 >(*elem))
        << ")) & (Fin(" << (std::get< 1 >(*elem))
        << ") | Inf(" << (std::get< 0 >(*elem))
        << "))" << std::flush;
      }
      set_acceptance(get_max_acc_elem(),
                     spot::acc_cond::acc_code(acceptance_condition.str()));
}
```

Using this Streett automaton, we can check that its accepted language is a subset of the language accepted by the Büchi automata that accepts infinite strings over node ids such that a progressing trace exists along the path. The following code snippet shows the body of our procedure that checks the soundness of our fair CTL proof system.

```
extern "C" value check_fair_soundness_second_condition() {
  CAMLparam0();
  CAMLlocal1(v_res);

  //Build a Streett automata and set its acceptance condition
  proof->set_acceptance_condition();
  spot::twa_graph_ptr street_graph = copy(proof, spot::twa::prop_set::all());

  // Build a trace automata from the proof graph
  spot::const_twa_ptr trace_aut = std::make_shared<TraceAutomaton>(*proof);
  spot::twa_graph_ptr t_graph = copy(trace_aut, spot::twa::prop_set::all());

  // Transform the Streett to a generalised buchi automata
  spot::twa_graph_ptr proof_sgba = to_generalized_buchi(street_graph);
```

```
    // Build the complement of the trace automata
    // and transform it to a generalized buchi automata
    spot::twa_graph_ptr proof_tgba = to_generalized_buchi(
      dtwa_complement(tgba_determinize(
        t_graph, false, true, true,
        spot::check_stutter_invariance(graph).is_true())));

    // Compute the product of both generalised Buchi automata
    spot::const_twa_ptr product =
      std::make_shared<spot::twa_product>(proof_sgba, proof_tgba);

    // Check for emptiness of the acceptance language of the product automata
    spot::couvreur99_check ec(product);
    std::shared_ptr<spot::emptiness_check_result> res = ec.check();

    // Return result
    bool retval = (res == 0);
    v_res = Val_bool(retval);
    CAMLreturn(v_res);
}
```

In this chapter we have discussed the implementation details of the CTL cyclic proof system. The implementation is built on top of the CYCLIST theorem prover and it mainly consists of a symbolic execution engine (embedded in the logic of our symbolic execution rules) and a proof search algorithm, whose main goal is to close all open nodes in the proof graph and to automatically verify the soundness of the proof. The source code is freely available at [45].

In the following chapter, we use our automated verification tool in proving temporal properties of common benchmarks found in the literature, showing the suitability of our approach for verifying temporal properties of programs with access to the heap.

Chapter 7

Experimental Results

In this chapter we evaluate the implementation of our cyclic proof systems on handcrafted nondeterministic and nonterminating programs similar to Example 1.1.1. Our test suite can be seen as an adaptation of the common model checking benchmarks presented in [40, 41] for the verification of temporal properties of nondeterministic programs, where operations/iterations on integer variables in the original benchmarks are replaced in favour of operations/iterations on heap data structures. Figure 7 demonstrates the adaptation of a particular program example, where the original implementation of the acquire-release example from [40] is compared to our heap manipulation adaptation. In particular, some of the changes include: (i) stack variables `A` and `R` are replaced in favour of heap pointers; (ii) iteration over integer variables is replaced in favour of iteration over heap structures (in this particular adaptation, the inside loop iterates over a nil terminating list (i.e. ls(x,nil)) of nondeterministic length); (iii) assignment of nondeterministic values that control the exit of the loops is replaced in favour of nondeterministic control structures that determine the value of the loop guards.

In the remainder of this chapter we describe the full set of program adaptations, similar to the example aforementioned, that compose our full benchmark suite. The details of the experiments for our CTL cyclic proof system, as well as its adaptation to fairness executions, are the subject of discussion in Section 7.1. We then presents the details of the experiments for our LTL cyclic proof system, along with its fairness adaptation, in Section 7.2.

Listing 7.1: Original implementation

```
__rho_1_ = nondet();
dobreak = __rho_1_;
while(1) {
  if (dobreak > 0) break;
  A = 1;
  A = 0;
  __rho_1_ = nondet();
  n = __rho_1_;
  while(1) {
    if (!(n>0)) break;
    n--;
  }
  R = 1;
  R = 0;
  __rho_1_ = nondet();
  dobreak = __rho_1_;
}
while(1) {
  dummy=dummy;
}
```

Listing 7.2: Adaptation

```
while(flag!=nil) {
  if( * ) {
      free(flag);
  else {
      A:=new();
      free(A);
      while(x!=nil) {
          temp:=x;
          x:=x.next;
          free(temp);
      }
      R:=new();
      free(R);
  };
};
while(flag=flag) {
  dummy:=dummy;
}
```

Figure 7.1: Comparison between the original implementation of a sample program and our heap manipulation adaptation used in the experiments

7.1 CTL cyclic proof system experiments

Our test suite for our CTL cyclic proof system comprises the following programs:

(i) Examples discussed in this thesis are named EXMP;

(ii) FIN-LOCK is a finite program that acquires a lock and, once obtained, proceeds to free from memory the elements of a list and reset the lock;

(iii) INF-LOCK wraps the previous program inside an infinite loop;

(iv) ND-IN-LOCK is an infinite loop that nondeterministically acquires a lock, then proceeds to perform a nondeterministic number of operations before releasing the lock;

(v) INF-LIST is an infinite loop that nondeterministically adds a new element to the list or advances the head of the list by one element on each iteration;

(vi) INSERT-LIST has a nondeterministic if statement that either adds a single elements to the head of the list or deletes all elements but one, and is followed by an infinite loop;

Program	Precondition	Property	Fairness	Time
EXMP	ls(x,nil)	AGEF emp	No	2.43
EXMP	ls(x,nil)	AGAF emp	Yes	4.29
EXMP	ls(x,nil)	AGAF (ls(x,nil))	No	0.26
EXMP	ls(x,nil)	AGEG (ls(x,nil))	No	0.44
EXMP	ls(x,nil)	AF emp	Yes	0.77
EXMP	ls(x,nil)	AFEG emp	Yes	0.86
FIN-LOCK	lock↦0 * ls(x,nil)	AF (lock↦1 * emp)	No	0.20
FIN-LOCK	lock↦0 * ls(x,nil)	AGAF (lock↦1 * emp)	No	0.62
FIN-LOCK	lock↦0 * ls(x,nil)	AGAF (lock↦1 * emp ∧ ◇lock↦0)	No	0.24
INF-LOCK	lock↦0 * ls(x,nil)	AGAF (lock↦1 * emp)	No	1.52
INF-LOCK	lock↦0 * ls(x,nil)	AGAF (lock↦1 * emp ∧ ◇lock↦0))	No	3.26
INF-LOCK	del=false : lock↦0 * ls(x,nil)	AG (del!=true ∨ AF (lock↦1))	No	3.87
ND-INF-LOCK	lock↦0	AF(lock↦1)	Yes	0.15
ND-INF-LOCK	lock↦0	AGAF (lock↦1)	Yes	0.25
INF-LIST	ls(x,nil)	AG ls(x,nil)	No	0.21
INF-LIST	ls(x,nil)	AGEF x=nil	No	4.39
INF-LIST	ls(x,nil)	AGAF x=nil	Yes	8.10
INSERT-LIST	ls(three,zero)	EF ls(five,zero)	No	0.14
INSERT-LIST	ls(three,zero)	AF ls(five,zero)	Yes	0.26
INSERT-LIST	ls(n,zero)	AGAF n!=zero	Yes	17.21
APPEND-LIST	ls(y,x) * ls(x,nil)	AF (ls(y,nil))	No	12.67
CYCLIC-LIST	cls(x,x)	AG cls(x,x)	No	0.88
CYCLIC-LIST	cls(x,x)	AGEG cls(x,x)	No	0.34
INF-BINTREE	x!=nil : bintree(x)	AGEG x!=nil	No	0.72
AFAG BRANCH	x↦zero	AFAG x↦one	No	1.80
EGAG BRANCH	x↦zero	EGAG x↦one	No	0.23
EGAF BRANCH	x↦zero	EGAF x↦one	No	15.48
EG⇒ EF BRANCH	p=zero ∧ q=zero : ls(zero,n)	EG(p!=one ∨ EF q=one)	No	1.60
EG⇒ AF BRANCH	p=zero ∧ q=zero : ls(zero,n)	EG(p!=one ∨ AF q=one)	Yes	5.33
AG⇒ EG BRANCH	p=zero ∧ q=one : ls(zero,n)	AG(p!=one ∨ EG q=one)	No	0.36
AG⇒ EF BRANCH	p=zero ∧ q=one : ls(zero,n)	AG(p!=one ∨ EF q=one)	No	1.53
ACQ-REL	ls(zero,three)	AG(acq=0 ∨ AF rel!=0)	No	1.25
ACQ-REL	ls(zero,three)	AG(acq=0 ∨ EF rel!=0)	No	1.25
ACQ-REL	ls(zero,three)	EF acq!=0 ∧ EF AG rel=0	No	0.33
ACQ-REL	ls(zero,three)	AF AG rel=0	Yes	0.42
ACQ-REL	ls(zero,three)	EF acq!=0 ∧ EF EG rel=0	No	0.25
ACQ-REL	ls(zero,three)	AF EG rel=0	Yes	0.33
POSTGRESQL	w=true ∧ s=s' ∧ f=f' : emp	AGAF w=true ∧ s=s' ∧ flag=f'	No	0.27
POSTGRESQL	w=true ∧ s=s' ∧ f=f' : emp	AGEF w=true ∧ s=s' ∧ flag=f'	No	0.26
POSTGRESQL	w=true ∧ s=s' ∧ f=f' : emp	EFEG w=false ∧ s=s' ∧ flag=f'	No	0.44
POSTGRESQL	w=true ∧ s=s' ∧ f=f' : emp	EFAG w=false ∧ s=s' ∧ flag=f'	No	0.77
WIN UPDATE	W!=nil : ls(W,nil)	AGAF W!=nil : ls(W,nil)	No	1.50
WIN UPDATE	W!=nil : ls(W,nil)	AGEF W!=nil : ls(W,nil)	No	1.00
WIN UPDATE	W!=nil : ls(W,nil)	EFEG W=nil : emp	No	3.60
WIN UPDATE	W!=nil : ls(W,nil)	AFEG W=nil : emp	Yes	3.70
WIN UPDATE	W!=nil : ls(W,nil)	EFAG W=nil : emp	No	3.15
WIN UPDATE	W!=nil : ls(W,nil)	AFAG W=nil : emp	Yes	4.16

Table 7.1: Experimental results for (fair) CTL system.

(vii) APPEND-LIST appends the second argument to the end of the first argument;

(viii) CYCLIC-LIST is a nonterminating program that iterates through a non-empty cyclic list;

(ix) INF-BINTREE is an infinite loop that nondeterministically inserts nodes to a binary tree or performs a random walk of the tree;

(x) The programs named with BRANCH define a somewhat arbitrary nesting of nondeterministic `if` and `while` statements, aimed at testing the capability of the tool in terms of lines of code and nesting of cycles;

(xi) Finally we also cover adaptations from sample programs taken from the Windows Update system (WIN UPDATE), the back-end infrastructure of the PostgreSQL database server (POSTGRESQL) and an implementation of the acquire-release algorithm (ACQ-REL) taken from the aforementioned benchmarks.

We show the results of the evaluation of the CTL system and its extension to consider fairness constraints in Table 7.1. For each test, we report whether fairness constraints were needed to verify the desired property and the time taken in seconds. The tests were carried out on an Intel x-64 i5 system at 2.50GHz.

7.2 LTL cyclic proof system experiments

Our test suite for our LTL cyclic proof system comprises the following programs:

(i) Examples discussed in this thesis are named EXMP;

(ii) Benchmarks obtained from model checking tools for Windows components are named WIN;

(iii) PG_BUFFER and PG_ARCH are modules of the PostgreSQL database backend

(iv) Finally APACHE is a component of the web server in charge of serving connection requests.

Program	Precondition	Property	Fairness	Time
Exmp2	x=true	FG x=true	No	0.02
Exmp2_Fair	x=true	FG x=true	Yes	0.04
Exmp4	x=y	FG x=y	No	0.05
Exmp4_Fair	x=y	FG x=y	Yes	0.15
Exmp5.3.4	y=false ∧ x↦x'	FG x=true	Yes	0.06
Win1	acq ↦ false * rel ↦ false	G(acq ↦ true ⇒ F rel ↦ true)	No	355.62
Win2	flag ↦ false	G(flag ↦ true ⇒ F flag ↦ false)	No	183.46
Win3	flag ↦ false * stored ↦ zero	FG(stored ↦ zero)	No	0.05
Win4	acq ↦ false * rel ↦ false	G(acq ↦ true ⇒ F rel ↦ true)	No	17.91
Win4_Fair	acq ↦ false * rel ↦ false	G(acq ↦ true ⇒ F rel ↦ true)	Yes	15.02
Win4_Simpl	acq ↦ false * rel ↦ false	F(acq ↦ true) ∨ F(rel ↦ true)	No	0.04
Win6	dll(zero,five) * W↦ nil,nil	FG W=0	No	0.05
Win6_Fair	dll(zero,five) * W↦ nil,nil	FG W=0	Yes	0.55
Win7	dll(zero,five) * W↦ nil,nil	GF W>1	No	0.04
Win8	ls(zero,five) * count=0	G(count>0 ⇒ count=0)	No	827.63
PG_buffer	istemp ↦ false * acq ↦ false	G(istemp ↦ true ⇒ acq ↦ false)	No	0.88
PG_arch	w↦zero	GF w↦one	No	0.28
PG_arch_Fair	w↦zero	GF w↦one	Yes	0.35
Apache	dnow ↦ false * acc ↦ false	G(dnow ↦ false ⇒ F acc=true)	No	6.35

Table 7.2: Experimental results for (fair) LTL system.

In Table 7.2 we show the results of the verification of temporal properties of these programs in our LTL cyclic proof system implementation. For each test, we report whether fairness constraints were needed to verify the desired property and the time taken in seconds. The tests were carried out on an Intel x-64 i5 system at 2.50GHz.

Our experiments demonstrate the viability of our approach: our runtimes are mostly in the range of milliseconds and show similar performance to existing tools for the model checking benchmarks. Overall, the execution times in the evaluation are quite varied as they depend on factors such as the complexity of the program and the temporal property, but sources of potential slowdown can be witnessed by different test cases.

Even at the level of pure memory assertions, the base case rule (Check) has to check entailments of the form $P \models Q$ between symbolic heaps, which involves calling an inductive theorem prover; this is reasonably fast in some cases, but very costly in others (e.g. the Append-List example). A possible improvement to our implementation could attempt to reduce the cost of this operation by use of a cache of entailment checks. In this way, if during the proof search, some entailment has been previously checked, we can instantly obtain the result from memory and avoid recomputing the result. This feature is nonetheless out of the scope of this work, and it is left for future work.

Another source of slowdown is in attempting to form back-links too eagerly (e.g. when encountering the same command at two different program locations); since we check soundness when forming a back-link, which involves calling a model checker (cf. [22]), this too is an expensive operation, as can be seen in the runtimes of test cases with suffix BRANCH. One can be tempted to limit the application of backlinks to parts of the proof search that involve program iterations, but by doing so we also limit the capability of lemma discovery for other parts of the program. Allowing the user to choose between different proof search heuristics, or better yet define their own, is a potential idea for future work, but falls out of the scope of this work.

Finally, note that despite the encouraging results, the implementation is not without limitations; it might, in some cases, fail to terminate and produce a valid proof. Generalising, our proof search tends to fail either when the temporal property in question does not hold, or when we fail to establish a sufficiently general "invariant" to form backlinks in the proof. The addition of counterexamples when a temporal property does not hold is another feature that could improve the usability of our implementation, and presents the opportunity for further work.

Chapter 8

General Conclusions

The aim of this thesis has been to devise a sound and fully automated temporal verification framework for infinite heap-aware programs. The main body of this work was dedicated to describing in detail our formulation, soundness, implementation and evaluation of this framework. In this chapter, we will recapitulate how our aim was fulfilled in line with the list of objectives outlined in Chapter 1. Then we will propose some possible lines of work for further developing the ideas presented in this thesis.

8.1 Contributions

Generality. Our proof system has been designed to handle arbitrary safety, liveness and fairness properties of heap-manipulating programs. As such, the present work could be seen as a generalisation of the cyclic proof system for proving termination of heap-manipulating programs proposed by Brotherston, Bornat and Calgano in [19]. Compared to this previous work, our judgement structure is expanded to include arbitrary temporal properties and the notion of proof traces is expanded to account for the progress induced by the analysis of temporal operators.

Moreover, our temporal verification framework includes specific instances tailored to both LTL and CTL temporal logics. The formulation of these instances aided our coverage of the full range of temporal logics, namely linear and branching time logics, but it naturally leaves other temporal logics out of the scope of this work.

Soundness. A benefit of using a deductive verification approach at the core of our framework is that the soundness of our system is reduced to:

1. checking for local soundness of each rule separately (along with properties on traces following the path, as was done in Lemma 3.3.1 and Lemma 4.2.1, and
2. checking the global soundness condition of pre-proofs to validate that each infinite cycle in the proof is, indeed, a valid cycle, as was done in Theorem 3.3.2 and Theorem 4.2.2.

Verifying the soundness of the cyclic temporal verification framework is, we would argue, quite reasonable.

One of the advantages of our approach is that we never obtain false positive results. This advantage is, as expected, not exclusive to deductive verification, as some automata-theoretic model checking approaches are also sound [96]. Nonetheless, when compared to such approaches, we believe our treatment of the temporal verification problem to be more natural, as we avoid both the translation of temporal formulas into complex automata [97] and the instrumentation of the original program with auxiliary constructs [39].

Memory awareness. The decision to rely on separation logic for the handling of heap-manipulating programs served two purposes. First, the use of an established and well-studied framework significantly lowered the effort of formulating our symbolic-execution proof rules, as they are mostly adaptations of similar rules used on previous proof systems for separation logic verification.

Second, thanks to the combined use of separation logic and a deductive proof system, our approach does not need to apply approximation or transformations to the program before attempting to verify it. This direct treatment of the original program and temporal formula has proven beneficial in the treatment of other aspects of temporal logic verification, say fairness, as we have demonstrated. Most importantly, the structure of the proof rules and the avoidance of complex side conditions were also a key factor in the automation of our framework.

Infinite state. The common benefits that cyclic proofs bring as an alternative to traditional proofs by explicit induction made a big impact on various aspects of our framework. A subtle but important one has been the transfer from a proof system that would have only capable of handling finite state systems to handling infinite state.

Moreover, the machinery built into cyclic proofs is the main factor behind the relative simplicity of the individual proof rules, which, compared to early approaches [74, 76, 57], do not rely on complex side verification conditions that require the computation of program invariants and ranking functions. In its place, the notion of traces embed the notion of a ranking function within the proof itself, allowing for its discovery as the proof is carried out.

In our particular proof systems, such notion corresponds to the progress made by the symbolic execution of the program under analysis. At each point where a symbolic execution rule is applied we say that our trace progresses. The global soundness conditions then guarantees that every infinite loop is indeed progressing infinitely often, leading to a proof that can be read as a proof by *infinite descent* à la Fermat. We note that this notion of progress is not unique, having a similar notion based on the infinite unfolding of a fixpoint operator along infinite paths, similar to our precondition traces. This is based on the premise that temporal formula $AG\varphi(EG\varphi)$ can be characterised as the greatest fixpoint operator of the form $\nu Z.\varphi \cap AXZ(\nu Z.\varphi \cap EXZ)$. Then since along our $\Box-(\Diamond-)$ trace an $AG(EG)$ formula is unfolded infinitely often, this infinite unfolding would too lead to a proof by *infinite descent*. A similar argument applies to the case of LTL G formulas and their corresponding LTL-traces.

The switch from explicit induction proofs to cyclic proofs also played an important role in the automation of our proof systems, mostly due to the generic cyclic theorem prover CYCLIST . This framework provided the tools required for the proof search algorithm, as well as the basis for the implementation of the decision procedure to check the decidability of our fair global soundness condition. We also note that, to the best of our knowledge, *we are the first automated cyclic proof theorem* *prover to address fairness conditions.*

Full automation. Throughout this thesis we have mentioned a few related works that have previously addressed deductive verification to the temporal verification problem. In particular, the present thesis could be seen as a specialisation of the work by Sprenger in [93] where we favour the analysis of heap-manipulating programs written in a specific programming language rather than allowing arbitrary transition system constructs. This resolution played a critical role in the ability to produce a fully automated implementation of a cyclic deductive verification tool for verification of arbitrary temporal properties of programs. To the best of our knowledge, we are the first to produce a system with this characteristics.

8.2 Future work

The vast number of proposed temporal logics for the verification of programs, of which LTL and CTL comprise only a small part, could be an indication for a possible line of future work: the enrichment of our temporal logic to other temporal logics. In particular, due to their higher level of expressiveness, CTL* [54] and μ-calculus [46] are prime candidates for such an undertaking.

In terms of CTL*, the structure of its formulas and their respective classification into path and state subformulas suggest a possible combination of our LTL and CTL systems to produce a proof object composed of smaller proof structures as was suggested in [13, 93]. The encoding of CTL* into μ-calculus[46] and the applicability of cyclic proofs for the verification of μ-calculus properties (see e.g. [88]) suggest the feasibility of such an extension.

If someone were to follow our approach, combining CTL* / μ-calculus with the expressivity of separation logic to handle the analysis of heap-aware programs, a second natural direction for extending our work is to consider larger classes of programs for which both logics have been successfully applied. In particular, programming languages with concurrency constructs are suggested as a very interesting direction for future work. At first sight it would seem that most of the challenge would be to provide appropriate constructs/restrictions on parallel program composition mainly due to the lack of compositionality of temporal logic. One could, for example, be tempted to build on the parallel composition rule of concurrent separation logic as shown in [94], which roughly takes the form of

$$\frac{P_1 \vdash C_1 : Q_1 \quad P_2 \vdash C_2 : Q_2}{P_1 * P_2 \vdash C_1 || C_2 : Q_1 * Q_2} \text{ (Par)}$$

with some side conditions that restrict the variables accessed by each separate program component. Unfortunately, it is not clear which operator (if any) would allow for such composition of arbitrary temporal formulas φ_1/φ_2 in place of separation logic assertions Q_1/Q_2.

A third major piece of future work that would be of great advantage is related to the manual construction of the proofs required to demonstrate the soundness of our system. One could argue that the manual elaboration of such proofs is in itself error prone. Whereas we took great care in our soundness proof and feel strongly confident about their correctness, we also appreciate there is room for improvement in terms of proof mechanisation. A possible implementation of this kind could directly translate each proof rule into a proposition in a mechanised proof assistant. The elaboration of their corresponding proofs could be carried out by a suitable set of tactics based on the semantics of both the programming language and the temporal logic in question (one would expect, following a very similar structure to the proofs produced by hand). Most of the technical difficulty of this task, perhaps, would be the representation of the structures needed in cyclic proofs (i.e. pre-proofs, directed acyclic graphs, proof traces) as explicit datatypes to aid in proving the global soundness condition of cyclic proofs. Therefore, it is important to ponder the increased level of confidence one would gain when compared to the effort required.

Finally, regarding automation, our implementation has opened the door to enhancements that would improve its performance and increase its feature set. The heavy use of entailment checks in our system, along with the expensive cost of this operation, suggest that adding a cache of entailment checks to avoid recomputing known results might improve the performance of the automated tool. A second line of work to increase the performance of our proof search algorithm involves developing improved mechanised techniques, such as generalisation / abstraction, to allow re-use of previously discovered cyclic proofs as lemmas. In terms of adding features to the implementation, conceivably the most desirable missing feature is the ability to produce counterexamples in case the program in question does not meet its desired temporal specification. This feature could be of great benefit for troubleshooting and finding software bugs.

Appendix A

Colophon

This document was created using LaTeX and BibTeX, composed with Emacs editor.

The figures appearing in this thesis were produced with `gastex` package and the proofs were produced using Paul Taylor's `prooftree` package.

Bibliography

[1] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *THEORETICAL COMPUTER SCIENCE*, 138:3–34, 1995.

[2] R. Alur, T.A. Henzinger, and Pei-Hsin Ho. Automatic symbolic verification of embedded systems. *Software Engineering, IEEE Transactions on*, 22:181–201, 1996.

[3] Alessandro Armando, Jacopo Mantovani, and Lorenzo Platania. Bounded model checking of software using smt solvers instead of sat solvers. *Int. J. Softw. Tools Technol. Transf.*, 11:69–83, 2009.

[4] Jürgen Avenhaus, Ulrich Kühler, Tobias Schmidt-Samoa, and Claus-Peter Wirth. How to prove inductive theorems? QuodLibet! In Franz Baader, editor, *Proceedings of CADE-19*, number 2741 in LNAI, pages 328–333. Springer, 2003.

[5] Domagoj Babic and Alan J. Hu. Calysto: scalable and precise extended static checking. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 211–220, 2008.

[6] Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 113–130. Springer-Verlag, 2000.

[7] Thomas Ball and Sriram K. Rajamani. The slam project: Debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 1–3. ACM, 2002.

[8] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. A decidable fragment of separation logic. In *Proceedings of FSTTCS-24*, pages 97–109. Springer-Verlag, 2004.

[9] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects*, FMCO'05, pages 115–137. Springer-Verlag, 2006.

[10] Josh Berdine, Byron Cook, Distefano, and W. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *Computer Aided Verification, 18th International Conference, CAV*, pages 386–400. Springer, 2006.

[11] Josh Berdine, Byron Cook, and Samin Ishtiaq. Slayer: Memory safety for systems-level code. In *Proceedings of CAV-23*, pages 178–183. Springer-Verlag, 2011.

[12] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker BLAST: Applications to software engineering. *Int. J. Softw. Tools Technol. Transf.*, 9:505–525, 2007.

[13] G. Bhat, R. Cleaveland, and O. Grumberg. Efficient on-the-fly model checking for CTL*. In *Proceedings of LICS-10*, pages 388–397. IEEE, 1995.

[14] Ahmed Bouajjani, Jean-Claude Fernandez, and Nicolas Halbwachs. Minimal model generation. In *Proceedings of the 2Nd International Workshop on Computer Aided Verification*, CAV '90, pages 197–203. Springer-Verlag, 1991.

[15] Julian Bradfield and Colin Stirling. Local model checking for infinite state spaces. *Theoretical Computer Science*, 96:157 – 174, 1992.

[16] James Brotherston. *Sequent Calculus Proof Systems for Inductive Definitions*. PhD thesis, University of Edinburgh, November 2006.

[17] James Brotherston. Formalised inductive reasoning in the logic of bunched implications. In *Proceedings of SAS-14*, volume 4634 of *LNCS*, pages 87–103. Springer-Verlag, 2007.

[18] James Brotherston, Richard Bornat, and Cristiano Calcagno. Cyclic proofs of program termination in separation logic. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 101–112. ACM, 2008.

[19] James Brotherston, Richard Bornat, and Cristiano Calcagno. Cyclic proofs of program termination in separation logic. In *Proceedings of POPL-35*, pages 101–112. ACM, 2008.

[20] James Brotherston, Dino Distefano, and Rasmus L. Petersen. Automated cyclic entailment proofs in separation logic. In *Proceedings of CADE-23*, volume 6803 of *LNAI*, pages 131–146. Springer, 2011.

[21] James Brotherston, Carsten Fuhs, Juan A. Navarro Pérez, and Nikos Gorogiannis. A decision procedure for satisfiability in separation logic with inductive predicates. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14, pages 25:1–25:10. ACM, 2014.

[22] James Brotherston, Nikos Gorogiannis, and Rasmus L. Petersen. A generic cyclic theorem prover. In *Proceedings of APLAS-10*, LNCS, pages 350–367. Springer, 2012.

[23] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35:677–691, 1986.

[24] Tevfik Bultan, Richard Gerber, and William Pugh. Symbolic model checking of infinite state systems using presburger arithmetic. In *CAV*, volume 1254 of *Lecture Notes in Computer Science*, pages 400–411. Springer, 1997.

[25] Tevfik Bultan, Richard Gerber, and William Pugh. Model-checking concurrent systems with unbounded integer variables: Symbolic representations, approximations, and experimental results. *ACM Trans. Program. Lang. Syst.*, 21:747–789, 1999.

[26] Doron Bustan and Orna Grumberg. Simulation-based minimization. *ACM Trans. Comput. Logic*, 4:181–206, 2003.

[27] Cristiano Calcagno, Dino Distefano, Jeremy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter OHearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In Klaus Havelund, Gerard Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 9058 of *Lecture Notes in Computer Science*, pages 3–11. Springer International Publishing, 2015.

[28] Cristiano Calcagno, Hongseok Yang, and Peter W. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In *Proceedings of the 21st Conference on Foundations of Software Technology and Theoretical Computer Science*, FST TCS '01, pages 108–119. Springer-Verlag, 2001.

[29] Sagar Chaki, Edmund Clarke, Alex Groce, and Ofer Strichman. Predicate abstraction with minimum predicates. In Daniel Geist and Enrico Tronci, editors, *Correct Hardware Design and Verification Methods*, volume 2860 of *Lecture Notes in Computer Science*, pages 19–34. Springer Berlin Heidelberg, 2003.

[30] Bor-Yuh Evan Chang and Xavier Rival. Relational inductive shape analysis. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 247–260. ACM, 2008.

[31] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Sci. Comput. Program.*, 77:1006–1036, 2012.

[32] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Proceedings of TACAS*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.

[33] Edmund Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of the 40th Annual Design Automation Conference*, DAC '03, pages 368–371. ACM, 2003.

[34] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71. Springer-Verlag, 1981.

[35] Edmund M. Clarke, Thomas Filkorn, and Somesh Jha. Exploiting symmetry in temporal logic model checking. In *Proceedings of the 5th International Conference on Computer Aided Verification*, CAV '93, pages 450–462. Springer-Verlag, 1993.

[36] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Proceedings of the 12th International Conference on Computer Aided Verification*, CAV '00, pages 154–169. Springer-Verlag, 2000.

[37] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[38] B. Cook, H. Khlaaf, and N. Piterman. On automation of CTL* verification for infinite-state systems. In *Proceedings of CAV-27*, volume 9206 of *LNCS*. Springer, 2015.

[39] Byron Cook, Alexey Gotsman, Andreas Podelski, Andrey Rybalchenko, and Moshe Y. Vardi. Proving that programs eventually do something good. In *Proceedings of POPL-07*, POPL 07, pages 265–276. ACM, 2007.

[40] Byron Cook and Eric Koskinen. Making prophecies with decision predicates. In *Proceedings of POPL-38*, volume 46, pages 399–410. ACM, 2011.

[41] Byron Cook and Eric Koskinen. Reasoning about nondeterminism in programs. In *Proceedings of PLDI-34*, pages 219–230. ACM, 2013.

[42] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252. ACM, 1977.

[43] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astrée analyzer. In *European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer-Verlag, 2005.

[44] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, pages 84–96. ACM, 1978.

[45] CYCLIST: software distribution. `https://github.com/ngorogiannis/cyclist/`.

[46] M. Dam. *Translating CTL* Into the Modal Mu-calculus*. ECS-LFCS-.

University of Edinburgh, Department of Computer Science, Laboratory for Foundations of Computer Science, 1990.

[47] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340. Springer-Verlag, 2008.

[48] Giorgio Delzanno and Andreas Podelski. Model checking in CLP. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 223–239. Springer-Verlag, 1999.

[49] Dino Distefano, Joost-Pieter Katoen, and Arend Rensink. Who is pointing when to whom? In *FSTTCS 2004: Foundations of Software Technology and Theoretical Computer Science*, volume 3328 of *Lecture Notes in Computer Science*, pages 250–262. Springer Berlin Heidelberg, 2005.

[50] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'06, pages 287–302. Springer-Verlag, 2006.

[51] Dino Distefano and Matthew J. Parkinson J. jStar: Towards practical verification for java. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 213–226. ACM, 2008.

[52] Alexandre Duret-Lutz, Alexandre Lewkowicz, Amaury Fauchille, Thibaud Michaud, Etienne Renault, and Laurent Xu. Spot 2.0 — a framework for LTL and ω-automata manipulation. In *Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA'16)*, volume 9938 of *Lecture Notes in Computer Science*, pages 122–129. Springer, 2016.

[53] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, pages 502–518, 2003.

[54] E. Allen Emerson and Joseph Y. Halpern. “Sometimes” and “Not never” revisited: On branching versus linear time temporal logic. *J. ACM*, 33:151–178, 1986.

[55] E. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. *Form. Methods Syst. Des.*, 9:105–131, 1996.

[56] Limor Fix and Orna Grumberg. Verification of temporal properties. *J. Log. Comput.*, 6:343–361, 1996.

[57] Dov M. Gabbay and Amir Pnueli. A sound and complete deductive system for CTL* verification. *Logic Journal of the IGPL*, 16:499–536, 2008.

[58] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag New York, Inc., 1996.

[59] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’97, pages 174–186. ACM, 1997.

[60] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Kenneth L. McMillan. Abstractions from proofs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’04, pages 232–244. ACM, 2004.

[61] Hardi Hungar, Orna Grumberg, and Werner Damm. What if model checking must be truly symbolic. In *Proceedings of CHARME*, pages 1–20. Springer-Verlag, 1995.

[62] C. Norris Ip and David L. Dill. Better verification through symmetry. *Form. Methods Syst. Des.*, 9:41–75, 1996.

[63] F. Ivančić, Z. Yang, M. K. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar. F-soft: Software verification platform. In *Proceedings of the 17th International Conference on Computer Aided Verification*, CAV'05, pages 301–306. Springer-Verlag, 2005.

[64] Franjo Ivančić, Zijiang Yang, Malay K. Ganai, Aarti Gupta, and Pranav Ashar. Efficient SAT-based bounded model checking for software verification. *Theor. Comput. Sci.*, 404:256–274, 2008.

[65] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Comput. Surv.*, 41:21:1–21:54, 2009.

[66] Yonit Kesten and Amir Pnueli. A compositional approach to CTL* verification. *Theor. Comput. Sci.*, 331:397–428, 2005.

[67] Robert P. Kurshan. *Computer-aided Verification of Coordinating Processes: The Automata-theoretic Approach*. Princeton University Press, 1994.

[68] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Software Eng.*, 3:125–143, 1977.

[69] Christof Löding. Methods for the transformation of ω-automata: Complexity and connection to second order logic, 2007.

[70] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Form. Methods Syst. Des.*, 6:11–44, 1995.

[71] Stephen Magill, Josh Berdine, Edmund Clarke, and Byron Cook. Arithmetic strengthening for shape analysis. In *Proceedings of the 14th International Conference on Static Analysis*, SAS'07, pages 419–436. Springer-Verlag, 2007.

[72] Stephen Magill, Ming-Hsien Tsai, Peter Lee, and Yih-Kuen Tsay. Automatic numeric abstractions for heap-manipulating programs. In *Proceedings of the 37th Annual Symposium on Principles of Programming Languages*, POPL '10, pages 211–222. ACM, 2010.

[73] Z Manna and Amir Pnueli. Verification of concurrent programs: A temporal proof system. Technical report, Stanford, CA, USA, 1983.

[74] Zohar Manna and Amir Pnueli. How to cook a temporal proof system for your pet language. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '83, pages 141–154. ACM, 1983.

[75] Zohar Manna and Amir Pnueli. Completing the temporal picture, 1991.

[76] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag New York, Inc., 1992.

[77] Kenneth L. McMillan. Lazy abstraction with interpolants. In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, pages 123–136. Springer-Verlag, 2006.

[78] Leonardo M. de Moura, Harald Rueß, and Maria Sorea. Lazy theorem proving for bounded model checking over infinite domains. In *Proceedings of the 18th International Conference on Automated Deduction*, CADE-18, pages 438–455. Springer-Verlag, 2002.

[79] Madanlal Musuvathi and Dawson R. Engler. Model checking large network protocol implementations. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI'04, pages 12–12. USENIX Association, 2004.

[80] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the 28th ACM*

SIGPLAN Conference on Programming Language Design and Implementation, PLDI '07, pages 446–455. ACM, 2007.

[81] Juan Antonio Navarro Pérez and Andrey Rybalchenko. Separation logic modulo theories. In *Proceedings of the 11th Asian Symposium on Programming Languages and Systems - Volume 8301*, pages 90–106. Springer-Verlag New York, Inc., 2013.

[82] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Abstract DPLL and abstract DPLL modulo theories. In *Proceedings of LPAR 2004*, pages 36–50. Springer, 2004.

[83] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE, 1977.

[84] Amir Pnueli and Yonit Kesten. A deductive proof system for CTL*. In *CONCUR 2002 — Concurrency Theory*, pages 24–40. Springer Berlin Heidelberg, 2002.

[85] Andreas Podelski and Andrey Rybalchenko. Armc: The logical choice for software model checking with abstraction refinement. In *Proceedings of the 9th International Conference on Practical Aspects of Declarative Languages*, PADL'07, pages 245–259. Springer-Verlag, 2007.

[86] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th CISP*, pages 337–351. Springer-Verlag, 1982.

[87] Reuben Rowe and James Brotherston. Automatic cyclic termination proofs for recursive procedures in separation logic. In *CPP*, pages 53–65. ACM, 2017.

[88] Ulrich Schopp and Alex Simpson. Verifying temporal properties using explicit approximants: Completeness for context-free processes. In *Proceedings of FoSSaCS*, pages 372–386. Springer, 2002.

[89] João P. Marques Silva and Karem A. Sakallah. GRASP-a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '96, pages 220–227. IEEE Computer Society, 1996.

[90] A. Prasad Sistla, Viktor Gyuris, and E. Allen Emerson. Smc: A symmetry-based model checker for verification of safety and liveness properties. *ACM Trans. Softw. Eng. Methodol.*, 9:133–166, 2000.

[91] Christoph Sprenger and Mads Dam. On global induction mechanisms in a μ-calculus with explicit approximations. *RAIRO - Theoretical Informatics and Applications - Informatique Thorique et Applications*, 37(4):365–391, 2003.

[92] Christoph Sprenger and Mads Dam. On the structure of inductive reasoning: circular and tree-shaped proofs in the μ-calculus. In *Proceedings of FOSSACS 2003*, volume 2620 of *LNCS*, pages 425–440. Springer-Verlag, 2003.

[93] Christopher Sprenger. *Deductive Local Model Checking - On the Verification of CTL* Properties of Infinite-State Reactive Systems*. PhD thesis, Swiss Federal Institute of Technology, 2000.

[94] Viktor Vafeiadis. Concurrent separation logic and operational semantics. *Electronic Notes in Theoretical Computer Science*, 276:335 – 351, 2011.

[95] Antti Valmari. A stubborn attack on state explosion. *Form. Methods Syst. Des.*, 1:297–322, 1992.

[96] Moshe Y. Vardi. Verification of concurrent programs: the automata-theoretic framework. *Annals of Pure and Applied Logic*, 51(1):79–98, 1991.

[97] Willem Visser and Howard Barringer. Practical CTL* model checking: Should spin be extended? *International Journal on Software Tools for Technology Transfer*, 2(4):350–365, 2000.

[98] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engg.*, 10:203–232, 2003.

[99] Yichen Xie and Alex Aiken. Scalable error detection using boolean satisfiability. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 351–363. ACM, 2005.

[100] Eran Yahav, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Verifying temporal heap properties specified via evolution logic. In *In ESOP2003: European Symp. on Programming, volume 2618 of LNCS*, pages 204–222. Springer, 2003.

[101] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter O'Hearn. Scalable shape analysis for systems code. In *Proceedings of the 20th International Conference on Computer Aided Verification*, CAV '08, pages 385–398. Springer-Verlag, 2008.