Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

Automatically Verifying Temporal Properties of Pointer Programs with Cyclic Proof

Gadi Tellez · James Brotherston

Received: date / Accepted: date

Abstract In this article, we investigate the automated verification of temporal properties of heap-aware programs. We propose a deductive reasoning approach based on *cyclic proof*. Judgements in our proof system assert that a program has a certain temporal property over memory state assertions, written in *separation logic* with user-defined inductive predicates, while the proof rules of the system unfold temporal modalities and predicate definitions as well as symbolically executing programs. Cyclic proofs in our system are, as usual, finite proof graphs subject to a natural, decidable soundness condition, encoding a form of proof by infinite descent.

We present a proof system tailored to proving CTL properties of nondeterministic pointer programs, and then adapt this system to handle *fair* execution conditions. We show both systems to be sound, and provide an implementation of each in the Cyclist theorem prover, yielding an automated tool that is capable of automatically discovering proofs of (fair) temporal properties of pointer programs. Experimental evaluation of our tool indicates that our approach is viable, and offers an interesting alternative to traditional model checking techniques.

Keywords Cyclic proof · Temporal logic · Separation logic

1 Introduction

Program verification can be described as the problem of deciding whether a given program exhibits a desired behaviour, often called its *specification*. Temporal logic, in its various flavours [24] is a very popular and widely studied specification formalism due to its relative simplicity and expressive power: a wide variety of *safety*

G. Tellez
Dept. of Computer Science, University College London
E-mail: gadi.tellez.13@ucl.ac.uk

J. Brotherston
Dept. of Computer Science, University College London
E-mail: J.Brotherston@ucl.ac.uk

(“something bad cannot happen”) and *liveness* properties (“something good eventually happens”) can be captured [20].

Historically, perhaps the most popular approach to automatically verify temporal properties of programs has been *model checking*: one first builds an abstract model that overapproximates all possible executions of the program, and then checks that the desired temporal property holds for this model (see e.g. [15,13, 10]). However, this approach has been applied mainly to integer programs; the situation for memory-aware programs over heap data structures becomes significantly more challenging, mainly because of the difficulties in constructing suitable abstract models. One possible approach is simply to translate such heap-aware programs into integer programs, in such a way that properties such as memory safety or termination of the original program follow from corresponding properties in its integer translation [22,15,13]. 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 given temporal property; in particular, numerical abstraction of heap programs often removes important information about the exact shape of heap data structures, which might be needed to prove some temporal properties.

Example 1 *Consider a “server” program that, given an acyclic linked list with head pointer* `x`*, nondeterministically alternates between adding an arbitrary number of “job requests” to the head of the list and removing all requests in the list:*

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

Memory safety of this program can be proven using a simple numeric abstraction recording emptiness/nonemptiness of the list. Proving instead that throughout the program execution it is always possible for the heap to become empty, expressed in CTL as $AGEF(\mathsf{emp})$, requires a finer abstraction, recording the length of the list. However, such an abstraction is still not sufficient to prove the property that the heap is always a nil-terminating acyclic list from x, expressed in CTL as $AG(ls(x, nil))$ (where ls is the standard list segment predicate of separation logic *[26]), because the information about acyclicity and head/tail pointer values is lost.*

Thus, although it is often possible to provide numeric abstractions to suit *specific* programs and temporal properties, it is not clear that this is so for *arbitrary* programs and properties.

In this article, we instead approach the above problem via the main (perhaps less fashionable) alternative to model checking, namely the *direct deductive verification.* Common limitations of previous temporal logic proof systems include restriction to finite state transition systems [18,19,4] and/or a reliance on complex verification conditions that are seemingly difficult to fully automate [23,29], the latter being arguably the most cited argument against deductive verification. In contrast, our proof system can handle infinite state systems, and an automated implementation is directly derivable from the proof rules and automata-based soundness checks, showing that temporal logic proof systems can indeed work in practice.

Our proof system manipulates temporal judgements about programs, and employs automatic proof search in this system to verify that a program has a given temporal property. Given some fixed program, the judgements of our system assert a temporal property of the program whose initial state satisfies some precondition, written in a fragment of *separation logic* [26], a well-known language for describing heap memory; in particular we also permit user-defined (inductive) predicates. The core of the proof system is a set of *logical rules* that operate on common logical connectives and unfold temporal modalities and predicate definitions, along with a set of *symbolic execution* rules that simulate program execution steps. To handle the fact that symbolic execution can in general be applied *ad infinitum*, we employ *cyclic proof* [30, 6, 7, 9], in which proofs are finite cyclic graphs subject to a global soundness condition. Using this approach, we are frequently able to verify temporal properties of heap programs in an automatic and sound way without the need of abstractions or program translations. Moreover, our analysis has the added benefit of producing independently checkable proof objects.

Our proof system is tailored to proving standard CTL program properties over separation logic assertions; subsequently, we show how to adapt this system to handle *fairness* constraints, where nondeterministic branching may not unfairly favour one branch over another. We have also adapted our system to (fair) LTL properties, though we do not present this adaptation in this paper due to space constraints. The details of our LTL cyclic proof systems will appear in the first author's forthcoming PhD thesis.

We provide an implementation of our proof system as an automated verification tool within the Cyclist theorem proving framework [9], and evaluate its performance on a range of examples. The source code, benchmark and executable binaries of the implementation are publicly available online [1]. Our tool is able to discover surprisingly complex cyclic proofs of temporal program properties, with times often in the millisecond range. Practically speaking, the advantages and disadvantages of our approach are entirely typical of deductive verification: on the one hand, we do not need to employ abstraction or program translation, and we guarantee soundness; on the other hand, our algorithms might fail to terminate, and (at least currently) we do not provide counterexamples in case of failure. Thus we believe our approach should be understood as a useful complement to, rather than a replacement for, model checking.

The remainder of this paper is structured as follows. Section 2 introduces our programming language, the memory state assertion language, and temporal (CTL) assertions over these. Section 3 introduces our proof system for verifying temporal properties of programs, and Section 4 modifies this system to account for fair program executions. Section 5 presents our implementation and experimental evaluation, Section 6 discusses related work and Section 7 concludes.

This is an expanded journal version of our previous conference paper [31]. In the present paper we have endeavoured to include as much of the technical detail of our development (particularly our soundness proofs) as space permits.

2 Programs and assertions

In this section we introduce our programming language, our language of assertions about *memory states* (based on a fragment of separation logic) and our language for expressing *temporal properties* of programs, given by CTL over memory assertions.

Programming language. We use a simple language of `while` programs with pointers and (de)allocation, but without procedures. We assume a countably infinite set Var of *variables* and a first-order language of *expressions* over Var. *Branching conditions* B and *commands* C are given by the following grammar:

$$\begin{aligned} B &::= E = E \mid E \neq E \mid * \\ C &::= \mathrm{x} := [E] \mid [E] := E \mid \mathrm{x} := \mathbf{alloc}() \mid \mathbf{free}(E) \mid \mathrm{x} := E \mid \\ &\quad\ \mathbf{skip} \mid \mathbf{if}\ B\ \mathbf{then}\ C\ \mathbf{else}\ C\ \mathbf{fi} \mid \mathbf{while}\ B\ \mathbf{do}\ C\ \mathbf{od} \mid C; C \mid \epsilon \end{aligned}$$

where $x \in \mathsf{Var}$ and E ranges over expressions. We write ϵ for the empty command, $*$ for a nondeterministic condition, and $[E]$ for dereferencing of expression E.

We define the semantics of the language in a *stack-and-heap model* employing heaps of records. We fix a set Val of *values*, and a set $\mathsf{Loc} \subset \mathsf{Val}$ of addressable memory *locations*. A *stack* is a map $s : \mathsf{Var} \to \mathsf{Val}$ from variables to values. The semantics $[\![E]\!]s$ of expression E under stack s is standard; in particular, $[\![x]\!]s = s(x)$ for $x \in \mathsf{Var}$. We extend stacks pointwise to act on tuples of terms. A *heap* is a partial, finite-domain function $h : \mathsf{Loc} \rightharpoonup_{\text{fin}} (\mathsf{Val\ List})$, mapping finitely many memory locations to *records*, i.e. arbitrary-length tuples of values; we write $\mathrm{dom}(h)$ for the set of locations on which h is defined. We write e for the *empty* heap, 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 stack or heap defined as f except that $f[x \mapsto v](x) = v$. A paired stack and heap, (s, h), is called a *(memory) state*.

A *(program) configuration* γ is a triple $\langle C, s, h\rangle$ where C is a command, 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 $\gamma_C = \epsilon$.

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, which results when a command tries to access non-allocated memory. The operational semantics of our programming language are shown in Figure 1.

An *execution path* is a (maximally 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 π is a path, then we write π_i for the ith element of π. A path π *starts from* configuration γ if $\pi_0 = \gamma$.

Remark 1 In temporal program verification, it is relatively common to consider all program execution paths to be infinite, and all temporal operators 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. $\langle \epsilon, s, h\rangle \rightsquigarrow \langle \epsilon, s, h\rangle$).

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

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

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

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

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

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

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

Fig. 1 Small-step operational semantics of programs, given by the binary relation $\rightsquigarrow$ over program configurations.

Here, instead, our temporal assertions quantify over paths that are either infinite or else maximally finite. This has the same effect as directly modifying programs or their operational semantics, but seems to us slightly cleaner.

Memory state assertions. We express properties of memory states (s, h) using a standard *symbolic-heap* fragment of separation logic (cf. [2]) extended with user-defined (inductive) predicates, typically used to express data structures in the memory.

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. 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}.

Definition 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* of the form $E = E$ or $E \neq E$, and Σ is a *spatial formula* given by the following grammar:

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

where E ranges over expressions, $\mathbf{E}$ over tuples of expressions and Ψ over predicate symbols (of arity matching the length of $\mathbf{E}$ in $\Psi(\mathbf{E})$).

Definition 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

$$\begin{array}{rcl}
s,h \models E_1 = E_2 & \Leftrightarrow & [\![E_1]\!]s = [\![E_2]\!]s \\
s,h \models E_1 \neq E_2 & \Leftrightarrow & [\![E_1]\!]s \neq [\![E_2]\!]s \\
s,h \models \mathsf{emp} & \Leftrightarrow & \mathrm{dom}(h) = \emptyset \\
s,h \models E \mapsto \mathbf{E} & \Leftrightarrow & \mathrm{dom}(h) = \{[\![E]\!]s\} \text{ and } h([\![E]\!]s) = [\![\mathbf{E}]\!]s \\
s,h \models \Psi(\mathbf{E}) & \Leftrightarrow & ([\![\mathbf{E}]\!]s, h) \in [\![\Psi]\!] \\
s,h \models \Sigma_1 * \Sigma_2 & \Leftrightarrow & h = h_1 \uplus h_2 \text{ and } s,h_1 \models \Sigma_1 \text{ and } s,h_2 \models \Sigma_2 \\
s,h \models \Omega_1 \vee \Omega_2 & \Leftrightarrow & s,h \models \Omega_1 \text{ or } s,h \models \Omega_2
\end{array}$$

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 [6] and is given by the following definition:

Definition 3 (Inductive definition) An *inductive definition* of an inductive predicate symbol Ψ is a finite set of inductive rules, each of the form $\Pi\colon \Sigma \Rightarrow \Psi(\mathbf{E})$, where $\Pi\colon \Sigma$ is a symbolic heap 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 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 $\chi_i : (\mathrm{Pow}(\mathsf{Heaps} \times \mathsf{Val\ List}^{\mathsf{a}_1}) \times \mathrm{Pow}(\mathsf{Heaps} \times \mathsf{Val\ List}^{\mathsf{a}_n})) \to \mathrm{Pow}(\mathsf{Heaps} \times \mathsf{Val\ List}^{\mathsf{a}_i})$ as follows:

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

where s is an arbitrary stack and $\models_{[\![\Psi]\!] \mapsto \mathbf{X}}$ is the satisfaction relation defined exactly as Definition 2 except that $[\![\Psi_i]\!] = \pi_i^n(\mathbf{X})$ for each $i \in \{1, \ldots, n\}$ (where π_i^n is the ith projection on n-tuples of sets defined by $\pi_i^n(X_1, \ldots, X_n) = X_i$). Then the *definition set operator* for $\Psi_i, \ldots, \Psi_n$ is the operator χ_Ψ defined by:

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

Example 2 *Consider the following inductive definition for a binary inductive predicate symbol* ls *denoting singly-linked list segments:*

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

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

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

Note that the operator generated from a set of inductive definitions by Definition 4 is monotone [6] 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 5 (Approximants) Let χ_Ψ be the definition set operator for the inductive predicates $\Psi_1, \ldots, \Psi_n$ as in Definition 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 $i \in 1, \ldots, n$, the set $\Psi_i^\alpha = \pi_i^n(\chi_\Psi^\alpha)$ is called the α-th *approximant* of Ψ_i.

Temporal assertions. We describe temporal properties of our programs using *temporal assertions*, built from the memory state assertions given above using standard operators of *computation tree logic* (CTL) [11], where the temporal operators quantify over execution paths from a given configuration.

Definition 6 *CTL assertions* are described by the grammar:

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

where P ranges over memory state assertions (Defn. 1).

Note that final and error denote final, respectively faulting configurations.

Definition 7 A configuration γ is a *model* of the CTL assertion φ if the relation $\gamma \models \varphi$ holds, defined by structural induction on φas follows:

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

Judgements in our system are given by $P \vdash C : \varphi$, where P is a symbolic heap, C is a command sequence and φ is a CTL assertion.

Definition 8 (Validity) 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 $\langle C, s, h\rangle \models \varphi$.

3 A cyclic proof system for verifying CTL properties

In this section, we present a cyclic proof system for establishing the validity of our CTL judgements on programs, as described in the previous section.

Our proof rules for CTL judgements are shown in Figure 2. The *symbolic execution* rules for commands are adapted from those in the proof system for program termination in [7], 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 our programs, namely: (i) nondeterministic `while`; (ii) nondeterministic `if`; and (iii) memory allocation. It is only for these constructs that we need a specific symbolic execution rule for $\Box$ and $\Diamond$ properties. 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 also introduce *faulting execution rules* to allow us to prove that a program faults. The logical rules comprise standard rules for the logical connectives and standard unfolding rules for the temporal operators and inductive predicates in memory assertions. In particular, 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, the inductive definition of the linked list segment predicate from Example 2 determines the following (Unfold-Pre) rule:

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

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. [6,7,9,8,27], formalise this notion.

Definition 9 (Pre-proof) A *leaf* of a derivation tree is called *open* if it is not the conclusion of an axiom. 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 $\mathcal{L}$ is a *back-link function* assigning to every open leaf of $\mathcal{D}$ a *companion*: an interior node of $\mathcal{D}$ labelled by an identical proof judgement.

A pre-proof $\mathcal{P} = (\mathcal{D}, \mathcal{L})$ can be seen as a finite cyclic graph by identifying each open leaf of $\mathcal{D}$ with its companion. A *path in* $\mathcal{P}$ is then a path in this graph. It is easy to see that a path in a pre-proof corresponds to one or more paths in the execution of a program, interleaved with logical inferences.

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 10 (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 along the path, $(\varphi_i)_{i\geq 0}$, is a $\Box$*-trace* ($\Diamond$*-trace*) following that path if there exists a formula ψ such that, for all $i \geq 0$:

- the formula φ_i is of the form $AG\psi$ ($EG\psi$) or $\Box AG\psi$ ($\Diamond EG\psi$); and
- $\varphi_{i+1} = \varphi_i$ whenever J_i is the conclusion of (Cons), (Split) or (Unfold-Pre), and respects any substitutions performed when J_i is the conclusion of (Subst).

Symbolic execution rules:

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

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

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

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

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

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

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

$$\frac{P, B \vdash (C_1\ ;\ \mathtt{while}\ B\ \mathtt{do}\ C_1\ \mathtt{od}\ ;\ C_2) \,:\, \varphi \quad P, \neg B \vdash C_2 \,:\, \varphi}{P \vdash (\mathtt{while}\ B\ \mathtt{do}\ C_1\ \mathtt{od}\ ;\ C_2) \,:\, \bigcirc\varphi}\ (\text{Wh}) \qquad \frac{}{P \vdash \epsilon \,:\, \mathsf{final}}\ (\text{Final})$$

Faulting execution rules:

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

Logical rules:

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

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

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

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

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

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

Fig. 2 Proof rules for CTL judgements. We write $\bigcirc\varphi$ to mean "either $\Box\varphi$ or $\Diamond\varphi$".

We say that a trace *progresses* whenever a symbolic execution rule is applied. A 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, as employed in [7]. Roughly speaking, a precondition trace tracks an occurrence of an inductive predicate in the preconditions of the judgements along the path, progressing whenever the predicate occurrence is unfolded.

Definition 11 (Precondition trace) Let $(J_i = P_i \vdash C_i : \varphi_i)_{i\geq 0}$ be a path in a pre-proof $\mathcal{P}$. A sequence of inductive predicate formulas, $(\Psi_i)_{i\geq 0}$, is a *precondition trace* following that path $(J_i)_{i\geq 0}$ if Ψ_i occurs in P_i for each i and:

(i) whenever J_i is the conclusion of the (Unfold-Pre) rule, the predicate formula Ψ_i is the predicate in the spatial formula of P_i being unfolded and Ψ_{i+1} is obtained in the premise J_{i+1} by unfolding Ψ_i; 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.

Definition 12 (Cyclic proof) A pre-proof $\mathcal{P}$ is a *cyclic 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 $\Box$-trace, $\Diamond$-trace or precondition trace following some tail $(P_i \vdash C_i : \varphi_i)_{i\geq n}$ of the path.

While admittedly not the most interesting program in itself, 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.

Example 3 *Consider the following program:*

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

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 shows the proof of this property in our system, including the 6 cycles that are formed during the proof search, with traces that follow the infinite paths. For the rightmost cycle in the lower branch of the figure we note that the temporal components of the sequents in the infinite proof path are all 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 10. 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 leftmost cycle in the top branch of the figure, we note that the temporal components 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. 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 remaining 4 back-links shown in the proof do not match their corresponding leaf node to a direct ancestor. Nevertheless, these infinite paths are, too, followed by $\Box$-traces that progress infinitely often. Consequently, our pre-proof qualifies as a valid cyclic proof.

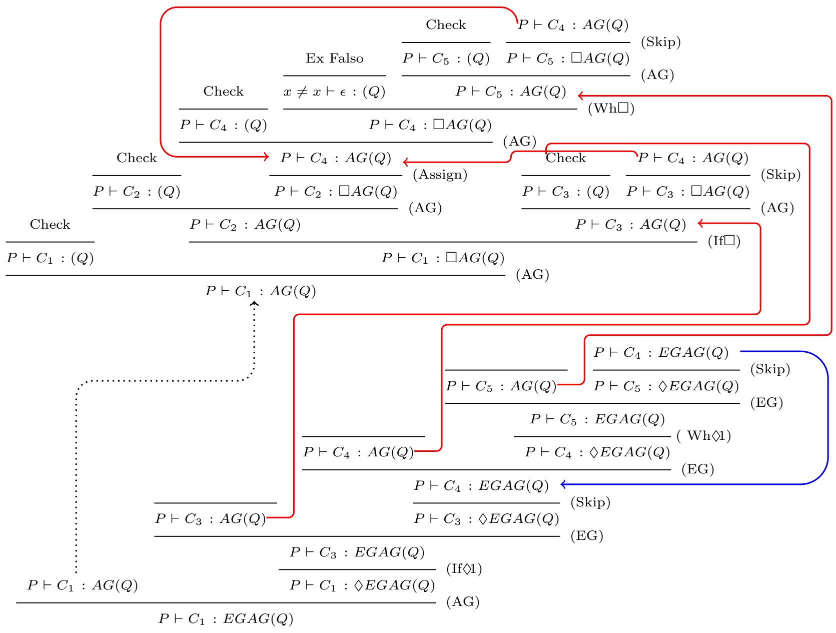


Fig. 3 Nested temporal operators example

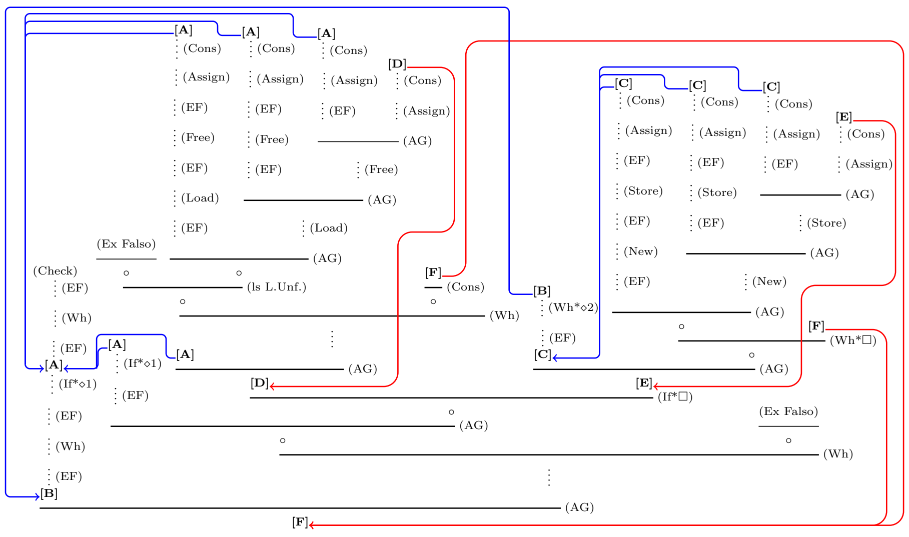
For a more realistic example, we now present a proof of a heap-aware server program that nondeterministically alternates between adding an arbitrary number of "job requests" to the head of a linked-list and processing job requests by means of deleting them from the list.

Example 4 *Consider the server-like program in Example 1 in the Introduction. We can show that, given that the heap is initially a linked list from* `x` *to* 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 4 shows an outline cyclic proof of this judgement in our system (we suppress the internal judgements for space reasons, but show the cycle structure and rule applications). Note that the back-links depicted in blue do not form infinite loops as they all point to a companion that eventually leads to a (Check) axiom. The red back-links do give rise to infinite paths; one can see that the pre-proof qualifies as a valid cyclic proof since there is an infinitely progressing $\Box$-trace along every infinite path.

Proposition 1 (Decidable soundness condition) *It is decidable whether a pre-proof is a cyclic proof.*

Proof (Sketch) Given a pre-proof $\mathcal{P}$, we construct two Büchi automata over strings of nodes of $\mathcal{P}$. The first automaton B_1 accepts all infinite paths of $\mathcal{P}$. The second automaton B_2 accepts all infinite paths of $\mathcal{P}$ such that an infinitely progressing $\Box$-, $\Diamond$- or precondition-trace exists on some tail of the path. We then simply check $\mathcal{L}(B_1) \subseteq \mathcal{L}(B_2)$, which is decidable.



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

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

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

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

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

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

Fig. 4 Single threaded monolithic server example

We now show that our proof system is sound.

Lemma 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 subformula of φ given by Defn. 10, there is a configuration γ such that $\gamma \not\models \psi$, and the finite execution 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 subformula of φ given by Defn. 10, 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.*
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. $(s, h_0) \models \Psi^{\alpha}(\mathbf{E})$ and $(s', h'_0) \models \Psi'^{\beta}(\mathbf{E'})$ for subheaps h_0 of h and h'_0 of h'), then α, β are well defined and $\beta \leq \alpha$. Moreover $\beta < \alpha$ when R is the (Unfold-Pre) rule.*

Proof We distinguish a case for each proof rule. Due to space constraints, we show only a few interesting cases: the symbolic execution rules (If*$\Diamond$1) and (If*$\Box$), which provide progress for $\Diamond$- and $\Box$-traces respectively, and the unfolding rule (Unfold-Pre), which provides progress for precondition traces.

Soundness of If$\Diamond$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 such that $(s,h) \models P$ but $\gamma = \langle \textbf{if} * \textbf{then}\ C_1\ \textbf{else}\ C_2\ \textbf{fi}\ ; C_3, s, h\rangle \not\models \Diamond\varphi$. By Defn. 7, if $\gamma \not\models \Diamond\varphi$ then for all γ' such that $\gamma \rightsquigarrow \gamma', \gamma' \not\models \varphi$. By the operational semantics of our programming language we know that there are two possible transitions: $\gamma \rightsquigarrow \langle C_1 \ ; C_3, s, h\rangle$ and $\gamma \rightsquigarrow \langle C_2 \ ; C_3, s, h\rangle$. Hence, $\langle C_1 \ ; C_3, s, h\rangle \not\models \varphi$ and $\langle C_2 \ ; C_3, s, h\rangle \not\models \varphi$. Consequently, since by our assumption $(s,h) \models P$ but $\langle C_1 \ ; C_3, s, h\rangle \not\models \varphi$ then 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 Defn. 10, $\varphi = EG\psi$. Since $\gamma \not\models EG\psi$, by Defn. 7 we know that for all paths π' starting from γ there exists $\gamma_\neg \in \pi$ such that $\gamma_\neg \not\models \psi$. In other words, there is a finite execution tree κ with root γ and whose leaves all fail to satisfy ψ. Moreover, since $\gamma \rightsquigarrow \langle C_1 \ ; C_3, s, h\rangle$ by the operational semantics, there is a maximal subtree κ' of κ with root $\langle C_1 \ ; C_3, s, h\rangle$ and whose leaves also all fail to satisfy ψ. Clearly $\text{height}(\kappa') \leq \text{height}(\kappa)$.

Finally, if there is a precondition trace $(\Psi(\mathbf{E}), \Psi'(\mathbf{E}))$ following the edge then $\Psi(\mathbf{E}) = \Psi'(\mathbf{E'})$ by Defn. 11. Since $(s,h) \models P$ and $\Psi(\mathbf{E})$ is a predicate formula in P, we have $(s,h) \models \Psi(\mathbf{E})$ and therefore a smallest approximant α such that $(s,h) \models \Psi^\alpha(\mathbf{E})$. Since here our (s',h') is just (s,h) and our $\Psi'(\mathbf{E'})$ is just $\Psi(\mathbf{E})$, we trivially have the required $\beta = \alpha \leq \alpha$.

Soundness of If$\Box$*

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

Assume the conclusion of the rule is invalid. Pick an arbitrary program state such that $(s,h) \models P$ but $\gamma = \langle \textbf{if} * \textbf{then}\ C_1\ \textbf{else}\ C_2\ \textbf{fi}\ ; C_3, s, h\rangle \not\models \Box\varphi$. By Defn. 7, if $\gamma \not\models \Box\varphi$ then there exists γ' such that $\gamma \rightsquigarrow \gamma'$ and $\gamma' \not\models \varphi$. By the operational semantics we know that there are two possibilities: $\gamma' = \langle C_1 \ ; C_3, s, h\rangle$ or $\gamma' = \langle C_2 \ ; C_3, s, h\rangle$. Hence, either $\langle C_1 \ ; C_3, s, h\rangle \not\models \varphi$ or $\langle C_2 \ ; C_3, s, h\rangle \not\models \varphi$. We show the first of these cases, the second being similar.

Case $\gamma' = \langle C_1 \ ; C_3, s, h\rangle$.

By our assumption $(s,h) \models P$ and invalidity result $\gamma' \not\models \varphi$ then it is the case that the left-most premise is invalid.

If there is a $\Box$-trace following the edge, then by Defn. 10, $\varphi = AG\psi$. Furthermore, since $\gamma' \not\models AG\psi$, then by Defn. 7 we know that there exists $\gamma_\neg$ such that $\gamma' \rightsquigarrow^* \gamma_\neg$ and $\gamma_\neg \not\models \psi$ (call this path π). Finally, since by the operational semantics $\gamma \rightsquigarrow \gamma'$, then there is a subpath π' of π from γ' to $\gamma\neg$, and moreover $\text{length}(\pi') < \text{length}(\pi)$.

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

The situation for precondition traces is similar to the previous case.

Soundness of Unfold-Pre

$$\frac{(\Pi \cup \Pi'_i\colon \Sigma * \Sigma'_i \vdash C\,:\,\varphi)_{1\leq i\leq k}}{\Pi\colon \Psi(\mathbf{E}) * \Sigma \vdash C\,:\,\varphi}\ \text{(Unfold-Pre)}$$

Assume the conclusion of the rule is invalid. Pick an arbitrary program state such that $(s,h) \models \Pi\,:\,\Psi(\mathbf{E}) * \Sigma$ but $\langle C,s,h\rangle \not\models \varphi$. By Defn. 2 we can split h into two disjoint subheaps $h = h' \uplus h''$ so that $(s,h') \models \Pi\,:\,\Psi(\mathbf{E})$ and $(s,h'') \models \Pi\,:\,\Sigma$. Since $(s,h') \models \Pi\,:\,\Psi(\mathbf{E})$ then by Defn. 2 we know that $(\llbracket\mathbf{E}\rrbracket s,h') \in \llbracket\Psi\rrbracket$. Moreover, by Defn. 5 we know that the program state (s,h') is in the αth approximant of Ψ for some smallest α (i.e. $(\llbracket\mathbf{E}\rrbracket s,h') \in \llbracket\Psi\rrbracket^{\alpha}$). By construction of the definition set operator for Ψ (Defn. 4), it follows that there is some inductive rule $\Pi_j\,:\,\Sigma_j \Rightarrow \Psi(\mathbf{E})$) such that $(s,h') \models \Pi_j\,:\,\Sigma_j$ (we ignore variable renaming issues for simplicity). We choose J_{i+1} to be the jth premise of the rule, $\Pi\cup\Pi_j\colon \Sigma*\Sigma_j \vdash C\,:\,\varphi$, and our falsifying state to be (s,h). It follows that $(s,h) \models \Pi\cup\Pi_j\colon \Sigma * \Sigma_j$, and so the premise J_{i+1} is not valid, as required.

If there is a $\Box$-trace following the edge (J_i,J_{i+1}), then by Defn. 10, $\varphi = AG\psi$ or $\varphi = \Box AG\psi$. Furthermore, since by our previous invalidity result $\langle C,s,h\rangle \not\models \varphi$, then by Defn. 7 we know that there exists $\gamma_\neg$ such that $\langle C,s,h\rangle \leadsto^* \gamma_\neg$ and $\gamma_\neg \not\models \psi$ (call this path π). Thus the paths $\pi' = \pi$ in item 1 of the lemma are well defined and, trivially, $\text{length}(\pi') \leq \text{length}(\pi)$.

If there is a precondition trace $(\Psi(\mathbf{E}),\Psi(\mathbf{E'}))$ following the edge then $\Psi'(\mathbf{E'})$ is an unfolding of $\Psi(\mathbf{E})$ appearing in Σ_j. Since $(\llbracket\mathbf{E}\rrbracket s,h') \in \llbracket\Psi\rrbracket^{\alpha}$, it follows that $(\llbracket\mathbf{E'}\rrbracket s,h_0) \in \llbracket\Psi'\rrbracket^{\beta}$ for some subheap h_0 of h' and some approximant $\beta < \alpha$. This completes the case.

Theorem 1 (Soundness) *If $P \vdash C\colon \varphi$ is provable, then it is valid.*

Proof Suppose for contradiction that there is a cyclic proof $\mathcal{P}$ of $J = P \vdash C\colon \varphi$ 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 (Lemma 1), we can construct an infinite path $(P_i \vdash C_i\colon \varphi_i)_{i\geq 0}$ in $\mathcal{P}$ of invalid judgements. Since $\mathcal{P}$ is a cyclic proof, by Defn. 12 there exists an infinitely progressing trace following some tail $(P_i \vdash C_i\colon \varphi_i)_{i\geq n}$ of this path.

If this trace is a $\Box$-trace, using condition 1 of Lemma 1, we can construct an infinite sequence of finite paths to a *fixed* configuration γ of infinitely decreasing length, contradiction. If the trace is a $\Diamond$ trace, we can construct an infinite sequence of execution trees (whose leaves are again fixed configurations) of infinitely decreasing height, contradiction. Finally, a precondition trace yields an infinitely decreasing sequence of ordinal approximations of some inductive predicate, also a contradiction. The conclusion is that $P \vdash C\colon \varphi$ must be valid after all.

We remark that the dichotomy between inductive and coinductive arguments can be discerned in our trace condition. Coinductive (G) properties need to show that something happens infinitely often whereas inductive (F) properties have to show that something *cannot* happen infinitely often. Both cases give us a progress condition: for coinductive properties, we essentially need infinite program progress on the right of judgements, whereas for inductive properties we need an infinite descent on the left of judgements (or for the proof to be finite).

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 [7], 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 [7] only treats termination, whereas we treat arbitrary temporal properties. Since our focus in this paper is on automation, we leave such an analysis to future work.

4 Fairness

An important component in the verification of reactive systems is a set of *fairness requirements* to guarantee that no computation is neglected forever. In this section, we show how our cyclic proof system for CTL program properties can be adapted to incorporate such fairness constraints when verifying nondeterministic programs.

Definition 13 (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 $\pi_i = \langle C, \_, \_ \rangle$. A program execution π is *fair for commands 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 pairs of commands 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 every finite program execution is trivially fair. Also, for the purposes of fairness, we consider program commands to be uniquely labelled (to avoid confusion between different instances of the same command).

We now modify our cyclic CTL system to treat fairness constraints. First, we adjust the interpretation of judgements to account for fairness, then we lift the definition of fairness from program executions to paths in a pre-proof.

Definition 14 (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 $s, h \models P$, we have $\langle C, s, h \rangle \models_f \varphi$, where $\models_f$ is the satisfaction relation obtained from $\models$ in Defn. 7 by interpreting the temporal operators as quantifying over *fair* paths, rather than all paths. For example, the clause for AG becomes

$$\gamma \models_f AG\varphi \Leftrightarrow \forall\ \textit{fair}\ \pi \text{ starting from } \gamma.\ \forall \gamma' \in \pi.\ \gamma' \models_f \varphi.$$

Definition 15 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 $i \geq 1$ such that $J_{i_C} = C$ and the rule applied at J_{i-1} is a symbolic execution rule. A path in a pre-proof is *fair for* *commands* C_i, C_j if it is the case that $(J_i)_{i\geq 0}$ visits C_i infinitely often if and only if it visits C_j infinitely often. Finally, the path is *fair for program* C iff it is fair for all pairs of commands 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 from Defn. 12 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) do { if (*) then x := 1 else x := 2 } od.
```

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 16 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$1) or (If*$\Diamond$1) is applied to infinitely many distinct J_i such that $J_{i_C} = C$ and (Wh*$\Diamond$2) or (If*$\Diamond$2) is applied to only finitely many distinct J_i such that $J_{i_C} = C$ or vice versa.

Definition 17 (Fair cyclic proof) A pre-proof $\mathcal{P}$ is a *fair cyclic proof* if it is not *bad* in the sense of Defn. 16 above and, 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.

Proposition 2 (Decidable soundness condition) *It is decidable whether a pre-proof is a valid fair cyclic proof.*

Proof (Sketch) Given a pre-proof $\mathcal{P}$, we first have to check that it is not *bad* in the sense of Defn. 16. This can be done using Büchi automata: given a particular program point C, one can construct automata B_1 and B_2 accepting the infinite paths in $\mathcal{P}$ such that ((Wh*$\Diamond$1)) or (If*$\Diamond$1), respectively (Wh*$\Diamond$2) or (If*$\Diamond$2) are applied infinitely often to C. We simply check $\mathcal{L}(\mathcal{A}_{B_1}) \subseteq \mathcal{L}(\mathcal{A}_{B_2})$ and $\mathcal{L}(\mathcal{A}_{B_2}) \subseteq \mathcal{L}(\mathcal{A}_{B_1})$, which is decidable (repeating this exercise for each needed program point).

It just remains to check that there exists an infinitely progressing trace along every infinite fair path of $\mathcal{P}$. The argument is similar to the argument for our non-fair cyclic proofs, except that instead of an automaton accepting all infinite paths of $\mathcal{P}$ we now require one accepting only *fair* infinite paths. This can be done using a *Streett automaton* with an acceptance condition of 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). We are then done since Streett automata can be transformed into Büchi automata [21].

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

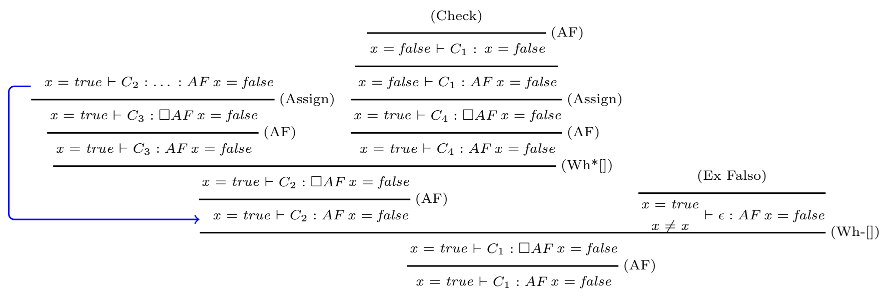

(Check)
(AF)
x = false ⊢ C_1 : x = false
x = true ⊢ C_2 : . . . : AF x = false
x = false ⊢ C_1 : AF x = false
(Assign)
(Assign)
x = true ⊢ C_3 : □AF x = false
x = true ⊢ C_4 : □AF x = false
(AF)
(AF)
x = true ⊢ C_3 : AF x = false
x = true ⊢ C_4 : AF x = false
(Wh*[])
(Ex Falso)
x = true ⊢ C_2 : □AF x = false
(AF)
x = true
x ≠ x ⊢ ϵ : AF x = false
x = true ⊢ C_2 : AF x = false
(Wh-[])
x = true ⊢ C_1 : □AF x = false
(AF)
x = true ⊢ C_1 : AF x = false

Fig. 5 Fair CTL cyclic proof example

Example 5 *Consider the following labelled program C:*

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

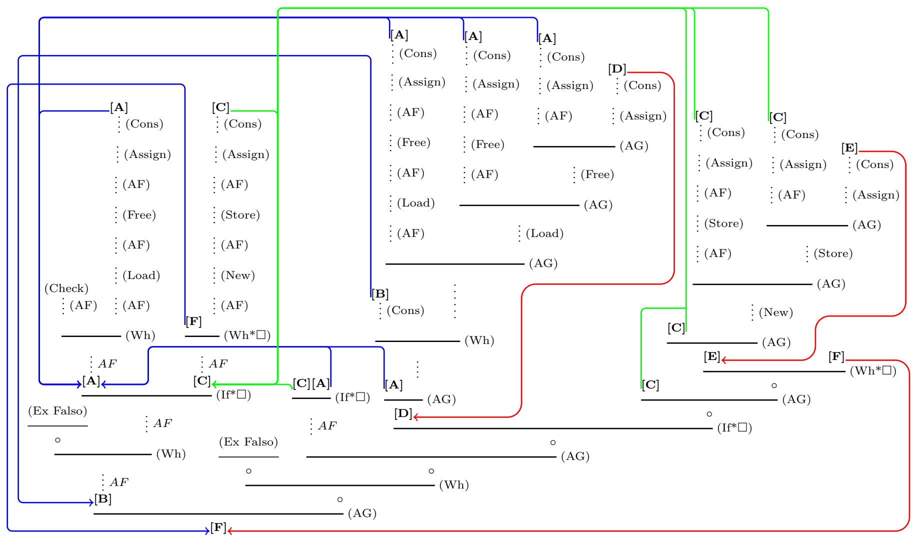
While the property $AF(x = false)$ fails for this program in general, it becomes true under the fairness constraint (C_3, C_4). Figure 5 shows an abridged proof of this property in our fair cyclic CTL proof system, where the application of a $(\vee)$ rule following each (AF) rule has been omitted from the proof for brevity. Note the formation of a cycle on the leftmost branch, along which there is no trace. Whereas this cycle invalidates the general CTL soundness condition, it does not invalidate the fair *CTL soundness condition, since the path in question visits C_3 infinitely often but not C_4; hence it is* unfair *as per Defn. 15 (but not* bad *in the sense of Defn. 16). Hence, the fair 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.*

Example 6 *We return to our server program from Examples 1 and 4. Suppose we wish to prove, not that it is always* possible *for the heap to become empty, i.e. $AGEF(\mathsf{emp})$, but that the heap will* always eventually *become empty, i.e. $AGAF(\mathsf{emp})$. While the program does* not *satisfy this property in general, it* does *satisfy the property under the assumption of fair execution, which prevents the second loop from being executed infinitely often without executing the first loop.*

Figure 6 shows an abridged proof of this property in the our fair cyclic CTL proof system. Adding the fairness constraints relaxes the conditions under which back-links can be formed. The back-links depicted in green induce infinite paths with no valid trace. However, because these infinite paths are unfair *(and not* bad*), they are not considered in the global soundness condition. Our pre-proof thus qualifies as a fair cyclic proof since along every* fair *infinite path there is either a $\Box$-trace or a precondition trace progressing infinitely often.*

Theorem 2 (Soundness) *If $P \vdash_f C : \varphi$ is provable, then it is valid.*

Proof (Sketch) Suppose for contradiction that there is a fair 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



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

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

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

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

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

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

Fig. 6 Single threaded monolithic server example

$(s, h) \models P$ but $\langle C, s, h\rangle \not\models_f \varphi$. Then, 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 have an infinitely progressing trace along this path and can thus obtain a contradiction exactly as in the proof of soundness for the standard system (Theorem 1) provided that Π is *fair*.

Suppose therefore that Π is unfair for commands (C_i, C_j) say. We consider the case in which C contains the command **if** $*$ **then** C_i **else** C_j **fi** and Π visits C_i infinitely often and C_j only finitely often; the **while** case is similar. Using Definition 15 we know that **if** $*$ **then** C_i **else** C_j **fi** itself is symbolically executed infinitely often on Π. It cannot be that (If $*\Diamond 1$) is applied infinitely often and (If$*\Diamond 2$) only finitely often on Π, since Π would in that case be a *bad* path, which is specifically excluded by Defn. 16. Nor can it be that both rules are applied infinitely often, since in that case Π would be fair for (C_i, C_j), contrary to assumption.

The only remaining possibility is that (If $*\,\Box$) is applied infinitely often on Π. In that case it must be that Π contains infinitely many occurrences of the left premise of the rule and only finitely many instances of the right premise (or vice versa). Hence the program execution underlying the pre-proof path Π is also unfair. Since the satisfaction relation $\models_f$ is restricted to fair program executions, this contradicts the assumption that $\langle C, s, h\rangle \not\models_f \varphi$. (The full justification of this last step requires the observation that, in order to produce a $\Box$ infinitely often, the underlying temporal property must contain an outermost AF or AG quantifier.)

□

5 Implementation and Evaluation

We implement our proof systems on top of the CYCLIST theorem prover [9], a mechanised cyclic theorem proving framework. The implementation, source code and benchmarks are publicly available at [1] (under the subdirectory titled as the present paper).

Our implementation performs iterative depth-first search, aimed at closing open nodes in the proof by either applying an inference rule or forming a back-link. If an open node cannot be closed, we attempt to apply symbolic execution; if this is not possible, we try unfolding temporal operators and inductive predicates in the precondition to enable symbolic execution to proceed. Forming back-links typically requires the use of the consequence rule (i.e. a lemma proven on demand) to re-establish preconditions altered by symbolic executions (as can be seen in Figures 4 and 6). When all open nodes have been closed, a global soundness check of the cyclic proof is performed automatically. Entailment queries over symbolic heaps in separation logic, which arise at backlinks and when applying the (Check) axiom or checking rule side conditions, are handled by a separate instantiation of CYCLIST for separation logic entailments [9].

We evaluate the implementation on handcrafted nondeterministic and nonterminating programs similar to Example 1. Our test suite is essentially an adaptation of the model checking benchmarks in [14,15] for temporal properties of nondeterministic programs. Roughly speaking, we replace operations on integer variables by analogous operations on heap data structures.

Our test suite comprises the following programs:

(i) Examples discussed in the paper are named EXMP;
(ii) FIN-LOCK - 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;
(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 three or performs a random walk of the three;
(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 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 taken from the aforementioned benchmarks (ACQ-REL).

Program	Precondition	Property	Fairness	Time (s)
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	l $\mapsto$0 * ls(x,nil)	AF (l $\mapsto$1 * emp)	No	0.20
Fin-Lock	l $\mapsto$0 * ls(x,nil)	AGAF(l $\mapsto$1 * emp)	No	0.62
Fin-Lock	l $\mapsto$0 * ls(x,nil)	AGAF(l $\mapsto$1 * emp $\wedge$ $\Diamond$l $\mapsto$0)	No	0.24
Inf-Lock	l $\mapsto$0 * ls(x,nil)	AGAF(l $\mapsto$1 * emp)	No	1.52
Inf-Lock	l $\mapsto$0 * ls(x,nil)	AGAF(l $\mapsto$1 * emp $\wedge$ $\Diamond$l $\mapsto$0))	No	3.26
Inf-Lock	d=f : l $\mapsto$0 * ls(x,nil)	AG(d!=t $\vee$ AF (l $\mapsto$1 * emp))	No	3.87
Nd-Inf-Lock	l $\mapsto$0	AF(l $\mapsto$1)	Yes	0.15
Nd-Inf-Lock	l $\mapsto$0	AGAF (l $\mapsto$1)	Yes	0.25
Inf-List	ls(x,nil)	AG ls(x,nil)	No	0.21
Inf-List	ls(x,nil)	AGEF x=nil	No	4.39
Inf-List	ls(x,nil)	AGAF x=nil	Yes	8.10
Insert-List	ls(three,zero)	EF ls(five,zero)	No	0.14
Insert-List	ls(three,zero)	AF ls(five,zero)	Yes	0.26
Insert-List	ls(n,zero)	AGAF n!=zero	Yes	17.21
Append-List	ls(y,x) * ls(x,nil)	AF (ls(y,nil))	No	12.67
Cyclic-List	cls(x,x)	AG cls(x,x)	No	0.88
Cyclic-List	cls(x,x)	AGEG cls(x,x)	No	0.34
Inf-BinTree	x!=nil : bintree(x)	AGEG x!=nil	No	0.72
AFAG Branch	x$\mapsto$zero	AFAG x$\mapsto$one	No	1.80
EGAG Branch	x$\mapsto$zero	EGAG x$\mapsto$one	No	0.23
EGAF Branch	x$\mapsto$zero	EGAF x$\mapsto$one	No	15.48
EG$\Rightarrow$ EF Branch	p=z $\wedge$ q=z : ls(z,n)	EG(p!=one $\vee$ EF q=one)	No	1.60
EG$\Rightarrow$ AF Branch	p=z $\wedge$ q=z : ls(z,n)	EG(p!=one $\vee$ AF q=one)	Yes	5.33
AG$\Rightarrow$ EG Branch	p=z $\wedge$ q=one : ls(z,n)	AG(p!=one $\vee$ EG q=one)	No	0.36
AG$\Rightarrow$ EF Branch	p=z $\wedge$ q=one :u ls(z,n)	AG(p!=one $\vee$ EF q=one)	No	1.53
Acq-rel	ls(zero,three)	AG(acq=0 $\vee$ AF rel!=0)	No	1.25
Acq-rel	ls(zero,three)	AG(acq=0 $\vee$ EF rel!=0)	No	1.25
Acq-rel	ls(zero,three)	EF acq!=0 $\wedge$ 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 $\wedge$ EF EG rel=0	No	0.25
Acq-rel	ls(zero,three)	AF EG rel=0	Yes	0.33
PostgreSQL	w=true $\wedge$ s=s' $\wedge$ f=f'	AGAF w=true $\wedge$ s=s' $\wedge$ f=f'	No	0.27
PostgreSQL	w=true $\wedge$ s=s' $\wedge$ f=f'	AGEF w=true $\wedge$ s=s' $\wedge$ f=f'	No	0.26
PostgreSQL	w=true $\wedge$ s=s' $\wedge$ f=f'	EFEG w=false $\wedge$ s=s' $\wedge$ f=f'	No	0.44
PostgreSQL	w=true $\wedge$ s=s' $\wedge$ f=f'	EFAG w=false $\wedge$ s=s' $\wedge$ f=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 1 Experimental results.

We show the results of the evaluation of the CTL system and its fair extension in Table 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.

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 temporal property in question, 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 $P \models Q$ 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). 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. [9]), this too is an expensive operation, as can be seen in the runtimes of test cases with suffix BRANCH.

Notwithstanding the broadly encouraging results, our implementation is not without limitations; in some cases, the proof search will fail to terminate. 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.

6 Related Work

Related work on the automated verification of temporal program properties can broadly be classified into two main schools, *model checking* and *deductive verification*. In recent years, model checking has been the more popular of these two. Although earlier work in model checking focused on finite-state transition systems (e.g. [11,25]), recent advances in areas such as state space restriction [3], precondition synthesis [13], CEGAR [15], bounded model checking [10] and automata theory [12] have enabled the treatment of infinite transition systems.

The present paper takes the deductive verification approach. In the realm of infinite state, previous proof systems for verifying temporal properties of arbitrary transition systems [23,29] have shed some light on the soundness and relative completeness of 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 [7] to arbitrary temporal properties. In [4], a procedure for the verification of CTL* properties is developed that employs a cyclic proof system for LTL as a subprocedure. 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 this restriction is a simplification of the soundness condition on cyclic proofs.

A cyclic proof system for the verification of CTL* properties of infinite-state transition systems is presented in [29]. 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. 12, 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 [30,5,9].

7 Conclusions and Future Work

Our main contribution in this paper is the formulation, implementation and evaluation of a deductive cyclic proof system for verifying temporal properties of pointer programs, building on previous systems for separation logic and for other temporal verification settings [4,7,29]. We present two variants of our system and prove both systems sound. We have implemented these proof systems, and proof search algorithms for them, in the Cyclist theorem prover, and evaluated them on benchmarks drawn from the literature.

The main advantage of our approach is that we never obtain false positive results. This advantage is not in fact exclusive to deductive verification: some automata-theoretic model checking approaches are also proven to be sound [33]. Nonetheless, when compared to such approaches, our treatment of the temporal verification problem has the advantage of being direct. Owing to our use of separation logic and a deductive proof system, we do not need to apply approximation or transformations to the program as a first step; in particular, we avoid the translation of temporal formulas into complex automata [34] and the instrumentation of the original program with auxiliary constructs [12].

One natural direction for future work is to develop improved mechanised techniques, such as generalisation / abstraction, to enhance the performance of proof search in our system(s). Another possible direction is to consider larger classes of programs. In particular, concurrency is one very interesting such possibility, perhaps building on existing verification techniques for concurrency in separation logic (e.g. [32]). A different direction to explore is the enrichment of our assertion language, for example to CTL* [17] or μ-calculus [16]. The structure of CTL* formulas and their classification into path and state subformulas suggest a possible combination of our CTL system with an LTL system to produce a proof object composed of smaller proof structures (cf. [4,29]). The encoding of CTL* into μ-calculus [16] and the applicability of cyclic proofs for the verification of μ-calculus properties (see e.g. [28]) hint at the feasibility of such an extension.

References

1. www.github.com/ngorogiannis/cyclist/releases
2. Berdine, J., Calcagno, C., O'Hearn, P.W.: A decidable fragment of separation logic. In: Proceedings of FSTTCS-24, pp. 97–109. Springer-Verlag (2004)
3. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker blast: Applications to software engineering. Int. J. Softw. Tools Technol. Transf. **9**, 505–525 (2007)

4. Bhat, G., Cleaveland, R., Grumberg, O.: Efficient on-the-fly model checking for CTL*. In: Proceedings of LICS-10, pp. 388–397. IEEE (1995)
5. Brotherston, J.: Sequent calculus proof systems for inductive definitions. Ph.D. thesis, University of Edinburgh (2006)
6. Brotherston, J.: Formalised inductive reasoning in the logic of bunched implications. In: Proceedings of SAS-14, *LNCS*, vol. 4634, pp. 87–103. Springer-Verlag (2007)
7. Brotherston, J., Bornat, R., Calcagno, C.: Cyclic proofs of program termination in separation logic. In: Proceedings of POPL-35, pp. 101–112. ACM (2008)
8. Brotherston, J., Gorogiannis, N.: Cyclic abduction of inductively defined safety and termination preconditions. In: Proceedings of SAS-21, *LNCS*, vol. 8723, pp. 68–84. Springer (2014)
9. Brotherston, J., Gorogiannis, N., Petersen, R.L.: A generic cyclic theorem prover. In: Proceedings of APLAS-10, LNCS, pp. 350–367. Springer (2012)
10. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Proceedings of TACAS, *LNCS*, vol. 2988, pp. 168–176. Springer (2004)
11. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Logic of Programs, Workshop, pp. 52–71. Springer-Verlag (1981)
12. Cook, B., Gotsman, A., Podelski, A., Rybalchenko, A., Vardi, M.Y.: Proving that programs eventually do something good. In: Proceedings of POPL-34, POPL '07, pp. 265–276. ACM (2007)
13. Cook, B., Khlaaf, H., Piterman, N.: On automation of CTL* verification for infinite-state systems. In: Proceedings of CAV-27, *LNCS*, vol. 9206. Springer (2015)
14. Cook, B., Koskinen, E.: Making prophecies with decision predicates. In: Proceedings of POPL-38, vol. 46, pp. 399–410. ACM (2011)
15. Cook, B., Koskinen, E.: Reasoning about nondeterminism in programs. In: Proceedings of PLDI-34, pp. 219–230. ACM (2013)
16. Dam, M.: Translating CTL* Into the Modal Mu-calculus. ECS-LFCS-. University of Edinburgh, Department of Computer Science, Laboratory for Foundations of Computer Science (1990)
17. Emerson, E.A., Halpern, J.Y.: "Sometimes" and "Not never" revisited: On branching versus linear time temporal logic. J. ACM **33**, 151–178 (1986)
18. Fix, L., Grumberg, O.: Verification of temporal properties. J. Log. Comput. **6**, 343–361 (1996)
19. Hungar, H., Grumberg, O., Damm, W.: What if model checking must be truly symbolic. In: Proceedings of CHARME, pp. 1–20. Springer-Verlag (1995)
20. Lamport, L.: Proving the correctness of multiprocess programs. IEEE Trans. Software Eng. **3**, 125–143 (1977)
21. Löding, C., Thomas, W.: Methods for the transformation of -automata: Complexity and connection to second order logic (2007)
22. Magill, S., Tsai, M.H., Lee, P., Tsay, Y.K.: Automatic numeric abstractions for heap-manipulating programs. In: Proceedings of the 37th Annual Symposium on Principles of Programming Languages, POPL '10, pp. 211–222. ACM (2010)
23. Manna, Z., Pnueli, A.: Completing the temporal picture (1991)
24. Pnueli, A.: The temporal logic of programs. In: 18th Annual Symposium on Foundations of Computer Science, pp. 46–57. IEEE (1977)
25. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in cesar. In: Proceedings of the 5th CISP, pp. 337–351. Springer-Verlag (1982)
26. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proc. LICS-17, pp. 55–74. IEEE (2002)
27. Rowe, R.N.S., Brotherston, J.: Automatic cyclic termination proofs for recursive procedures in separation logic. In: Proceedings of CPP-6. ACM (2016)
28. Schopp, U., Simpson, A.: Verifying temporal properties using explicit approximants: Completeness for context-free processes. In: Proceedings of FoSSaCS, pp. 372–386. Springer (2002)
29. Sprenger, C.: Deductive local model checking - on the verification of ctl* properties of infinite-state reactive systems. Ph.D. thesis, Swiss Federal Institute of Technology (2000)
30. Sprenger, C., Dam, M.: On the structure of inductive reasoning: circular and tree-shaped proofs in the μ-calculus. In: Proceedings of FOSSACS 2003, *LNCS*, vol. 2620, pp. 425–440. Springer-Verlag (2003)

31. Tellez, G., Brotherston, J.: Automatically verifying temporal properties of pointer programs with cyclic proof. In: Automated Deduction – CADE 26, pp. 491–508. Springer (2017)
32. Vafeiadis, V., Parkinson, M.: A marriage of rely/guarantee and separation logic. In: Proceedings of CONCUR-18, pp. 256–271. Springer (2007)
33. Vardi, M.Y.: Verification of concurrent programs: the automata-theoretic framework*. Annals of Pure and Applied Logic **51**(1), 79–98 (1991)
34. Visser, W., Barringer, H.: Practical CTL* model checking: Should spin be extended? International Journal on Software Tools for Technology Transfer **2**(4), 350–365 (2000)