Automatically Verifying Temporal Properties of Pointer Programs with Cyclic Proof

Gadi Tellez and James Brotherston

Dept. of Computer Science, University College London

Abstract. We propose 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 over memory state assertions in *separation logic*, and whose rules operate directly on the temporal modalities 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 heap-aware programs. Experimental evaluation of our tool indicates that our approach is viable, and offers an interesting alternative to traditional model checking techniques.

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

Historically, perhaps the most popular approach to 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. [15,12,10]). 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 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 [22,15,12]. 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; 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 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 *to* nil*, 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 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 paper, we instead approach the above problem via the main (perhaps less fashionable) alternative to model checking, namely the *direct deductive verification* of pointer programs. We formulate a *cyclic* proof system manipulating temporal judgements about programs, and employ 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 express a temporal property of the program when started from any state satisfying a precondition written in a fragment of *separation logic*, a well-known language for describing heap memory [26]. The core of the proof system is 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* [29,6,7,9], 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 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.

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 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.

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:

$$B ::= E = E \mid E \neq E \mid *$$

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

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 $h : \mathsf{Loc} \rightharpoonup_{\mathrm{fin}} (\mathsf{Val}\ \mathsf{List})$, 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 $\mathrm{dom}(h_1) \cap \mathrm{dom}(h_2) = \varnothing$ (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 the program configuration γ' in one step. 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. For brevity, we omit the operational semantics here, since it is essentially standard.

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 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. $\langle \epsilon, s, h\rangle \rightsquigarrow \langle \epsilon, s, h\rangle$).

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 needed to express data structures in the memory. We omit the schema for inductive predicates and their interpretations here, since they are identical to those used, e.g., in [7,9,8,27].

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 $s, h \vDash \Pi : \Sigma$ if $s, h \vDash \varpi$ for all pure formulas $\varpi \in \Pi$, and $s, h \vDash \Sigma$, where the relation $s, h \vDash A$*

between states and formulas is defined by

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

Note that the semantics of a predicate symbol, $[\![\Psi]\!] \subseteq \mathsf{Val\ List} \times \mathsf{Heaps}$*, is typically obtained from an* inductive definition *of* Ψ *in a standard way (see e.g. [6]).*

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 3. CTL assertions *are described by the grammar:*

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

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

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

Definition 4. *A configuration* γ *is a* model *of the CTL assertion* φ *if the relation* $\gamma \vDash \varphi$ *holds, defined by structural induction as follows:*

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

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 temporal assertion.

Definition 5 (Validity). *A CTL judgement* $P \vdash C : \varphi$ *is* valid *if and only if, for all memory states* (s,h) *such that* $s,h \vDash P$*, we have* $\langle C, s, h \rangle \vDash \varphi$*.*

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 1. 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 a program. In the specific case of our language, the nondeterministic constructs are (i) nondeterministic while; (ii) nondetreministic if; and (iii) memory allocation; it is only for these constructs that we need a specific rule for each case, as shown in our symbolic execution rules. 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. For brevity, we omit here the somewhat complex unfolding rule for inductive predicates, but similar rules can be found in, e.g., [7,9,8,27].

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 6 (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 7 (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 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*
- $\varphi_i = \varphi_{i+1}$ *whenever* J_i *is the conclusion of the consequence rule (Cons).*

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.*

Symbolic execution rules:

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

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

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

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

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

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

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

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

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:

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

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

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

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

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

Fig. 1. Proof rules for CTL judgements. We write $\bigcirc\varphi$ to mean "either $\Box\varphi$ or $\Diamond\varphi$".

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. Again, see [7,9,8,27] for similar notions.*

Definition 8 (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 $\Box$-trace, $\Diamond$-trace or precondition trace following some tail $(P_i \vdash C_i : \varphi_i)_{i \geq n}$ of the path.*

Example 2 *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 2 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.

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 Definition 7, 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 $\mathrm{length}(\pi') \leq \mathrm{length}(\pi)$. Moreover, $\mathrm{length}(\pi) < \mathrm{length}(\pi')$ 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 Definition 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 $\mathrm{height}(\kappa') \leq \mathrm{height}(\kappa)$. Moreover, $\mathrm{height}(\kappa') < \mathrm{height}(\kappa)$ when R is a symbolic execution rule.*

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

Proof. (Sketch) 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

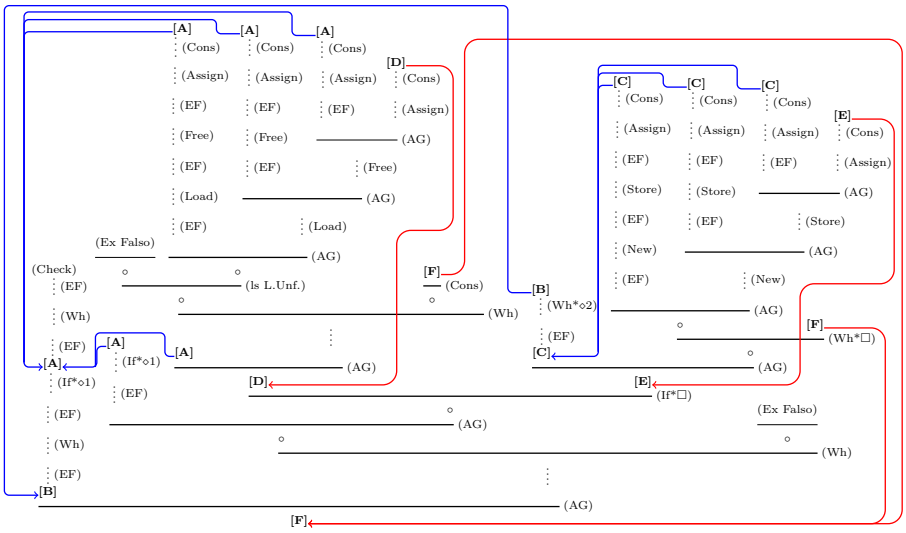

Fig. 2. Single threaded monolithic server example

an infinite path $(P_i \vdash C_i : \varphi_i)_{i \geq 0}$ in $\mathcal{P}$ of invalid judgements. Since $\mathcal{P}$ is a cyclic proof, by Defn. 8 there exists an infinitely progressing trace following some tail $(P_i \vdash C_i : \varphi_i)_{i \geq n}$ of the 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. 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; see [7] for details.

The inductive-coinductive dichotomy shows nicely 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 program progress on the right of the judgements. For inductive properties, we need an infinite descent on the left of the 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. These fairness constraints are usually categorised as *weak* and *strong* fairness [20]. However, since weak fairness requirements are usually restricted to the parallel composition of processes, a property that our programming language lacks, we limit ourselves to the treatment of strong fairness.

Definition 9 (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 10 (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 $\langle C, s, h \rangle \vDash_f \varphi$, where $\vDash_f$ is the satisfaction relation obtained from $\vDash$ in Definition 4 by interpreting the temporal operators as quantifying over* fair *paths, rather than all paths. For example, the clause for AG becomes*

$$\gamma \vDash_f AG\varphi \iff \forall \text{ fair } \pi \text{ starting from } \gamma.\ \forall \gamma' \in \pi.\, \gamma' \vDash_f \varphi.$$

Definition 11. *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 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* $(J_i)_{i\geq 0}$ *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* $\textbf{\textit{if}} * \textbf{\textit{then}}\, C_i\, \textbf{\textit{else}}\, C_j\, \textbf{\textit{fi}}$ *or* $\textbf{\textit{while}} * \textbf{\textit{do}}\, C_i\, \textbf{\textit{od}}\, C_j$.

Given these new definitions, the global soundness condition of our proofs is restricted to account only for fair paths in a pre-proof.

Definition 12 (Bad pre-proof). *A pre-proof* $\mathcal{P}$ *is* bad *if there is an infinite path in* $\mathcal{P}$ *such that the rule* $(Wh^* \Diamond 1)/(If^* \Diamond 1)$ *is applied infinitely often and* $(Wh^* \Diamond 2)/(If^* \Diamond 2)$ *is applied only finitely often, or vice versa.*

Definition 13 (Fair proof). *A pre-proof* $\mathcal{P}$ *is a* fair cyclic proof *if it is not bad, 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 1 (Decidable soundness condition). *It is decidable whether a pre-proof is a valid fair cyclic proof.*

Proof. (Sketch) To check that a pre-proof $\mathcal{P}$ is not bad, we construct two Büchi automata; the first one $\mathcal{A}_{B_1}$ accepts all infinite paths in $\mathcal{P}$ such that the rule $((\text{Wh}^* \Diamond 1))/(\text{If}^* \Diamond 1)$ is applied infinitely often. The second Büchi automata $\mathcal{A}_{B_2}$ accepts all infinite paths such that the rule $(\text{Wh}^* \Diamond 2)/(\text{If}^* \Diamond 2)$ is applied infinitely often. We then check that the following relation holds of the languages accepted by each automata: $\mathcal{L}(\mathcal{A}_{B_1}) \subseteq \mathcal{L}(\mathcal{A}_{B_2})$ and $\mathcal{L}(\mathcal{A}_{B_2}) \subseteq \mathcal{L}(\mathcal{A}_{B_1})$, where language inclusion of Büchi automata is decidable.

Moreover, to check that there exists an infinitely progressing trace along every infinite path of $\mathcal{P}$ we construct two automata over strings of nodes of $\mathcal{P}$. The *fair automata* $\mathcal{A}_{Fair}$ that accepts all infinte fair paths in $\mathcal{P}$ is a Streett automata 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). The *trace automata* $\mathcal{A}_{Trace}$ is a Büchi automata that accepts all infinite paths in $\mathcal{P}$ such that an infinitely progressing trace exists along the path (cf. [5]). $\mathcal{P}$ is then a valid cyclic proof if and only if $\mathcal{A}_{Trace}$ accepts all strings accepted by $\mathcal{A}_{Fair}$. We are then done since Streett automata can be transformed into Büchi automata [21] and inclusion between Büchi automata is decidable.

Example 3 *We return to our server program from Examples 1 and 2. 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.* $AGAF(\mathsf{emp})$*. Our server program in fact does* not *satisfy this property, because the program can always choose to execute the second inner loop infinitely often, adding job requests to the list forever. However, it* does *satisfy this property under the assumption of fair execution, which prevents the second loop from being executed infinitely often without executing the first loop.*

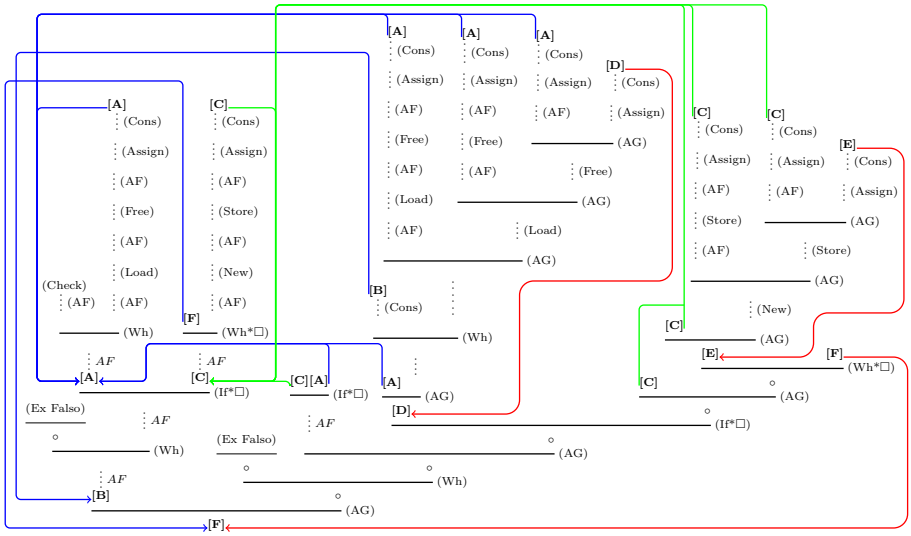

Fig. 3. Single threaded monolithic server example

Figure 3 shows the proof of this property in the adaptation of our system that is aware of fairness constraints as described above. Adding the fairness constraints relaxes the conditions under which back-links can be formed. This relaxed condition can be seen in back-links depicted in green as they cause an infinite path with no valid trace to be formed. Yet, because this infinite path is unfair*, it is not considered in the global soundness condition. Our pre-proof qualifies as a valid 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 $(s, h) \vDash P$ but $\langle C, s, h\rangle \nvDash_f \varphi$. Then, 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. By Defn. 13 we know that said infinite path is a *fair* path (as any unfair path has been ruled out by requiring that $\mathcal{P}$ is not a *bad pre-proof* according to Defn. 12). Since the path is an infinite fair path, by Defn. 13 we also know that there is an infinitely progressing $\Box$-trace, $\Diamond$-trace or precondition trace following some tail of the path. Showing that the existence of an infinitely progressing trace along the path leads to a contradiction follows the same argument as in Section 3.

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 2 and 3). 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 non-terminating programs similar to Example 1. Our test suite can be seen as an adaptation of the common model checking benchmarks presented in [14,15] for the verification of temporal properties of nondeterministic programs. Roughly speaking, operations/iterations on integer variables in the original benchmarks are replaced in favour of operations/iterations 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).

We show the results of the evaluation of the CTL system and its extension to consider fairness constraints 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 a few factors such as the complexity of the program in question and temporal property to verify, 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 $P \vDash Q$ 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.

Note that despite the encouraging results, the implementation is not without limitations as 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.

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 [12], CEGAR [15], bounded model checking [10] and automata theory [13] have enabled the treatment of infinite transition systems.

The present paper takes the deductive verification approach. A common limitation of early proof systems for various temporal logics is their restriction to finite state transition systems [18,19,4]. In the realm of infinite state systems, previous proof systems for verifying temporal properties of arbitrary transition systems [23,30] have shed some light on the soundness and relative completeness of deductive verification. However, these early systems have typically relied upon complex verification conditions that are seemingly difficult to fully automate, arguably the most cited argument against deductive verification. In contrast, our

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	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 * emp))	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 :u 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' : emp	No	0.27
PostgreSQL	w=true ∧ s=s' ∧ f=f' : emp	AGEF w=true ∧ s=s' ∧ flag=f' : emp	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 1. Experimental results.

proof system can handle infinite state, non-terminating programs, even under fairness restrictions, and we provide an implementation and evaluation, showing that it can indeed work in practice.

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 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 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 [30]. 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. 8, 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 [29,5,9].

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 [4,7,30]. 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 [32]. 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 [33] and the instrumentation of the original program with auxiliary constructs [13].

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. [31]). 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. [4,30]). 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 (November 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. pp. 350–367. LNCS, 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., Khlaaf, H., Piterman, N.: On automation of CTL* verification for infinite-state systems. In: Proceedings of CAV-27. LNCS, vol. 9206. Springer (2015)
13. Cook, B., Gotsman, A., Podelski, A., Rybalchenko, A., Vardi, M.Y.: Proving that programs eventually do something good. In: Proceedings of POPL-34. pp. 265–276. POPL '07, ACM (2007)
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. pp. 211–222. POPL '10, 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., 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)
30. 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)
31. Vafeiadis, V., Parkinson, M.: A marriage of rely/guarantee and separation logic. In: Proceedings of CONCUR-18. pp. 256–271. Springer (2007)
32. Vardi, M.Y.: Verification of concurrent programs: the automata-theoretic framework*. Annals of Pure and Applied Logic 51(1), 79–98 (1991)
33. 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)