A Generic Dynamic Logic for Program Reasoning based on Operational Semantics

Yuanrui Zhang[0000−0002−0685−6905] and Zhibin Yang[0000−0002−9888−6975]

Collage of Software, Nanjing University of Aeronautics and Astronautics, China
yuanruizhang@nuaa.edu.cn,yangzhibin168@163.com

Abstract. Dynamic logic is a valuable formalism that has many applications in ensuring the correctness of safety-critical systems. We present a novel theory of parameterized dynamic logic, namely DL_p, for specifying and reasoning about program models based on their operational semantics. Different from most dynamic logics that deal with regular expressions or a particular type of models, DL_p allows arbitrary forms of programs and formulas according to specific domains. It provides a language-independent proof calculus under dynamic-logic settings, supporting symbolic-execution-based reasoning with a general notion of labels for capturing program configurations. To admit certain infinite proof deductions caused by loop programs, we adapt the cyclic proof approach to the theory of DL_p by building a cyclic proof structure specific to DL_p. The soundness of DL_p is analyzed and formally proved. A case study displays an instantiation of DL_p in particular domains, demonstrating the potential usage of DL_p in program verification.

Keywords: Dynamic Logic · Program Deduction · Verification Framework · Cyclic Proof · Operational Semantics · Symbolic Execution

1 Introduction

Dynamic logic [20] has proven to be a valuable formalism for specifying and reasoning about different types of programs. It has been successfully applied to domains such as process algebras [6], programming languages [5], synchronous systems [50,49], hybrid systems [35,36] and probabilistic systems [34,26]. These theories have inspired the development of related verification tools for safety-critical systems, such as KIV [39], KeY [43], KeYmaera [37], and the tools developed in [50,14]. Recent developments in dynamic logic include a range of extensions aimed at addressing contemporary challenges, such as ensuring the correctness of blockchain systems [23], describing hyperproperties [19], and verifying quantum computations [47,14]. It also has attracted attention as a promising framework for "incorrectness reasoning", as explored recently in [33,51].

As an extension of modal logic, dynamic logic embeds a program α into the modality $\Box$ of a modal formula $\Box\phi$ as a form: $[\alpha]\phi$, meaning that after all executions of α, formula ϕ holds. This so-called "dynamic formula" allows multiple and nested modalities in forms like $[\alpha]\phi \to \langle\beta\rangle\psi$ and $[\alpha]\langle\beta\rangle\phi$ (where

$\langle\cdot\rangle$ is the dual modality of $[\cdot]$), being able to directly express and reason about more complex program properties than Hoare triple $\{\phi\}\alpha\{\psi\}$ in Hoare logics (e.g. [22,40]). Particularly, when restricting a program α to be deterministic, formulas $\phi \to [\alpha]\psi$ and $\phi \to \langle\alpha\rangle\psi$ exactly capture the meanings of partial and total correctness of program α respectively in Hoare logics.

Program logics, like standard dynamic logics [20] and Hoare logics [22], are usually in explicit forms. When applying them to a certain system model or computer program, new theories are required to adapt to the target languages. For instance, to apply first-order dynamic logic (FODL) to the verification of Java programs, many primitives in Java and inference rules specific to the structures of Java need to be added into the FODL theory (cf. [5]). Otherwise, additional language transformations are inevitable, causing loss of critical program structural information during deductions. However, many languages in reality, such as AADL, UML/MARTE, Esterel [7], Java, C, etc., often have complex structures. Building a specific logic theory for them requires a lot of work. Moreover, the proof system of a logic theory is often error prone, thus requiring validation of its soundness (and even completeness), which also can be costly. One example is that Verifiable C [3] spends nearly 40,000 lines of Coq code to define and validate its logic theory for C programming language based on separation logic [40].

Another main issue is that the proof systems of these program logics are often based on programs' denotational semantics (cf. [20]) (, in which the semantics of a program is usually interpreted as a set of behavioral traces). For many interesting computer systems and programs, on the other hand, operational semantics — which describes that how a program α under a configuration σ (denoted by (α, σ)) is transitioned to another program α' under a configuration σ' (denoted by (α', σ')) — is in their nature. In these cases, the consistency from denotational semantics to operational semantics has to be validated (cf. [41,31]). This thus increases the burden of using the logic theories. Moreover, the denotational semantics of some languages are not compositional w.r.t. some of their language operators. Typical examples are synchronous programming languages like Esterel [8] and Quartz [17]. For these models, reasoning based on operational semantics is more direct, in the sense that we do not need to perform extra program transformations. [17] shows an example of non-compositional derivations of Quartz programs that rely on heavy program transformations.

This paper focuses on a theoretical solution of the above two problems in existing dynamic-logic theories. We present a novel dynamic logic called "parameterized dynamic logic" (abbreviated as DL_p) for reasoning about programs based on their operational semantics. Generally speaking, DL_p is a unified dynamic-logic theory modulo programs' operational semantics. It provides a language-independent verification framework in which program deductions rely solely on program transitions of a universal form: $(\alpha, \sigma) \longrightarrow (\alpha', \sigma')$ (for arbitrary $\alpha, \alpha', \sigma, \sigma'$). Instead of carefully designing rules according to programs' denotational semantics, the inference rules for program transitions can be built directly from the operational semantics of programs. Their validations are straightforward without the need of additional proofs.

The methodology of reasoning based on operational semantics has been proposed and studied for years within different mathematical theories, among them the work based on rewriting logics [41,42,45,12], set theory and coinduction [31,27], and program updates[35,4,5] are closest to our work (see Section 6 for a detailed comparison). Except a few work such as [32,21], to the best of our knowledge, most of the previous work has not yet addressed a similar approach under dynamic-logic settings, i.e., to provide an efficient logical calculus for deriving dynamic formulas. In our opinion, it is valuable to fill in this gap.

Illustration of Main Idea. Informally, DL_p treats program α and formula ϕ of $[\alpha]\phi$ as 'parameters', while introducing 'labels' σ as program configurations to capture current program states for symbolic executions. The structures of α, ϕ and σ are irrelevant to the main skeleton of the verification framework. In DL_p, we derive a labelled DL_p dynamic formula $\sigma : [\alpha]\phi$ instead of $[\alpha]\phi$. When σ represents an arbitrary configuration, $\sigma : [\alpha]\phi$ exactly captures the same meaning as $[\alpha]\phi$.

To see how we can benefit from deriving a labelled DL_p formula, consider a simple example. We want to prove a formula $\phi_1 =_{df} (x \geq 0 \to [x := x+1]x > 0)$ in FODL [38], where x is a variable ranging over integers $\mathbb{Z}$. Intuitively, formula ϕ_1 means that if $x \geq 0$ holds, then $x > 0$ holds after assigning the expression $x + 1$ to x. In FODL, to derive ϕ_1, we apply rule:

$$\frac{\phi[x/e]}{[x := e]\phi} \ {\scriptstyle (x:=e)}$$

for assignment on formula $[x := x+1]x > 0$ by substituting x of $x > 0$ with $x+1$, and obtain $x + 1 > 0$. Formula ϕ_1 thus becomes $\phi_1' =_{df} (x \geq 0 \to x + 1 > 0)$, which is true for any $x \in \mathbb{Z}$.

In DL_p, however, formula ϕ_1 can be expressed as a form: $\psi_1 =_{df} (t \geq 0 \to \{x \mapsto t\} : [x := x + 1]x > 0)$. In ψ_1, formula $[x := x + 1]x > 0$ is labelled by configuration $\{x \mapsto t\}$, which means that variable x stores value t (with t a free variable). With a label explicitly showing up, to derive formula ψ_1, we instead directly perform a program transition of $x := x + 1$ as:

$$(x := x + 1, \{x \mapsto t\}) \longrightarrow (\downarrow, \{x \mapsto t + 1\}), \qquad (ex\ x := e)$$

which assigns the value $t + 1$ to x afterwards. Here $\downarrow$ indicates a program termination. Formula ψ_1 thus becomes $\psi_1' =_{df} (t \geq 0 \to \{x \mapsto t + 1\} : x > 0)$, by replacing the part '$\{x \mapsto t\} : [x := x+1]$' with its execution result '$\{x \mapsto t+1\}$'. Formula $\{x \mapsto t + 1\} : x > 0$ exactly means $t + 1 > 0$, by replacing x with its current value $t + 1$. So from ψ_1' we obtain formula $t \geq 0 \to t + 1 > 0$, which is exactly formula ϕ_1' (modulo free-variable renaming).

In the above example, the form of the transition $(ex\ x := e)$ directly comes from the operational semantics of the assignment $x := e$. It is universal for all types of program transitions. By choosing different labels one can specify the rules for different languages. On the other hand, rule $(x := e)$ here is actually a special assignment rule in FODL. It cannot be directly applied to other languages, for example, a Java statement $x := new\ C(...)$ that creates a new object of class C (cf. [5]).

Main Contributions & Challenges. In this paper, we firstly define the syntax and semantics of DL_p based on the standard theory of propositional dynamic logic (PDL) [16]. The challenge of this part is to find a suitable way to define the semantics of DL_p (Definition 4), since its ingredients are all unknown parameters. Following the conventions of defining a dynamic logic [20], we propose a so-called "program-labelled" (PL) Kripke structure (Definition 2) to support describing the operational semantics of programs in arbitrary forms. And we follow an approach similar in [15] to propose a labelled sequent calculus (Section 3.1) for symbolic-execution-based reasoning. But here the forms of labels in DL_p can be an explicit program structure rather than just an abstract state in [15] (see Section 6).

After defining the logic theory, we build a cyclic verification framework for DL_p. Cyclic proof approach (cf. [11]) is a powerful technique to admit a certain type of infinite deductions (i.e. "cyclic proofs") caused by symbolic executions of programs with loop structures (Section 3.3). It has been attracting more and more attention and recently has been applied to many logic theories such as [48,24,2]. We investigate this approach and adapt it to the theory of DL_p. The main challenges of this part are (1) identifying a sound cyclic proof system, where the most critical work is to design rule (Sub) and the "substitutions of labels" (Definition 7); and (2) constructing a cyclic proof structure, where the key step is to define "progressive derivation traces" (Definition 9). At last, as an important contribution, we prove the soundness of our cyclic proof system (Section 5).

To sum up, the main contributions of this paper are threefold:

- We define the syntax and semantics of DL_p formulas.
- We build a labelled proof system for DL_p.
- We construct a sound cyclic proof system for DL_p and prove its soundness.

The rest of the paper is organized as follows. Section 2 defines the syntax and semantics of DL_p formulas. In Section 3, we propose a cyclic proof system for DL_p. In Section 4, we give an example of cyclic deductions of DL_p formulas. Section 5 analyzes and proves the soundness of DL_p. Section 6 introduces related work, while Section 7 makes a conclusion and discusses about future work. Due to the space limit, some more details of this work is given in an extended version of this paper online [1].

2 Dynamic Logic DL_p

The theory of DL_p extends PDL [16] by permitting the program α and formula ϕ in modalities $[\alpha]\phi$ to take arbitrary forms, subject to a restriction condition (Definition 10). A brief introduction to PDL and FODL is given in the online report [1].

We assume two pre-defined sets $\mathbf{P}$ and $\mathbf{F}$, namely *parameters* of DL_p. $\mathbf{P}$ is a set of programs, in which we distinguish a special program $\downarrow \in \mathbf{P}$ called *termination*. $\mathbf{F}$ is a set of formulas.

Definition 1 (DL_p Formulas). *A dynamic logical formula ϕ w.r.t. parameters $\mathbf{P}$ and $\mathbf{F}$, called a "parameterized dynamic logic" (DL_p) formula, is defined as follows in BNF form:*

$$\phi =_{df} F \mid \neg\phi \mid \phi \wedge \phi \mid [\alpha]\phi,$$

where $F \in \mathbf{F}$, $\alpha \in \mathbf{P}$. We denote the set of DL_p formulas as $\mathfrak{F}_{dlp}$.

Intuitively, formula $[\alpha]\phi$ means that after all executions of program α, formula ϕ holds. $\langle\cdot\rangle$ is the dual operator of $[\cdot]$. Formula $\langle\alpha\rangle\phi$ is expressed as $\neg[\alpha]\neg\phi$. Other formulas with logical connectives such as $\vee$ and $\to$ can be expressed by formulas with $\neg$ and $\wedge$ accordingly. Note that in a DL_p formula, there can be multiple and nested modalities, e.g., both $[\alpha]\phi \to \langle\beta\rangle\psi$ and $[\alpha](\langle\beta\rangle\phi)$ are legal DL_p formulas.

Following the convention of defining a dynamic logic (cf. [20]), we introduce a novel Kripke structure to tackle parameterized program behaviours in $\mathbf{P}$.

Definition 2 (Program-Labelled Kripke Structure). *A "program-labelled" (PL) Kripke structure w.r.t. parameters $\mathbf{P}$ and $\mathbf{F}$ is a triple*

$$K(\mathbf{P}, \mathbf{F}) =_{df} (\mathcal{S}, \longrightarrow, \mathcal{I}),$$

where S is a set of worlds; $\longrightarrow \subseteq \mathcal{S} \times (\mathbf{P} \times \mathbf{P}) \times \mathcal{S}$ is a set of relations labelled by program pairs, in the form of $w_1 \xrightarrow{\alpha/\alpha'} w_2$ for some $w_1, w_2 \in \mathcal{S}$, $\alpha, \alpha' \in \mathbf{P}$; $\mathcal{I} : \mathbf{F} \to \mathcal{P}(\mathcal{S})$ is an interpretation of formulas in $\mathbf{F}$ on the power set of worlds, satisfying that $w \not\xrightarrow{\downarrow/\alpha} \cdot$ for any $w \in \mathcal{S}$ and $\alpha \in \mathbf{P}$ (capturing the meaning of program $\downarrow$).

PL Kripke structures differ from the Kripke structures of PDL by introducing program-labelled relations $w_1 \xrightarrow{\alpha/\alpha'} w_2$, which describe programs' transitional behaviours. Intuitively, it means that from world w_1, program α is transitioned to program α', ending with world w_2.

Below in this paper, **our discussion is always based on an assumed PL Kripke structure namely** $K(\mathbf{P}, \mathbf{F}) = (\mathcal{S}, \longrightarrow, \mathcal{I})$, where transitional behaviours $\longrightarrow$ is usually captured by a set of inference rules on program transitions (see Section 3.2), as the operational semantics of $\mathbf{P}$.

Example 1 (An Instantiation of Programs and Formulas). Consider a while program WP in an instantiation namely $\mathbf{P}_W$ of parameter $\mathbf{P}$:

$$WP =_{df} \{while\ (n > 0)\ do\ s := s + n\ ;\ n := n - 1\ end\ \}.$$

Given an initial value of variables n and s, program WP computes the sum from n to 1 stored in the variable s. The underlying theory of programs $\mathbf{P}_W$ is the arithmetic number theory in integer domain $\mathbb{Z}$. Let Var_W be the set of integer variables. $x := e$ is an assignment, which assigns the value of expression e to variable x. In the PL Kripke structure $K_W = (\mathcal{S}_W, \longrightarrow, \mathcal{I}_K)$ for $\mathbf{P}_W$, a world $w \in S_W$ is a mapping $w : Var_W \to \mathbb{Z}$ from variables to integers. Programs' transitional behaviours of $\mathbf{P}_W$ is captured by the relations on K_W. For example, we have a relation $w \xrightarrow{x:=x+1/\downarrow} w[x \mapsto w(x)+1]$, where $w[x \mapsto v]$ is a mapping that only differs from w on mapping x to value v. The formulas in while programs namely $\mathbf{F}_W$ are the usual arithmetic first-order logical formulas in integer domain $\mathbb{Z}$.

Definition 3 (Execution Path). *An "execution path" on K is a finite sequence of relations on $\longrightarrow$: $w_1 \xrightarrow{\alpha_1/\beta_1} ... \xrightarrow{\alpha_n/\beta_n} w_{n+1}$ ($n \geq 0$) satisfying that $\beta_n \in \{\downarrow\}$, and $\beta_i = \alpha_{i+1} \notin \{\downarrow\}$ for all $1 \leq i < n$.*

In Definition 3, the execution path is sometimes simply written as a sequence of worlds: $w_1...w_{n+1}$. When $n = 0$, the execution path is a single world w_1 (without any relations on $\longrightarrow$).

Definition 4 (Semantics of DL_p Formulas). *Given a DL_p formula ϕ, the satisfaction of ϕ by a world $w \in S$ under K, denoted by $K, w \models \phi$, is inductively defined as follows:*

1. *$K, w \models F$ where $F \in \mathbf{F}$, if $w \in \mathcal{I}(F)$;*
2. *$K, w \models \neg\phi$, if $K, w \not\models \phi$;*
3. *$K, w \models \phi \wedge \psi$, if $K, w \models \phi$ and $K, w \models \psi$;*
4. *$K, w \models [\alpha]\phi$, if for all execution paths of the form: $w \xrightarrow{\alpha/\cdot} ... \xrightarrow{\cdot/\downarrow} w'$ for some $w' \in \mathcal{S}$, $K, w' \models \phi$.*

According to the definition of operator $\langle\cdot\rangle$, its semantics is defined such that $K, w \models \langle\alpha\rangle\phi$, if there exists an execution path of the form $w \xrightarrow{\alpha/\cdot} ... \xrightarrow{\cdot/\downarrow} w'$ for some $w' \in \mathcal{S}$ such that $K, w' \models \phi$.

A DL_p formula ϕ is called *valid* w.r.t. K, denoted by $K \models \phi$ (or simply $\models \phi$), if $K, w \models \phi$ for all $w \in \mathcal{S}$.

Example 2 (DL_p Specifications). A property of program WP (Example 1) is described as the following formula

$$(n \geq 0 \wedge n = N \wedge s = 0) \rightarrow [WP](s = ((N+1)N)/2),$$

which means that given an initial condition of n and s, after executing WP, s equals to $((N+1)N)/2$, which is the sum of $1 + 2 + ... + N$, with N a free variable in $\mathbb{Z}$. We will prove an equivalent labelled version of this formula in DL_p in Section 4.

3 A Cyclic Proof System for DL_p

We propose a cyclic proof system for DL_p. We firstly propose a labelled proof system P_{dlp} to support reasoning based on symbolic executions (Section 3.2). Then we construct a cyclic proof structure for system P_{dlp}, which support deriving infinite proof trees under certain conditions (Section 3.3). Section 3.1 introduces the notions of labelled sequent calculus and cyclic proof.

3.1 Prerequisites

Labelled Sequent Calculus. We assume a set $\mathbf{L}$ of labels as a *parameter* of DL_p. A label mapping $\mathfrak{m} : \mathbf{L} \to \mathcal{S}$ maps each label of $\mathbf{L}$ to a world of set $\mathcal{S}$. Denote the set of all label mappings as $\mathbf{M}$. A *labelled DL_p formula* is of the form $\sigma : \phi$, where $\sigma \in \mathbf{L}$ and $\phi \in \mathfrak{F}_{dlp}$. Denote the set of all labelled DL_p formulas as $\mathfrak{F}_{ldlp}$.

From program-labelled relations defined in Definition 2 we introduce symbolic executions of programs as a type of abstract transitions on labels and program terminations. A *program transition* is a relation of the form $\sigma \xrightarrow{\alpha/\alpha'} \sigma'$ (also written as $(\alpha, \sigma) \longrightarrow (\alpha', \sigma')$ below) between labels and labelled by a program pair, with $\sigma, \sigma' \in \mathbf{L}$ and $\alpha, \alpha' \in \mathbf{P}$. Call pair (α, σ) a *program state*. We use $\mathfrak{F}_{pt}$ to represent the set of all program transitions. A *program termination* is a relation of the form $\sigma \Downarrow \alpha$ between a label and a program, where $\sigma \in \mathbf{L}$ and $\alpha \in \mathbf{P}$. The set of all program terminations is denoted by $\mathfrak{F}_{ter}$.

A *sequent* is a logical argumentation of the form: $\Gamma \Rightarrow \Delta$, where Γ and Δ are finite multi-sets of formulas, called the *left side* and the *right side* of the sequent respectively. We use dot $\cdot$ to express Γ or Δ when they are empty sets. Intuitively, a sequent $\Gamma \Rightarrow \Delta$ means that if all formulas in Γ hold, then one of formulas in Δ holds. We use ν to represent a sequent.

A *labelled sequent* is a sequent in which each formula is a formula in $\mathfrak{F}_{ldlp} \cup \mathfrak{F}_{pt} \cup \mathfrak{F}_{ter}$. We use τ to represent a formula of a labelled sequent.

Definition 5 (Semantics of Formulas in Labelled Sequents). *Given a labelled sequent ν and a label mapping $\mathfrak{m} \in \mathbf{M}$, the satisfaction relation $K, \mathfrak{m} \models \tau$ of a formula τ in ν under K is defined as follows according to the different cases of τ:*

1. $K, \mathfrak{m} \models \sigma : \phi$, *if* $K, \mathfrak{m}(\sigma) \models \phi$*;*
2. $K, \mathfrak{m} \models \sigma \xrightarrow{\alpha/\alpha'} \sigma'$, *if* $\mathfrak{m}(\sigma) \xrightarrow{\alpha/\alpha'} \mathfrak{m}(\sigma')$ *is a relation on* K*;*
3. $K, \mathfrak{m} \models \sigma \Downarrow \alpha$, *if there exists an execution path* $\mathfrak{m}(\sigma) \xrightarrow{\alpha/\cdot} ... \xrightarrow{\cdot/\downarrow} w$ *on* K *for some world* $w \in \mathcal{S}$.

A formula τ in a labelled sequent is *valid*, denoted by $K \models \tau$ (or simply $\models \tau$), if $K, \mathfrak{m} \models \tau$ for all $\mathfrak{m} \in \mathbf{M}$. According to the meaning of a sequent above, a labelled sequent $\Gamma \Rightarrow \Delta$ is *valid*, if for every $\mathfrak{m} \in \mathbf{M}$, $K, \mathfrak{m} \models \tau$ for all $\tau \in \Gamma$ implies $K, \mathfrak{m} \models \tau'$ for some $\tau' \in \Delta$.

For a multi-set Γ of formulas, we write $K, \mathfrak{m} \models \Gamma$ to mean that $K, \mathfrak{m} \models \tau$ for all $\tau \in \Gamma$.

Example 3 (Instantiation of Labels). In while programs, we consider a type of labels namely $\mathbf{L}_W$ of the form: $\{x_1 \mapsto e_1, ..., x_n \mapsto e_n\}$ $(n \geq 0)$ as program configurations, where each variable $x_i \in Var_W$ stores a unique value of arithmetic expression e_i $(1 \leq i \leq n)$. To make it simple, we restrict that variables $x_1, ...x_n$ must appear in the discussed programs and any free variable in $e_1, ..., e_n$ cannot be any of $x_1, ..., x_n$. For example, in program *WP* (Example 1), $\{n \mapsto N, s \mapsto 0\}$ is a configuration that maps n to value N (as a free variable) and s to 0.

Example 4 (Instantiation of Label Mappings). In while programs, we consider a set $\mathbf{M}_W$ of label mappings where each label mapping is associated to a world, denoted by $\mathfrak{m}_w$ for some $w \in \mathcal{S}_W$. Given a configuration $\sigma =_{df} \{x_1 \mapsto e_1, ..., x_n \mapsto e_n\}$ $(n \geq 1)$, $\mathfrak{m}_w(\sigma)$ is defined as a world such that (1) $\mathfrak{m}_w(\sigma)(x_i) = w(e_i)$ for each x_i $(1 \leq i \leq n)$; (2) $\mathfrak{m}_w(\sigma)(y) = w(y)$ for other variable $y \in \mathit{Var}_W$. Where $w(e_i)$ returns a value by substituting each free variable x of e_i with value $w(x)$. For example, let $w(N) = 5$, then we have $\mathfrak{m}_w(\{n \mapsto N, s \mapsto 0\})(n) = w(N) = 5$, $\mathfrak{m}_w(\{n \mapsto N, s \mapsto 0\})(s) = 0$. In fact, in this case, $\mathfrak{m}_w(\{n \mapsto N, s \mapsto 0\}) = w$.

An *inference rule* is of the form $\dfrac{\nu_1 \quad ... \quad \nu_n}{\nu}$, where each of ν, ν_i $(1 \leq i \leq n)$ is also called a *node*. Each of $\nu_1, ..., \nu_n$ is called a *premise*, and ν is called the *conclusion*, of the rule. The semantics of the rule is that the validity of sequents $\nu_1, ..., \nu_n$ implies the validity of sequent ν. A formula τ of node ν is called the *target formula* if except τ other formulas are kept unchanged in the derivation from ν to some node ν_i $(1 \leq i \leq n)$. And in this case other formulas except τ in node ν are called the *context* of ν. A formula pair (τ_1, τ_2) with τ_1 in ν and τ_2 in some ν_i is called a *conclusion-premise* (CP) pair of the derivation from ν to ν_i.

Proof & Preproof & Cyclic Proof. A *proof tree* (or *proof*) is a finite tree structure formed by making derivations backward from a root node. In a proof tree, a node is called *terminal* if it is the conclusion of an axiom.

In the cyclic proof approach (cf. [11]), a *preproof* is an infinite proof tree (i.e. some of its derivations contain infinitely many nodes) in which there exist non-terminal leaf nodes, called *buds*. Each bud is identical to one of its ancestors in the tree. The ancestor identical to a bud ν is called a *companion of* ν. A *derivation path* in a preproof is an infinite sequence of nodes $\nu_1\nu_2...\nu_m...$ $(m \geq 1)$ starting from the root node ν_1, where each node pair (ν_i, ν_{i+1}) $(i \geq 1)$ is a CP pair of a rule. A proof tree is *cyclic*, if it is a preproof in which there exists a "progressive derivation trace", whose definition depends on specific logic theories (see Definition 9 later for DL_p), over every derivation path.

A *proof system* P consists of a finite set of inference rules. We say that a node ν can be derived from P, denoted by $P \vdash \nu$, if a proof tree can be constructed (with ν the root node) by applying the rules in P, which satisfies either (1) all of its leaf nodes terminate or (2) it is a cyclic proof tree.

3.2 A Proof System for DL_p

The labelled proof system P_{dlp} of DL_p is defined as: $P_{dlp} =_{df} P_{ldlp} \cup P_{pt} \cup P_{ter}$, where P_{ldlp} is a set of *core rules* listed in Table 2, P_{pt} and P_{ter} are two finite pre-defined sets of rules according to parameter $\mathbf{P}$.

Sets P_{pt} and P_{ter} are for deriving program transitions $\mathfrak{F}_{pt}$ and terminations $\mathfrak{F}_{ter}$ respectively. Each rule in P_{pt} (resp. P_{ter}) has a restricted form: $\dfrac{\Gamma_1 \Rightarrow \Delta_1 \quad ... \quad \Gamma_n \Rightarrow \Delta_n}{\Gamma \Rightarrow \tau, \Delta}$, where $n \geq 0$, $\tau \in \mathfrak{F}_{pt}$ (resp. $\tau \in \mathfrak{F}_{ter}$), τ is the target formula of the rule.

P_{pt} and P_{ter} are assumed to be *sound and complete* for the operational semantics of $\mathbf{P}$ in the sense of the following definition.

$\dfrac{}{\Gamma \Rightarrow (x := e, \sigma) \longrightarrow (\downarrow, \sigma^x_e), \Delta}\ (x{:=}e)$	$\dfrac{\Gamma \Rightarrow (\alpha_1, \sigma) \longrightarrow (\alpha_1', \sigma'), \Delta}{\Gamma \Rightarrow (\alpha_1; \alpha_2, \sigma) \longrightarrow (\alpha_1'; \alpha_2, \sigma'), \Delta}\ (;)$
$\dfrac{\Gamma \Rightarrow (\alpha_1, \sigma) \longrightarrow (\downarrow, \sigma'), \Delta}{\Gamma \Rightarrow (\alpha_1; \alpha_2, \sigma) \longrightarrow (\alpha_2, \sigma'), \Delta}\ (;\downarrow)$	$\dfrac{\Gamma, \sigma : \phi \Rightarrow (\alpha, \sigma) \longrightarrow (\alpha', \sigma'), \Delta \quad \Gamma \Rightarrow \phi : \sigma, \Delta}{\Gamma \Rightarrow (\mathit{while}\ \phi\ \mathit{do}\ \alpha\ \mathit{end}, \sigma) \longrightarrow (\alpha';\ \mathit{while}\ \phi\ \mathit{do}\ \alpha\ \mathit{end}, \sigma'), \Delta}\ (wh1)$
$\dfrac{\Gamma, \sigma : \phi \Rightarrow (\alpha, \sigma) \longrightarrow (\downarrow, \sigma'), \Delta \quad \Gamma \Rightarrow \sigma : \phi, \Delta}{\Gamma \Rightarrow (\mathit{while}\ \phi\ \mathit{do}\ \alpha\ \mathit{end}, \sigma) \longrightarrow (\mathit{while}\ \phi\ \mathit{do}\ \alpha\ \mathit{end}, \sigma'), \Delta}\ (wh1\downarrow)$	$\dfrac{\Gamma \Rightarrow \sigma : \neg\phi, \Delta}{\Gamma \Rightarrow (\mathit{while}\ \phi\ \mathit{do}\ \alpha\ \mathit{end}, \sigma) \longrightarrow (\downarrow, \sigma), \Delta}\ (wh2)$

Table 1. Partial Inference Rules for Program Transitions of While Programs

$\dfrac{\{\Gamma \Rightarrow \sigma' : [\alpha']\phi, \Delta\}_{(\alpha', \sigma') \in \Phi}}{\Gamma \Rightarrow \sigma : [\alpha]\phi, \Delta}\ ^{1}\ ([\alpha]R)$, where $\Phi =_{df} \{(\alpha', \sigma') \mid P_{dlp} \vdash (\Gamma \Rightarrow \sigma \xrightarrow{\alpha/\alpha'} \sigma', \Delta)\}$			
$\dfrac{\Gamma, \sigma' : [\alpha']\phi \Rightarrow \Delta}{\Gamma, \sigma : [\alpha]\phi \Rightarrow \Delta}\ ^{1}\ ([\alpha]L)$, if $P_{dlp} \vdash (\Gamma \Rightarrow \sigma \xrightarrow{\alpha/\alpha'} \sigma', \Delta)$			
$\dfrac{\sigma : \phi}{\sigma : [\downarrow]\phi}\ ([\downarrow])$ (double line)	$\dfrac{}{\Gamma \Rightarrow \Delta}\ ^{2}\ (Ter)$	$\dfrac{\Gamma \Rightarrow \Delta}{\mathit{Sub}(\Gamma) \Rightarrow \mathit{Sub}(\Delta)}\ ^{3}\ (Sub)$	$\dfrac{}{\Gamma, \sigma : \phi \Rightarrow \sigma : \phi, \Delta}\ (ax)$
$\dfrac{\Gamma \Rightarrow \sigma : \phi, \Delta \quad \Gamma, \sigma : \phi \Rightarrow \Delta}{\Gamma \Rightarrow \Delta}\ (Cut)$	$\dfrac{\Gamma \Rightarrow \Delta}{\Gamma \Rightarrow \sigma : \phi, \Delta}\ (WkR)$	$\dfrac{\Gamma \Rightarrow \Delta}{\Gamma, \sigma : \phi \Rightarrow \Delta}\ (WkL)$	$\dfrac{\sigma : \phi, \sigma : \phi}{\sigma : \phi}\ (Con)$ (double line)
$\dfrac{\Gamma, \sigma : \phi \Rightarrow \Delta}{\Gamma \Rightarrow \sigma : \neg\phi, \Delta}\ (\neg R)$	$\dfrac{\Gamma \Rightarrow \sigma : \phi, \Delta}{\Gamma, \sigma : \neg\phi \Rightarrow \Delta}\ (\neg L)$	$\dfrac{\Gamma \Rightarrow \sigma : \phi, \Delta \quad \Gamma \Rightarrow \sigma : \psi, \Delta}{\Gamma \Rightarrow \sigma : \phi \wedge \psi, \Delta}\ (\wedge R)$	$\dfrac{\Gamma, \sigma : \phi, \sigma : \psi \Rightarrow \Delta}{\Gamma, \sigma : \phi \wedge \psi \Rightarrow \Delta}\ (\wedge L)$
[1] $\alpha \notin \{\downarrow\}$. [2] for each $\sigma : \phi \in \Gamma \cup \Delta$, $\phi \in \mathbf{F}$; Sequent $\Gamma \Rightarrow \Delta$ is valid. [3] *Sub* is given by Definition 7.			

Table 2. Rules P_{ldlp} for the Proof System of DL_p

Definition 6 (Assumptions on P_{dlp}). *System P_{dlp} is assumed to satisfy the following properties:*

1. *Soundness of P_{pt} and P_{ter}. All rules of P_{pt} and P_{ter} are sound.*
2. *Completeness w.r.t. K. For any $\sigma \in \mathbf{L}$, Γ and $\mathfrak{m} \in \mathbf{M}$ with $\mathfrak{m} \models \Gamma$, if $\mathfrak{m}(\sigma) \xrightarrow{\alpha/\alpha'} w$ is a relation on K for some $\alpha, \alpha' \in \mathbf{P}$ and $w \in \mathcal{S}$, then there exists a label $\sigma' \in \mathbf{L}$ such that $\mathfrak{m}(\sigma') = w$ and $P_{dlp} \vdash (\Gamma \Rightarrow \sigma \xrightarrow{\alpha/\alpha'} \sigma')$.*

Example 5 (Instantiation of $\mathfrak{F}_{pt}$). Table 1 displays a set $(\mathfrak{F}_{pt})_W$ of partial inference rules for the transitions of while programs. Here σ^x_e represents a configuration that stores variable x as value e, while storing other variables as the same value as σ.

Through the rules in P_{ldlp}, a labelled DL_p formula can be transformed into proof obligations as non-dynamic formulas, which can then be encoded and verified accordingly through, for example, an SAT/SMT checking procedure. The rules for other operators like $\vee$, $\rightarrow$ can be derived accordingly using the rules in Table 2.

The illustration of each rule in Table 2 is as follows. We use a double-lined inference form: $\dfrac{\phi_1 \quad ... \quad \phi_n}{\phi}$ to represent both rules $\dfrac{\Gamma \Rightarrow \phi_1, \Delta \quad ... \quad \Gamma \Rightarrow \phi_n, \Delta}{\Gamma \Rightarrow \phi, \Delta}$ and $\dfrac{\Gamma, \phi_1 \Rightarrow \Delta \quad ... \quad \Gamma, \phi_n \Rightarrow \Delta}{\Gamma, \phi \Rightarrow \Delta}$, provided any context Γ and Δ.

Rules ($[\alpha]R$) and ($[\alpha]L$) reason about dynamic parts of labelled DL_p formulas. Both rules rely on side deductions: '$P_{dlp} \vdash (\Gamma \Rightarrow \sigma \xrightarrow{\alpha/\alpha'} \sigma', \Delta)$' as sub-proof procedures of program transitions. In rule ($[\alpha]R$), $\{...\}_{(\alpha',\sigma')\in\Phi}$ represents the collection of premises for all program states $(\alpha', \sigma') \in \Phi$. By the finiteness of system P_{dlp}, set Φ must be finite (because only a finite number of forms (α', σ') can be derived). So rule ($[\alpha]R$) only has a finite number of premises. When Φ is empty, the conclusion terminates. Compared to rule ($[\alpha]R$), rule ($[\alpha]L$) has only one premise for some program state (α', σ').

Rule ($[\downarrow]$) deals with the situation when the program is a termination $\downarrow$. Its soundness is straightforward by the semantics of $\downarrow$ in Definition 2.

Rule (*Ter*) indicates that one proof branch terminates when all labelled formulas do not contain any dynamic parts.

Rule (*Sub*) describes a specialization process for labelled DL_p formulas. For a set A of labelled formulas, $Sub(A) =_{df} \{Sub(\sigma) : \phi \mid (\sigma : \phi) \in A\}$, with *Sub* an abstract notion of substitution defined as follows in Definition 7. Intuitively, if sequent $\Gamma \Rightarrow \Delta$ is valid, then its one of special cases $Sub(\Gamma) \Rightarrow Sub(\Delta)$ is also valid. Rule (*Sub*) plays an important role in constructing a bud in a cyclic proof structure (Section 3.3). See Section 4 for more details.

Definition 7 (Substitution of Labels). *A 'substitution' $\eta : \mathbf{L} \to \mathbf{L}$ is a function on $\mathbf{L}$ satisfying that for any label mapping $\mathfrak{m} \in \mathbf{M}$, there exists a label mapping $\mathfrak{m}'(\mathfrak{m}, \eta)$ (determined only by $\mathfrak{m}$ and η) such that $\mathfrak{m}'(\sigma) = \mathfrak{m}(\eta(\sigma))$ for all labels $\sigma \in \mathbf{L}$.*

Definition 7 will be used in the proof of soundness of rules P_{ldlp} and the cyclic proof system of DL_p.

Rules from (*ax*) to ($\wedge L$) are the "labelled verions" of the corresponding rules inherited from traditional first-order logic. Their meanings are classical and we omit their discussions here.

Theorem 1. *Each rule from P_{ldlp} in Table 2 is sound.*

Following the above explanations, Theorem 1 can be proved according to the semantics of labelled DL_p formulas under the assumption of Definition 6. See Appendix A of [1] for more details.

3.3 Construction of a Cyclic Proof Structure for DL_p

In an ordinary proof system we usually expect a finite proof tree. However, in system P_{dlp}, a branch of a proof tree does not always terminate, because the process of symbolically executing a program via rule ($[\alpha]R$) or/and rule ($[\alpha]L$)

might not stop. This is well-known when a program has an explicit/implicit loop structure that may run infinitely. For example, a while program $W =_{df}$ *while true do* $x := x + 1$ *end* will proceed infinitely as the following program transitions: $(W, \{x \mapsto 0\}) \longrightarrow (W, \{x \mapsto 1\}) \longrightarrow ....$

In this paper, we build a cyclic labelled proof system for DL_p, in order to recognize and admit potential infinite derivations as above when deriving labelled DL_p formulas $\mathfrak{F}_{ldlp}$ using the rules in P_{ldlp}. Based on the notion of preproof (Section 3.1), we build a cyclic proof structure for system P_{dlp}, where the key part is to introduce the notion of progressive derivation traces in DL_p (Definition 9). Section 5 will further show that a cyclic proof of DL_p ensures a valid conclusion.

Next we first introduce the notion of progressive derivation traces for DL_p, then we define the cyclic proof structure for DL_p, as a special case of the notion already given in Section 3.1.

Definition 8 (Derivation Traces). *A "derivation trace" over a derivation path* $\mu_1\mu_2...\mu_k\nu_1\nu_2...\nu_m...$ *(*$k \geq 0, m \geq 1$*) is an infinite sequence* $\tau_1\tau_2...\tau_m...$ *of formulas with each formula* τ_i *(*$1 \leq i \leq m$*) in node* ν_i*. Each CP pair* (τ_i, τ_{i+1}) *(*$i \geq 1$*) of derivation* (ν_i, ν_{i+1}) *satisfies special conditions as follows according to* (ν_i, ν_{i+1}) *being the different instances of rules from* P_{ldlp}*:*

1. *If* (ν_i, ν_{i+1}) *is an instance of rule* $([\alpha]R)$, $([\alpha]L)$, $([\downarrow])$, $(\neg R)$, $(\neg L)$, $(\wedge R)$ *or* $(\wedge L)$*, then either* τ_i *is the target formula and* τ_{i+1} *is its replacement by application of the rule, or* $\tau_i = \tau_{i+1}$*;*
2. *If* (ν_i, ν_{i+1}) *is an instance of rule* (Sub)*, then* $\tau_i = Sub(\sigma) : \phi$ *and* $\tau_{i+1} = \sigma : \phi$ *for some* $\sigma \in \mathbf{L}$ *and* $\phi \in \mathfrak{F}_{dlp}$*;*
3. *If* (ν_i, ν_{i+1}) *is an instance of other rules, then* $\tau_i = \tau_{i+1}$*.*

Definition 9 (Progressive Derivation Traces). *In a preproof of system* P_{dlp}*, given a derivation trace* $\tau_1\tau_2...\tau_m...$ *over a derivation path* $...\nu_1\nu_2...\nu_m...$ *(*$m \geq 1$*) starting from* τ_1 *in node* ν_1*, a CP pair* (τ_i, τ_{i+1}) *(*$1 \leq i \leq m$*) of derivation* (ν_i, ν_{i+1}) *is called a "progressive step", if* (τ_i, τ_{i+1}) *is the following CP pair of an instance of rule* $([\alpha]R)$*:*

$$\frac{... \quad \nu_{i+1} :: (\Gamma \Rightarrow \tau_{i+1} :: (\sigma' : [\alpha']\phi), \Delta) \quad ...}{\nu_i :: (\Gamma \Rightarrow \tau_i :: (\sigma : [\alpha]\phi), \Delta),} \,{\scriptstyle ([\alpha]R)};$$

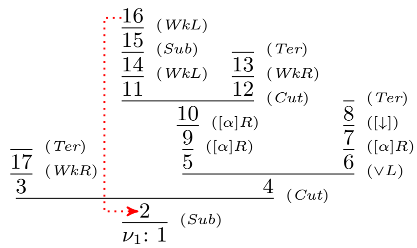
or the following CP pair of an instance of rule $([\alpha]L)$*:*

$$\frac{\nu_{i+1} :: (\Gamma, \tau_{i+1} :: (\sigma' : [\alpha']\phi) \Rightarrow \Delta)}{\nu_i :: (\Gamma, \tau_i :: (\sigma : [\alpha]\phi) \Rightarrow \Delta)} \,{\scriptstyle ([\alpha]L)},$$

provided with an additional side deduction $P_{dlp} \vdash (\Gamma \Rightarrow \sigma' \Downarrow \alpha', \Delta)$*.*

If a derivation trace has an infinite number of progressive steps, we say that the trace is 'progressive'.

The additional side condition of the instance of rule $([\alpha]L)$ is the key to prove the corresponding case in Lemma 1 (see Appendix A of [1]).



Definitions of other symbols:
$WP =_{df} \{while\ (n > 0)\ do\ s := s + n\ ;\ n := n - 1\ end\ \}$
$\alpha_1 =_{df} s := s + n\ ;\ n := n - 1$
$\phi_1 =_{df} (s = ((N + 1)N)/2)$
$\sigma_1 =_{df} \{n \mapsto N, s \mapsto 0\}$
$\sigma_2 =_{df} \{n \mapsto N - m, s \mapsto (2N - m + 1)m/2\}$
$\sigma_3 =_{df} \{n \mapsto N - m, s \mapsto (2N - (m + 1) + 1)(m + 1)/2\}$
$\sigma_4 =_{df} \{n \mapsto N - (m + 1), s \mapsto (2N - (m + 1) + 1)(m + 1)/2\}$

1:	$\sigma_1 : n \geq 0$	$\Rightarrow$	$\sigma_1 : [while\ (n > 0)\ do\ \alpha_1\ end\ ]\phi_1$
2:	$\sigma_2 : n \geq 0$	$\Rightarrow$	$\sigma_2 : [while\ (n > 0)\ do\ \alpha_1\ end\ ]\phi_1$
3:	$\sigma_2 : n \geq 0$	$\Rightarrow$	$\sigma_2 : [while\ (n > 0)\ do\ \alpha_1\ end\ ]\phi_1, \sigma_2 : (n > 0 \vee n \leq 0)$
17:	$\sigma_2 : n \geq 0$	$\Rightarrow$	$\sigma_2 : (n > 0 \vee n \leq 0)$
4:	$\sigma_2 : n \geq 0, \sigma_2 : (n > 0 \vee n \leq 0)$	$\Rightarrow$	$\sigma_2 : [while\ (n > 0)\ do\ \alpha_1\ end\ ]\phi_1$
5:	$\sigma_2 : n \geq 0, \sigma_2 : n > 0$	$\Rightarrow$	$\sigma_2 : [while\ (n > 0)\ do\ \alpha_1\ end\ ]\phi_1$
9:	$\sigma_2 : n \geq 0, \sigma_2 : n > 0$	$\Rightarrow$	$\sigma_3 : [n := n - 1;\ while\ (n > 0)\ do\ \alpha_1\ end\ ]\phi_1$
10:	$\sigma_2 : n \geq 0, \sigma_2 : n > 0$	$\Rightarrow$	$\sigma_4 : [while\ (n > 0)\ do\ \alpha_1\ end\ ]\phi_1$
11:	$\sigma_2 : n \geq 0, \sigma_2 : n > 0, \sigma_4 : n \geq -1, \sigma_4 : n \geq 0$	$\Rightarrow$	$\sigma_4 : [while\ (n > 0)\ do\ \alpha_1\ end\ ]\phi_1$
14:	$\sigma_4 : n \geq -1, \sigma_4 : n \geq 0$	$\Rightarrow$	$\sigma_4 : [while\ (n > 0)\ do\ \alpha_1\ end\ ]\phi_1$
15:	$\sigma_2 : n \geq -1, \sigma_2 : n \geq 0$	$\Rightarrow$	$\sigma_2 : [while\ (n > 0)\ do\ \alpha_1\ end\ ]\phi_1$
16:	$\sigma_2 : n \geq 0$	$\Rightarrow$	$\sigma_2 : [while\ (n > 0)\ do\ \alpha_1\ end\ ]\phi_1$
12:	$\sigma_2 : n \geq 0, \sigma_2 : n > 0$	$\Rightarrow$	$\sigma_4 : [while\ (n > 0)\ do\ \alpha_1\ end\ ]\phi_1, \sigma_4 : n \geq -1, \sigma_4 : n \geq 0$
13:	$\sigma_2 : n \geq 0, \sigma_2 : n > 0$	$\Rightarrow$	$\sigma_4 : n \geq -1, \sigma_4 : n \geq 0$
6:	$\sigma_2 : n \geq 0, \sigma_2 : n \leq 0$	$\Rightarrow$	$\sigma_2 : [while\ (n > 0)\ do\ \alpha_1\ end\ ]\phi_1$
7:	$\sigma_2 : n \geq 0, \sigma_2 : n \leq 0$	$\Rightarrow$	$\sigma_2 : [\downarrow]\phi_1$
8:	$\sigma_2 : n \geq 0, \sigma_2 : n \leq 0$	$\Rightarrow$	$\sigma_2 : (s = ((N + 1)N)/2)$

Table 3. Derivations of Property ν_1

Theorem 2. *Every cyclic proof of system P_{dlp} has a sound conclusion.*

In Section 5, we will analyze and prove Theorem 2 under a restriction on **P**.

Since DL_p is not a specific logic, it is impossible to discuss about its decidability, completeness or whether it is cut-free without any restrictions on parameters **P**, **F** and **L**. One of our future work will focus on analyzing under what restrictions, these properties can be obtained in a general sense.

4 Case Study — A Cyclic Deduction for While Programs

We give an example to show how a DL_p formula can be derived according to rules in Table 2. We prove the property in Example 2, which can be captured by the following equivalent labelled sequent

$$\nu_1 =_{df} \sigma_1 : n \geq 0 \Rightarrow \sigma_1 : [WP](s = ((N + 1)N)/2),$$

where $\sigma_1 =_{df} \{n \mapsto N, s \mapsto 0\}$, describing the initial configuration of WP.

Table 3 shows its derivations. We omit all side deductions as sub-proof procedures in instances of rule ($[\alpha]R$) derived using the inference rules in Table 1. Non-primitive rule ($\vee L$) can be derived by the rules for $\neg$ and $\wedge$ accordingly.

The derivation from sequent 1 to 2 is according to rule (*Sub*), where the substitution *Sub* of labels defined in Definition 7 is instantiated as function $(\cdot)[e/x]$. Informally, for any label σ, $\sigma[e/x]$ returns the label by substituting each free variable x of σ with term e. We observe that $\sigma_1 = \sigma_2[0/m]$, so sequent 1 is a special case of sequent 2 by substitution $(\cdot)[0/m]$. Intuitively, label σ_2 captures the program configuration after the mth loop ($m \geq 0$) of program *WP*. This step is crucial as starting from sequent 2, we can find a bud node — 16 — that is identical to node 2.

The derivation from sequent 2 to $\{3, 4\}$ provides a lemma: $\sigma_2 : (n > 0 \vee n \leq 0)$, which is trivially valid. Sequent 16 indicates the end of the $(m+1)$th loop of program *WP*. From node 10 to 16, we transform the formulas on the left side into a trivial logical equivalent form in order to apply rule (*Sub*) from sequent 14 to 15. Sequent 14 is a special case of sequent 15 since $\sigma_4 = \sigma_2[m+1/m]$.

The whole proof tree is cyclic because the only derivation path: $2, 4, 5, 9, 10, 11, 14, 15, 16, 2, ...$ has a progressive derivation trace whose elements are underlined in Table 3.

Compared to the deduction processes in traditional dynamic logics and Hoare logics, a notable feature of the above deduction process is that the search for a loop invariant is reflected in looking for a suitable configuration (i.e. σ_2). One advantage brought by this cyclic derivation approach is that it does not rely on the inference rule for decomposing an explicit loop structure (here *while...do...end*), which also makes it easily amendable for reasoning about programs with implicit loop structures, such as CCS-like process algebras [29,30] and some synchronous languages [8,44].

As a demonstration of its powerfulness, in Appendix B of [1], we briefly introduce another instantiation of DL_p for the synchronous language Esterel [8] and show a cyclic derivation of an Esterel program. That example can better highlight the advantages of DL_p since the loop structures of some Esterel programs are implicit. In [1], we also briefly show that FODL [38] can be instantiated in DL_p.

5 Soundness of Cyclic Proof System P_{dlp}

In this section, we prove Theorem 2 under a restriction on $\mathbf{P}$ given as in Definition 10.

An execution path (Definition 3) $w_1...w_n$ ($n \geq 1$) is called *minimum*, if there are no two relations $w_i \xrightarrow{\alpha_i/\cdot} \cdot$ and $w_j \xrightarrow{\alpha_j/\cdot} \cdot$ for some $1 \leq i < j < n$ such that $w_i = w_j$ and $\alpha_i = \alpha_j$. Intuitively, in a minimum execution path, there are no two relations starting from the same world and program.

Definition 10 (Termination Finiteness). *In PL Kripke structure K, a program $\alpha \in \mathbf{P}$ satisfies the "termination finiteness" property, if for a world $w \in \mathcal{S}$,* *there is only a finite number of minimum execution paths starting from a relation of the form* $w \xrightarrow{\alpha/\cdot} \cdot$.

The programs satisfying termination finiteness are in fact a rich set, including, for example, all the programs whose behaviour is deterministic, such as while programs discussed in this paper, programming languages like Esterel, C, Java, etc. There exist non-deterministic programs that fall into this category. For example, automata that have non-deterministic transitions and a finite number of states. More on this restriction will be discussed in our future work.

Main Idea. We follow the main idea behind [10] to prove Theorem 2 by contradiction. The key point is that, if the conclusion of a cyclic proof is invalid, then there must exist an *invalid derivation path* in which each node is invalid, and one of its progressive traces would lead to an infinite descent sequence of some well-founded set (introduced below), which violates the definition of the well-foundedness (cf. [13]) itself.

Below we firstly introduce the well-founded relation $\prec_m$ we will rely on, then we focus on the main skeleton of proving Theorem 2. Other proof details are given in Appendix A of the online report [1].

Well-foundedness & Relation $\preceq_m$. Given a set S and a partial-order relation $\preceq$ on S, $\preceq$ is called a *well-founded relation* over S, if for any element a in S, there is no infinite descent sequence: $a \succ a_1 \succ a_2 \succ ...$ in S. Set S is called a *well-founded* set w.r.t. $\preceq$.

Definition 11 (Relation $\preceq_m$). *Given two finite sets C_1 and C_2 of finite execution paths, $C_1 \preceq_m C_2$ is defined if either (1) $C_1 = C_2$; or (2) set C_1 can be obtained from C_2 by replacing (or removing) one or more elements of C_2 each with a finite number of elements, such that for each replaced element tr, its replacements $tr_1, ..., tr_n$ ($n \geq 1$) in C_1 are proper suffixes of tr.*

$\preceq_m$ is a partial-order relation. The proof is given in Appendix A of [1].

Example 6. Let $C_1 = \{tr_1, tr_2, tr_3\}$, where $tr_1 =_{df} ww_1w_2w_3w_4, tr_2 =_{df} ww_1w_5w_6w_7$ and $tr_3 =_{df} ww_8$; $C_2 = \{tr'_1, tr'_2\}$, where $tr'_1 =_{df} w_1w_2w_3w_4, tr'_2 =_{df} w_1w_5w_6w_7$. We see that tr'_1 is a proper suffix of tr_1 and tr'_2 is a proper suffix of tr_2. C_2 can be obtained from C_1 by replacing tr_1 and tr_2 with tr'_1 and tr'_2 respectively, and removing tr_3. Hence $C_2 \preceq_m C_1$. Since $C_1 \neq C_2$, $C_2 \prec_m C_1$.

Proposition 1. *Relation $\preceq_m$ is a well-founded relation.*

We omit the proof of Proposition 1. Relation $\preceq_m$ is just a special case of the "multi-set ordering" introduced in [13], where it has been proved to be well-founded.

Proof Skeleton of Theorem 2. Below we give the main skeleton of the proof by skipping the details of the proof of Lemma 1, which can be found in Appendix A of [1].

Following the main idea above, we first introduce the concept of "execution paths of a dynamic DL_p formula". They are the elements of a well-founded relation $\preceq_m$. Next, we propose Lemma 1, which plays a key role in the proof of Theorem 2 that follows.

Given two execution paths $w_1...w_n$ and $w'_1w'_2...w'_m$ $(n, m \geq 1)$, *path concatenation* $\cdot$ is a partial function defined such that $(w_1...w_n)\cdot(w'_1w'_2...w'_m) =_{df} w_1...w_nw'_2...w'_m$, if $w_n = w'_1$.

Definition 12 (Execution Paths of Dynamic Formulas). *Given a world $w \in \mathcal{S}$ and a dynamic formula ϕ, the execution paths $EX(w, \phi)$ of ϕ w.r.t. w is inductively defined according to the structure of ϕ as follows:*

1. $EX(w, [\alpha]F) =_{df} mex(w, \alpha)$, *where* $F \in \mathbf{F}$*;*
2. $EX(w, [\alpha]\phi_1) =_{df} mex(w, \alpha) \cup \{tr_1 \cdot tr_2 \mid tr_1 \in mex(w, \alpha), tr_2 \in EX((tr_1)_e, \phi_1)\}$*;*
3. $EX(w, \neg\phi_1) =_{df} EX(w, \phi_1)$*;*
4. $EX(w, \phi_1 \wedge \phi_2) =_{df} EX(w, \phi_1) \cup EX(w, \phi_2)$.

Where $mex(w, \alpha) =_{df} \{w...w' \mid w \xrightarrow{\alpha/\cdot} ... \xrightarrow{\cdot/\downarrow} w'$ *is a min. exec. path for some* $w' \in \mathcal{S}\}$ *is the set of all minimum paths of* α *starting from world* w.

We call $\mathfrak{m} \in \mathbf{M}$ a *counter-example mapping* of a node ν, if it makes ν invalid.

Lemma 1. *In a cyclic proof (where there is at least one derivation path), let $(\sigma : \phi, \sigma' : \phi')$ be a step of a derivation trace over a derivation (ν, ν') of an invalid derivation path, where $\phi, \phi' \in \mathfrak{F}_{dlp}$. For any set $EX(\mathfrak{m}(\sigma), \phi)$ of $\sigma : \phi$ w.r.t. a counter-example mapping $\mathfrak{m}$ of ν, there exists a counter-example mapping $\mathfrak{m}'$ of ν' and a set $EX(\mathfrak{m}'(\sigma'), \phi')$ of $\sigma' : \phi'$ such that $EX(\mathfrak{m}'(\sigma'), \phi') \preceq_m EX(\mathfrak{m}(\sigma), \phi)$. Moreover, if $(\sigma : \phi, \sigma' : \phi')$ is a progressive step, then $EX(\mathfrak{m}'(\sigma'), \phi') \prec_m EX(\mathfrak{m}(\sigma), \phi)$.*

Intuitively, Lemma 1 helps us discover suitable execution-path sets imposed by a well-founded relation $\preceq_m$ between them in an invalid derivation path.

Based on Proposition 1 and Lemma 1, we give the proof of Theorem 2 as follows.

Proof (Proof of Theorem 2). By contradiction. Let the progressive trace be $\tau_1\tau_2...\tau_k...$ over a derivation path $...\nu_1\nu_2...\nu_k...$ $(k \geq 1)$ starting from τ_1 in ν_1, where $\tau_i =_{df} \sigma_i : \phi_i$ $(i \geq 1)$, each ν_i $(i \geq 1)$ is invalid.

Since ν_1 is invalid, let $\mathfrak{m}_1$ be one of its counter-example mappings. By Lemma 1, from $EX(\mathfrak{m}_1(\sigma_1), \phi_1)$, there exists an infinite sequence of sets $EX_1, ..., EX_k, ...$ $(k \geq 1)$, where each $EX_i =_{df} EX(\mathfrak{m}_i(\sigma_i), \phi_i)$ $(i \geq 1)$ with $\mathfrak{m}_i$ a counter-example mapping of node ν_i, and which satisfies that $EX_1 \succeq_m ... \succeq_m EX_k \succeq_m ...$. Moreover, since trace $\tau_1\tau_2...\tau_k...$ is progressive (Definition 9), there must be an infinite number of $j \geq 1$ such that $EX_j \succ_m EX_{j+1}$. This thus forms an infinite descent sequence w.r.t. $\prec_m$, violating the well-foundedness of relation $\preceq_m$ (Proposition 1).

6 Related Work

The idea of reasoning about programs based on their operational semantics is not new. Previous work such as [41,42,45,12] in the last decade has addressed this issue using theories based on rewriting logic [28]. Matching logic [41] is based on patterns and pattern matching. Its basic form, a reachability rule $\varphi \Rightarrow \varphi'$ (where $\Rightarrow$ has another meaning from its use in this paper), captures whether pattern φ' is reachable from pattern φ in a given pattern reachability system. Based on matching logic, one-path and all-paths reachability logics [42,45] were developed by enhancing the expressive power of the reachability rule. A more powerful matching μ-logic [12] was proposed by adding a least fixpoint μ-binder to matching logic.

In these theories, 'patterns' are more general structures. So to encode the dynamic forms $[\alpha]\phi$ in DL_p requires additional work and program transformations. On the other hand, dynamic logics like DL_p provide more direct ways to express and reason about complex before-after and temporal program properties with their modalities $[\cdot]$ and $\langle\cdot\rangle$. In terms of expressiveness, matching logic and one-path reachability logic cannot capture the semantics of modality $[\cdot]$ when the programs are non-deterministic (which means that there are more than one execution path). We conjecture that matching μ-logic can encode DL_p, as it has been claimed that it can encode traditional dynamic logics (cf. [12]).

[31] proposed a general program verification framework based on coinduction. Using the terminology in this paper, a program specification $\sigma : [\alpha]\phi$ can be expressed as a pair $((\alpha, \sigma), P(\phi))$ in [31], with $P(\phi)$ a set of program states capturing the semantics of formula ϕ. A method was designed to derive a program specification in a coinductive way according to the operational semantics of (α, σ). Following [31], [27] also proposed a general framework for program reasoning, but via big-step operational semantics. Unlike the frameworks in [31] and [27] which are directly built up on mathematical set theory, DL_p is in logical forms, and is based on a cyclic deduction approach rather than coinduction. In terms of expressiveness, the meaning of modality $\langle\cdot\rangle$ in DL_p cannot be expressed in the framework of [31].

The structure 'updates' adopted in work [35,4,5] are "delay substitutions" of variables and terms. They in fact can be defined as a special case of the more general labels in DL_p by choosing suitable label mappings accordingly.

The proof system of DL_p relies on the cyclic proof theory which firstly arose in [46] and was later developed in different logics such as [11] and [10]. [25] proposed a complete cyclic proof system for μ-calculus, which subsumes PDL [16] in its expressiveness. In [15], the authors proposed a complete labelled cyclic proof system for PDL. Both logics in [25,15] are propositional and cannot be used to prove many valid formulas in particular domains, for example, the arithmetic first-order formulas in number theory as shown in our example. The labelled form of DL_p formula $\sigma : [\alpha]\phi$ is inspired from [15], where a label is just a variable of worlds in a traditional Kripke structure. On the other hand, the labels in DL_p allow arbitrary terms from actual program configurations.

There has been some other work for generalizing the theories of dynamic logics, such as [32,21]. However, generally speaking, they neither consider structures as general as allowed by programs and configurations in DL_p, nor adopt a similar approach for reasoning about programs. [32] proposed a complete generic dynamic logic for monads of programming languages. In the dynamic logic proposed in [21], more general programs can be reasoned about than regular expressions of PDL, based on so-called "interaction-based" behaviours. But there program transitions are captured by abstract actions in a form e.g. $\alpha \xrightarrow{a} \beta$, where no explicit structures of program configurations are allowed. And yet no proof systems have been built for that logic.

7 Conclusion & Future Work

In this paper, we propose a novel dynamic logic DL_p that supports reasoning about general forms of programs and formulas based on programs' operational semantics. We mainly build the theory of DL_p and propose a sound cyclic proof system of DL_p that is proved to be useful and applicable. As the main theoretical result we prove the soundness of DL_p.

On theoretical aspects, we are interested in obtaining a complete proof system by fixing the parameters of DL_p in some algebraic domains, e.g., a multi-sorted signature for imperative programs as in [18]. We also want to analyze whether our framework can be adapted to a wider range of program behaviours, i.e., to relax the property Definition 10. From an applied perspective, we are carrying out a full mechanization of DL_p in Coq [9]. To see the full potential of DL_p, we will try to instantiate more types of programs or system models in DL_p, and to specify and verify their properties using DL_p formulas.

References

1. `https://github.com/yrz5a/setta2025-onlineReport.git`
2. Afshari, B., Enqvist, S., Leigh, G.E.: Cyclic proofs for the first-order µ-calculus. Logic Journal of the IGPL **32**(1), 1–34 (08 2022)
3. Appel, A.W., Dockins, R., et al.: Program Logics for Certified Compilers. Cambridge University Press (2014)
4. Beckert, B., Bruns, D.: Dynamic logic with trace semantics. In: CADE 2013. pp. 315–329. Springer Berlin Heidelberg (2013)
5. Beckert, B., Klebanov, V., Weiß, B.: Dynamic Logic for Java, pp. 49–106. Springer International Publishing (2016)
6. Benevides, M.R., Schechter, L.M.: A propositional dynamic logic for concurrent programs based on the π-calculus. In: M4M 2009. pp. 49–64. Elsevier (2010)
7. Berry, G., Cosserat, L.: The esterel synchronous programming language and its mathematical semantics. In: Seminar on Concurrency. pp. 389–448. Springer Berlin Heidelberg, Berlin, Heidelberg (1985)
8. Berry, G., Gonthier, G.: The Esterel synchronous programming language: design, semantics, implementation. Science of Computer Programming **19**(2), 87 – 152 (1992)

9. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series, Springer (2004)
10. Brotherston, J., Bornat, R., Calcagno, C.: Cyclic proofs of program termination in separation logic. SIGPLAN Not. **43**(1), 101–112 (2008)
11. Brotherston, J., Simpson, A.: Complete sequent calculi for induction and infinite descent. In: LICS 2007. pp. 51–62 (2007)
12. Chen, X., Rosu, G.: Matching *mu*-logic. In: LICS 2019. pp. 1–13. IEEE Computer Society (jun 2019)
13. Dershowitz, N., Manna, Z.: Proving termination with multiset orderings. In: Automata, Languages and Programming. pp. 188–202. Springer Berlin Heidelberg, Berlin, Heidelberg (1979)
14. Do, C.M., Takagi, T., Ogata, K.: Automated quantum protocol verification based on concurrent dynamic quantum logic. ACM Trans. Softw. Eng. Methodol. (Dec 2024), just Accepted
15. Docherty, S., Rowe, R.N.S.: A non-wellfounded, labelled proof system for propositional dynamic logic. In: TABLEAUX 2019. pp. 335–352. Springer International Publishing (2019)
16. Fischer, M.J., Ladner, R.E.: Propositional dynamic logic of regular programs. Journal of Computer and System Sciences **18**(2), 194–211 (1979)
17. Gesell, M., Schneider, K.: A Hoare calculus for the verification of synchronous languages. In: PLPV 2012. p. 37–48. Association for Computing Machinery (2012)
18. Goguen, J.A., Malcolm, G.: Algebraic Semantics of Imperative Programs. The MIT Press (05 1996)
19. Gutsfeld, J.O., Müller-Olm, M., Ohrem, C.: Propositional Dynamic Logic for Hyperproperties. In: 31st International Conference on Concurrency Theory (CONCUR). pp. 50:1–50:22 (2020)
20. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press (2000)
21. Hennicker, R., Wirsing, M.: A Generic Dynamic Logic with Applications to Interaction-Based Systems, pp. 172–187. Springer International Publishing (2019)
22. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
23. Icard, B.: A dynamic logic for information evaluation in intelligence (2024), `https://arxiv.org/abs/2405.19968`
24. Jones, E., Ong, C.H.L., Ramsay, S.: Cycleq: an efficient basis for cyclic equational reasoning. In: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. p. 395–409. PLDI 2022, Association for Computing Machinery, New York, NY, USA (2022)
25. Jungteerapanich, N.: A tableau system for the modal μ-calculus. In: TABLEAUX 2009. pp. 220–234. Springer Berlin Heidelberg (2009)
26. Kozen, D.: A probabilistic pdl. Journal of Computer and System Sciences **30**(2), 162–178 (1985)
27. Li, X., Zhang, Q., Wang, G., Shi, Z., Guan, Y.: Reasoning about iteration and recursion uniformly based on big-step semantics. In: SETTA 2021. pp. 61–80. Springer International Publishing (2021)
28. Meseguer, J.: Twenty years of rewriting logic. The Journal of Logic and Algebraic Programming **81**(7), 721–781 (2012)
29. Milner, R.: A Calculus of Communicating Systems. Springer-Verlag, Berlin, Heidelberg (1982)
30. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, i. Information and Computation **100**(1), 1–40 (1992)

31. Moore, B., Peña, L., Rosu, G.: Program verification by coinduction. In: ESOP 2018. pp. 589–618. Springer International Publishing (2018)
32. Mossakowski, T., Schröder, L., Goncharov, S.: A generic complete dynamic logic for reasoning about purity and effects. Formal Aspects of Computing **22**(3-4), 363–384 (2010)
33. O'Hearn, P.W.: Incorrectness logic. Proc. ACM Program. Lang. **4**(POPL) (2019)
34. Pardo, R., Johnsen, E.B., et al.: A specification logic for programs in the probabilistic guarded command language. In: ICTAC 2022. pp. 369–387. Springer-Verlag (2022)
35. Platzer, A.: Differential dynamic logic for verifying parametric hybrid systems. In: TABLEAUX 2007. pp. 216–232. Springer Berlin Heidelberg (2007)
36. Platzer, A.: Logical Foundations of Cyber-Physical Systems. Springer (2018)
37. Platzer, A., Quesel, J.D.: Keymaera: A hybrid theorem prover for hybrid systems (system description). In: Automated Reasoning. pp. 171–178. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
38. Pratt, V.R.: Semantical considerations on Floyd-Hoare logic. In: Annual IEEE Symposium on Foundations of Computer Science (FOCS). pp. 109–121. IEEE Computer Society (1976)
39. Reif, W.: The Kiv-approach to software verification, pp. 339–368. Springer Berlin Heidelberg, Berlin, Heidelberg (1995)
40. Reynolds, J.: Separation logic: a logic for shared mutable data structures. In: Proceedings 17th Annual IEEE Symposium on Logic in Computer Science. pp. 55–74 (2002)
41. Roşu, G., Ştefănescu, A.: Towards a unified theory of operational and axiomatic semantics. In: ICALP 2012. pp. 351–363. Springer Berlin Heidelberg (2012)
42. Rosu, G., Stefanescu, A., Ciobâcá, S., Moore, B.M.: One-path reachability logic. In: LICS 2013. pp. 358–367 (2013)
43. Rustan, K., Leino, M.: Verification of Object-Oriented Software. The KeY Approach, Lecture Notes in Computer Science (LNCS), vol. 4334. Springer (2007)
44. Schneider, K., Brandt, J.: Quartz: A Synchronous Language for Model-Based Design of Reactive Embedded Systems, pp. 1–30. Springer Netherlands, Dordrecht (2017)
45. Ştefănescu, A., Ciobâcă, Ş., et al.: All-path reachability logic. In: RTA-TLCA 2014. pp. 425–440. Springer International Publishing (2014)
46. Stirling, C., Walker, D.: Local model checking in the modal mu-calculus. Theor. Comput. Sci. **89**(1), 161–177 (1991)
47. Takagi, T., Do, C.M., Ogata, K.: Automated quantum program verification in dynamic quantum logic. In: Dynamic Logic. New Trends and Applications. pp. 68–84. Springer Nature Switzerland, Cham (2024)
48. Tellez, G., Brotherston, J.: Automatically verifying temporal properties of pointer programs with cyclic proof. Journal of Automated Reasoning **64**(3), 555–578 (2020)
49. Zhang, Y., Mallet, F., Liu, Z.: A dynamic logic for verification of synchronous models based on theorem proving. Front. Comput. Sci. **16**(4) (2022)
50. Zhang, Y., Wu, H., Chen, Y., Mallet, F.: A clock-based dynamic logic for the verification of ccsl specifications in synchronous systems. Science of Computer Programming **203**, 102591 (2021)
51. Zilberstein, N., Dreyer, D., Silva, A.: Outcome logic: A unifying foundation for correctness and incorrectness reasoning. Proc. ACM Program. Lang. **7**(OOPSLA1) (apr 2023)