Parameterized Dynamic Logic — Towards A Cyclic Logical Framework for General Program Specification and Verification

Yuanrui Zhang[1 [0000−0002−0685−6905]]

Collage of Software, Nanjing University of Aeronautics and Astronautics, China
yuanruizhang@nuaa.edu.cn, zhangyrmath@126.com

Abstract. We present a theory of parameterized dynamic logic, namely DL_p, for specifying and reasoning about general program models based on their transitional behaviours. Different from most dynamic-logic theories that deal with regular expressions or a particular type of models, DL_p introduces a labelled system in which symbolic executions of general programs can be fully captured and derived. Our deduction approach is a cyclic one, tailored from previous work onto DL_p, in order to deal with potentially infinite derivations arose from symbolic executions of loop programs. The soundness of DL_p is formally analyzed and partially proved. We give an example of instantiations of DL_p in particular domains, showing the usage and potential of DL_p in practice. The cyclic proof system of DL_p is mechanized in Coq.

Keywords: Dynamic Logic · Program Deduction · Logical Framework · Cyclic Proof · Symbolic Execution.

1 Introduction

Background and Motivations. Dynamic logic [16] is an extension of modal logic for reasoning about programs. Here, the term 'program' can mean an abstract formal model, not necessarily an explicit computer program. In dynamic logic, a program α is embedded into a modality $[\cdot]$ in a form of $[\alpha]\phi$, meaning that after all executions of α, formula ϕ holds. Formula $\phi \to [\alpha]\psi$ exactly captures partial correctness of programs expressed by triple $\{\phi\}\alpha\{\psi\}$ in Hoare logic [18]. With a dynamic form $[\alpha]\phi$ that mixes both programs and formulas, dynamic logic allows multiple modalities in a single formula, allowing expressing many complex program properties such as $[\alpha]\langle\beta\rangle\phi$ and $[\alpha]\psi \to \langle\beta\rangle\phi$, which cannot be directly captured in traditional Hoare logic. As one of popular logical formalisms, dynamic logic has been applied for many types of programs, such as process algebras [5], programming languages [4], synchronous systems [35], hybrid systems [27] and probabilistic systems [25]. Because of modality $\langle\cdot\rangle$, it is a natural candidate for unifying correctness and 'incorrectness' reasoning recently proposed and developed in work e.g. [24].

The programs of dynamic logics are usually (tailored) regular expressions with tests (e.g. [11,27]) or actual programming languages like Java [4]. The

denotational semantics of these models is usually *compositional*, in the sense that the deductions of the logics mean to dissolve the syntactic structures of these models. For example, in propositional dynamic logic (PDL) [11], to prove a formula $[\alpha \cup \beta]\phi$, we prove both formulas $[\alpha]\phi$ and $[\beta]\phi$, where α and β are sub-regular-expressions of $\alpha \cup \beta$. This so-called 'divide-and-conquer' approach to program verification has brought numerous benefits, the most significant being its ability to scale verifications.

However, this typical we call *structure-based* reasoning heavily relies on programs' denotational semantics, thus brings two major drawbacks: (1) Firstly, the program models of a dynamic logic are usually in particular forms. Thus if applying the dynamic logic to a new type of programs, one needs to either perform a transformation from target programs into the programs of the dynamic logic so that existing inference rules can be utilized, or carefully design a set of particular rules to adapt the new programs' semantics. The former usually means losses of partial program structural information during the transformations, while the latter often demands a large amount of work. For example, Verifiable C [2], a famous program verifier for C programming language based on Hoare logic, used nearly 40,000 lines of Coq code to define the logic theory. (2) Secondly but importantly as well, some programs are naturally non-compositional or do not temporally provide a compositional semantics at some level of system design. Typical examples are neural networks [14], automata-based system models (e.g. a transition model of AADL [33]) and some programming languages (such as synchronous imperative programming languages Esterel [6] and Quartz [13]). For these models, one can only re-formalize their structures by additional transformation procedures so that structure-based reasoning is possible.

Recent years, there is a trend to unify different types of program models into a single logical framework by reasoning directly based on programs' transitional behaviours [29,30,31,9,22,20,17], which can in large scale compensate for the above two shortcomings. To our best knowledge, up to now there is only a few study (see Section 6 for a comparison) for dynamic-logic theories.

In this paper, we present a theory of parameterized dynamic logic (DL_p) to support a general approach for program verification in the framework based on dynamic logic. DL_p offers an abstract-leveled setting for describing a rich set of programs and formulas, and is empowered to support a *symbolic-execution-based* reasoning based on a set of program transitional behaviours (simply "program behaviours" below). This, on one hand, reduces the burden of developing different dynamic-logic theories for different programs; on the other, it saves the additional transformations in the derivations of non-compositional programs.

Illustration of Main Idea. Informally, in dynamic formula $[\alpha]\phi$ we treat program α and formula ϕ as 'parameters' that can be of arbitrary forms, and adopt a structure σ called "program configuration" to record current program status for programs' symbolic executions. So, a DL_p dynamic formula is of the form: $\sigma : [\alpha]\phi$, following a convention of labeled formulas [10].

To see how we can benefit from deriving a DL_p formula, consider a simple example. We prove a formula $\phi_1 =_{df} (x \geq 0 \rightarrow [x := x + 1]x > 0)$ in first-order

dynamic logic [15] (FODL), where x is a variable ranging over the set of integer numbers. Intuitively, ϕ_1 means that if $x \geq 0$ holds, then $x > 0$ holds after the execution of the assignment $x := x+1$. In FODL, to derive formula ϕ_1, we apply the structural rule: $\frac{\phi[x/e]}{[x := e]\phi}\ ^{(x:=e)}$ 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 $\phi_1' =_{df} (x \geq 0 \rightarrow x + 1 > 0)$, which is true for any integer number $x \in \mathbb{Z}$.

While in DL_p, formula ϕ_1 can be expressed as a form: $\psi_1 =_{df} (t \geq 0 \rightarrow \{x \mapsto t\} : [x := x+1]x > 0)$, where formula $[x := x+1]x > 0$ is labeled by configuration $\{x \mapsto t\}$, capturing the current program status with t a free variable, meaning "variable x has value t". With a configuration explicitly showing up, to derive formula ψ_1, we instead directly perform a program transition of $x := x + 1$ as: $(x := x + 1, \{x \mapsto t\}) \longrightarrow (\downarrow, \{x \mapsto t + 1\})$, which assign x's value with its current value added by 1. Here $\downarrow$ indicates a program termination. Formula ψ_1 thus becomes $\psi_1' =_{df} (t \geq 0 \rightarrow \{x \mapsto t + 1\} : x > 0)$, by ignoring the dynamic part '$[\downarrow]$' since $\downarrow$ behaves nothing. By applying configuration $\{x \mapsto t + 1\}$ on formula $x > 0$ (which means replacing variable x of $x > 0$ with its value $t+1$ in $\{x \mapsto t + 1\}$), we obtain the same valid formula ϕ_1': $t \geq 0 \rightarrow t + 1 > 0$ (modulo the free variables x, t).

For many programs, transitions like $(x := x+1, \{x \mapsto t\}) \longrightarrow (\downarrow, \{x \mapsto t+1\})$ are natural from their operational semantics and can be easily formalized, while structural rules like $(x := e)$ would consume more efforts to design. And there might exists no structural rules in certain cases, as in Esterel programs illustrated in our online report [34].

Main Contributions. In this paper, after defining the logic, we propose a cyclic proof system for DL_p based on program behaviours. Cyclic proof approach (cf. [8]) provides a solution for reasoning about programs with explicit/implicit loop structures in DL_p formulas. Generally, it is a technique to ensure that a certain type of infinite proof trees can lead to a valid conclusion if it meets some certain conditions.

The main contributions of this paper are three folds: (1) We define the syntax and semantics of DL_p formulas. (2) We build a proof system for reasoning about DL_p formulas based on program behaviours. (3) We construct a sound cyclic preproof structure for DL_p and analyzes 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 the soundness of DL_p. Section 6 introduces related work.

2 Dynamic Logic DL_p

In this section, we firstly introduce the basic settings for the ingredients of DL_p; then we introduce the notion of program behaviours; lastly we formally define the syntax and semantics of DL_p formulas.

Programs, Configurations and Formulas. The construction of DL_p is based on a set **TA** of *terms* defined over a set *Var* of variables and a set Σ of signatures. In **TA**, we distinguish three subsets **Prog**, **Conf** and **Form**, representing the sets of programs, configurations and formulas respectively. **TA** $\supseteq$ **Prog** $\cup$ **Conf** $\cup$ **Form**. We use $\equiv$ to represent the identical equivalence between two terms in **TA**. A function f : **TA** $\rightarrow$ **TA** is called *structural* (w.r.t. Σ) if it satisfies that for any $n-$ary signature $s_n \in \Sigma$ ($n \geq 0$, $0-$ary signature is a constant) and term $s_n(t_1, ..., t_n)$ with $t_1, ..., t_n \in$ **TA**, $f(s_n(t_1, ..., t_n)) \equiv s_n(f_1(t_1), ..., f_n(t_n))$ for some structural functions $f_1, ..., f_n$: **TA** $\rightarrow$ **TA**. Naturally, we assume that a structural function always maps a term to a term with the same type: **Prog**, **Conf** and **Form**.

In **Prog**, there is one distinguished program $\downarrow$, called *termination*. It indicates a completion of executions of a program.

Program configurations have impacts on formulas. We assume that **Conf** is associated with a function $\mathfrak{I}$: (**Conf** $\times$ **Form**) $\rightarrow$ **Form**, called *configuration interpretation*, that returns a formula by applying a configuration on a formula.

An *evaluation* ρ : **TA** $\rightarrow$ **TA** is a structural function that maps each formula $\phi \in$ **Form** to a *proposition* — a special formula that has a boolean semantics of either truth (represented as 1) or falsehood (represented as 0). We denote the set of all propositions as **Prop**. **Prop** $\subseteq$ **Form**. The boolean semantics of formulas is expressed by function $\mathfrak{T}$: **Prop** $\rightarrow \{0, 1\}$. **Prop** forms the semantical domain as the basis for defining DL_p. An evaluation $\rho \in$ **Eval** *satisfies* a formula $\phi \in$ **Form**, denoted by $\rho \models_{\mathfrak{T}} \phi$, if $\mathfrak{T}(\rho(\phi)) = 1$. A formula $\phi \in$ **Form** is *valid*, denoted by $\models_{\mathfrak{T}} \phi$, if $\rho \models_{\mathfrak{T}} \phi$ for all $\rho \in$ **Eval**.

Example 1 (An Instantiation of Program Structures). Consider a *While* program *WP* as an example of an instantiation of structures **Prog**, **Conf** and $\mathfrak{I}$:

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

Given an initial value of variables n, program *WP* computes the sum from n to 1 stored in variable s. A configuration σ_{WP} of *WP* is of the form: $\{x_1 \mapsto e_1, x_2 \mapsto e_2, ..., x_n \mapsto e_n\}$ ($n \geq 0$) as a variable storage that maps a variable x_i to a unique value of arithmetic expression e_i ($1 \leq i \leq n$). For example, $\{n \mapsto 5, s \mapsto 0\}$ denotes a configuration that maps n to 5 and s to 0. Formulas in this particular domain are first-order arithmetical formulas over integer numbers, with propositions are those "closed formulas" without free variables. The interpretation $\mathfrak{I}_{WP}(\sigma, \phi)$ is defined as a a substitution that replaces each free variable x of formula ϕ with its value-expression stored in σ. For instance, we have $\mathfrak{I}_{WP}(\{n \mapsto 5, s \mapsto 0\}, n > 0) = (5 > 0)$, which is true in the theory of integer numbers. An evaluation ρ_{WP} maps each variable to an integer number. $\rho_{WP}(t)$ given a term t is defined in the usual sense that it replaces each free variable x of term t with an integer number $\rho_{WP}(x)$.

A formal definition of *While* programs is illustrated in Appendix B of [34].

Program Behaviours. A *program state* is a pair $(\alpha, \sigma) \in$ **Prog** $\times$ **Conf** of programs and configurations. In **Form**, we assume a binary relation $(\alpha, \sigma) \longrightarrow$

(α', σ') called a *program transition* between two program states (α, σ) and (α', σ'). For any evaluation $\rho \in \mathbf{Eval}$, proposition $\rho((\alpha, \sigma) \longrightarrow (\alpha', \sigma'))$ is assumed to be defined as: $\rho((\alpha, \sigma) \longrightarrow (\alpha', \sigma')) =_{df} \rho(\alpha, \sigma) \longrightarrow \rho(\alpha', \sigma')$. *Program behaviours*, denoted by Λ, is the set of all true program transitions:

$$\Lambda =_{df} \{(\alpha, \sigma) \longrightarrow (\alpha', \sigma') \mid \mathfrak{T}((\alpha, \sigma) \longrightarrow (\alpha', \sigma')) = 1\},$$

which is assumed to be *well-defined* in the sense that there is no true transitions of the form: $(\downarrow, \sigma) \longrightarrow ...$ from a terminal program.

A *path tr* over Λ is a finite or infinite sequence: $s_1 s_2 ... s_n ...$ $(n \geq 1)$, where for each pair (s_i, s_{i+1}) $(i \geq 1)$, $(s_i \longrightarrow s_{i+1}) \in \Lambda$ is a true program transition. A path is *terminal*, if it ends with program $\downarrow$ in the form of $(\alpha_1, \sigma_1)...(\downarrow, \sigma_n)$ $(n \geq 1)$. We call a path *minimum*, in the sense that in it there is no two identical program states. A state s is called *terminal*, if from s there is a terminal path over Λ.

In order to reason about termination of a program, in **Form** we assume a unary operator $(\alpha, \sigma) \Downarrow$ called the *termination* of a program state (α, σ). For an evaluation $\rho \in \mathbf{Eval}$, proposition $\rho((\alpha, \sigma) \Downarrow)$ is defined as: $\rho((\alpha, \sigma) \Downarrow) =_{df} (\rho(\alpha, \sigma)) \Downarrow$. $\mathfrak{T}((\alpha, \sigma) \Downarrow) =_{df} 1$ if there exists a terminal path over Λ starting from (α, σ). We use Ω to denote the set of all true program terminations: $\Omega =_{df} \{(\alpha, \sigma) \Downarrow \mid \mathfrak{T}((\alpha, \sigma) \Downarrow) = 1\}$.

For programming languages, usually, program behaviours Λ (also program terminations Ω) are defined as structural operational semantics [28] of Plotkin's style expressed as a set of rules, in a manner that a transition $\mathfrak{T}((\alpha, \sigma) \longrightarrow (\alpha', \sigma')) = 1$ only when it can be inferred according to these rules in a finite number of steps.

Example 2 (Examples of Program Behaviours). In Example 1, we have a program transition $(WP, \{n \mapsto 5, s \mapsto 0\}) \longrightarrow (WP', \{n \mapsto 5, s \mapsto 5\})$, where $WP' =_{df} n := n - 1 \ ; \ WP$, by executing assignment $s := s + n$. This transition is also a program behaviour, since it matches the operational semantics of $s := s + n$ (not shown in this paper, one can refer to Appendix B of [34]). However, transition $(WP, \{n \mapsto v, s \mapsto 0\}) \longrightarrow (WP', \{n \mapsto v, s \mapsto 5\})$ is not a proposition, because its truth value depends on whether $v > 0$ is true.

Syntax and Semantics of DL_p. Based on the assumed sets **Prog**, **Conf** and **Form**, we give the syntax of DL_p as follows.

Definition 1 (DL_p Formulas). *A parameterized dynamic logical (DL_p) formula ϕ is defined as follows in BNF form:*

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

*where $F \in$ **Form**, $\alpha \in$ **Prog** and $\sigma \in$ **Conf**.*

We denote the set of DL_p formulas as $\mathbf{DL_p}$.

σ and ϕ are called the *label* and the *unlabeled part* of formula $\sigma : \phi$ respectively. We call $[\alpha]$ the *dynamic part* of a formula, and call a DL_p formula having

dynamic parts a *dynamic formula*. 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 $\langle\alpha\rangle\phi$ can be written as $\neg[\alpha]\neg\phi$. Other formulas with logical connectives such as $\vee$ and $\rightarrow$ can be expressed by formulas with $\neg$ and $\wedge$ accordingly. Formula $\sigma : \phi$ means that ϕ holds under configuration σ, as σ has an impact on the semantics of formulas as indicated by interpretation $\mathfrak{I}$.

The semantics of a traditional dynamic logic is based on a Kripke structure [16], in which programs are interpreted as a set of world pairs and logical formulas are interpreted as a set of worlds. In DL_p, we define the semantics by extending ρ and $\mathfrak{T}$ to formulas $\mathbf{DL_p}$ and directly base on program behaviours Λ.

Definition 2 (Semantics of DL_p Formulas). *An evaluation $\rho \in$ **Eval** structurally maps a DL_p formula ϕ into a proposition, whose truth value is defined inductively as follows by extending $\mathfrak{T}$:*

1. $\mathfrak{T}(\rho(F))$ *is already defined, if* $F \in$ **Form**;
2. $\mathfrak{T}(\rho(\sigma : F)) =_{df} 1$, *if* $F \in$ **Form** *and* $\rho \models_{\mathfrak{T}} \mathfrak{I}(\sigma, F)$;
3. $\mathfrak{T}(\rho(\sigma : \neg\phi)) =_{df} \mathfrak{T}(\rho(\neg(\sigma : \phi)))$;
4. $\mathfrak{T}(\rho(\sigma : \phi_1 \wedge \phi_2)) =_{df} \mathfrak{T}(\rho((\sigma : \phi_1) \wedge (\sigma : \phi_2)))$;
5. $\mathfrak{T}(\rho(\sigma : [\alpha]\phi)) =_{df} 1$, *if for all terminal paths* $(\alpha_1, \sigma_1)...(\downarrow, \sigma_n)$ $(n \geq 1)$ *over* Λ *with* $(\alpha_1, \sigma_1) \equiv \rho(\alpha, \sigma)$, $\mathfrak{T}(\rho(\sigma_n : \phi)) = 1$;
6. $\mathfrak{T}(\rho(\neg\phi)) =_{df} 1$, *if* $\mathfrak{T}(\rho(\phi)) = 0$;
7. $\mathfrak{T}(\rho(\phi_1 \wedge \phi_2)) =_{df} 1$, *if both* $\mathfrak{T}(\rho(\phi_1)) = 1$ *and* $\mathfrak{T}(\rho(\phi_2)) = 1$;
8. $\mathfrak{T}(\rho(\phi)) =_{df} 0$ *for any* $\phi \in \mathbf{DL_p}$, *otherwise.*

For any $\phi \in \mathbf{DL_p}$, write $\rho \models_{\mathfrak{T}} \phi$ if $\mathfrak{T}(\rho(\phi)) = 1$, or simply $\rho \models \phi$.

Notice that in Definition 2 we do not specify $\rho(\phi)$ for a DL_p formula ϕ due to its generality. We only assume that ρ is structural.

According to the semantics of operator $\langle\cdot\rangle$, we have that $\mathfrak{T}(\rho(\langle\alpha\rangle\phi)) = 1$ iff there exists a terminal path $(\alpha_1, \sigma_1)...(\downarrow, \sigma_n)$ $(n \geq 1)$ over Λ with $(\alpha_1, \sigma_1) \equiv \rho(\alpha, \sigma)$ such that $\mathfrak{T}(\rho(\sigma_n : \phi)) = 1$.

A DL_p formula ϕ is called *valid*, if $\rho \models \phi$ for all evaluation $\rho \in$ **Eval**.

Example 3 (DL_p Specifications). A property of program *WP* (Example 1) is described as the following formula

$$v \geq 0 \rightarrow \sigma_1 : [WP]s = ((v+1)v)/2,$$

where $\sigma_1 =_{df} \{n \mapsto v, s \mapsto 0\}$ with v a free variable. This formula means that given an initial value $v \geq 0$, after executing *WP*, s equals to $((v+1)v)/2$, which is the sum of $1 + 2 + ... + v$. We will prove this formula in Section 4.

3 A Cyclic Proof System for DL_p

We propose a cyclic proof system for DL_p. In Section 3.1, we firstly propose a proof system P_{DLP} to support reasoning based on program behaviours. Then in Section 3.2 we construct a cyclic preproof structure for P_{DLP}, which support deriving infinite proof trees under certain soundness conditions.

$$\frac{}{\Gamma \Rightarrow (x := e, \sigma) \longrightarrow (\downarrow, \sigma^{x}_{\sigma^*(e)}), \Delta}\ (x{:=}e) \qquad \frac{\Gamma \Rightarrow (\alpha_1, \sigma) \longrightarrow (\alpha_1', \sigma'), \Delta}{\Gamma \Rightarrow (\alpha_1; \alpha_2, \sigma) \longrightarrow (\alpha_1'; \alpha_2, \sigma'), \Delta}\ (;)$$

$$\frac{\Gamma \Rightarrow (\alpha_1, \sigma) \longrightarrow (\downarrow, \sigma'), \Delta}{\Gamma \Rightarrow (\alpha_1; \alpha_2, \sigma) \longrightarrow (\alpha_2, \sigma'), \Delta}\ (;\downarrow) \qquad \frac{\Gamma, \Im_{WP}(\sigma, \phi) \Rightarrow (\alpha, \sigma) \longrightarrow (\alpha', \sigma'), \Delta \quad \Gamma \Rightarrow \Im_{WP}(\sigma, \phi), \Delta}{\Gamma \Rightarrow (\mathit{while}\ \phi\ \mathit{do}\ \alpha\ \mathit{end}, \sigma) \longrightarrow (\alpha';\ \mathit{while}\ \phi\ \mathit{do}\ \alpha\ \mathit{end}, \sigma'), \Delta}\ (wh1)$$

$$\frac{\Gamma, \Im_{WP}(\sigma, \phi) \Rightarrow (\alpha, \sigma) \longrightarrow (\downarrow, \sigma'), \Delta \quad \Gamma \Rightarrow \Im_{WP}(\sigma, \phi), \Delta}{\Gamma \Rightarrow (\mathit{while}\ \phi\ \mathit{do}\ \alpha\ \mathit{end}, \sigma) \longrightarrow (\mathit{while}\ \phi\ \mathit{do}\ \alpha\ \mathit{end}, \sigma'), \Delta}\ (wh1\downarrow) \qquad \frac{\Gamma \Rightarrow \neg \Im_{WP}(\sigma, \phi), \Delta}{\Gamma \Rightarrow (\mathit{while}\ \phi\ \mathit{do}\ \alpha\ \mathit{end}, \sigma) \longrightarrow (\downarrow, \sigma), \Delta}\ (wh2)$$

Table 1. Partial Inference Rules for Program Behaviours of *While* programs

3.1 A Proof System for DL_p

Sequent Calculus. We adopt sequents [12] as the basic deduction structure to carry logical formulas. A sequent is of the form: $\Gamma \Rightarrow \Delta$, which expresses the formula $\bigwedge_{\phi \in \Gamma} \phi \rightarrow \bigvee_{\psi \in \Delta} \psi$, meaning that if all formulas in Γ hold, then one of formulas in Δ holds.

The Proof System. The proof system P_{DLP} of DL_p relies on a pre-defined proof system $P_{\Lambda,\Omega}$ for deriving program behaviours Λ and terminations Ω. $P_{\Lambda,\Omega}$ is *sound* and *complete* w.r.t. Λ and Ω in the sense that for any transition $(\alpha_1, \sigma_1) \longrightarrow (\alpha_2, \sigma_2) \in \mathbf{Prop}$ and termination $(\alpha, \sigma) \Downarrow \in \mathbf{Prop}$, $(\alpha_1, \sigma_1) \longrightarrow (\alpha_2, \sigma_2) \in \Lambda$ iff $\vdash_{P_{\Lambda,\Omega}} \cdot \Rightarrow (\alpha_1, \sigma_1) \longrightarrow (\alpha_2, \sigma_2)$, and $(\alpha, \sigma) \Downarrow \in \Omega$ iff $\vdash_{P_{\Lambda,\Omega}} \cdot \Rightarrow (\alpha, \sigma) \Downarrow$.

Note that in P_{DLP}, we usually assume that a formula does not contain a program transition or termination.

Example 4. As an example, Table 1 displays a part of inference rules of the proof system $P_{\Lambda_{WP},\Omega_{WP}}$ for the program behaviours Λ_{WP} of *While* programs. σ^x_e represents a configuration that store variable x as value e, while storing other variables as the same value as σ. $\sigma^*(e)$ is defined to return a term obtained by replacing each free variable x of e by the value of x in σ. See Appendix B of report [34] for a formal definition and for a proof system for program terminations Ω_{WP} of *While* programs.

Table 2 lists the primitive rules of P_{DLP}. Through P_{DLP}, a 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.

In Table 2 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 *contexts* Γ and Δ.

Target Pairs. In each rule of Table 2 except rule (*Sub*), we call a conclusion-premise (CP) pair (τ, τ') of formulas that are not in Γ and Δ a *target pair* of the rule. For instance, $(\sigma : [\alpha]\phi, \sigma' : [\alpha']\phi)$ is the target pair of rule $([\alpha]2)$, $(\neg\phi, \phi)$ is the target pair of rule $(\neg R)$.

$\dfrac{\{\Gamma \Rightarrow \sigma' : [\alpha']\phi, \Delta\}_{(\alpha',\sigma')\in\Phi}}{\Gamma \Rightarrow \sigma : [\alpha]\phi, \Delta}$ [1] ([α]), where $\Phi =_{df} \{(\alpha',\sigma') \mid \vdash_{P_{\Lambda,\Omega}} (\Gamma \Rightarrow (\alpha,\sigma) \longrightarrow (\alpha',\sigma'), \Delta)\}$					
$\dfrac{\Gamma, \sigma' : [\alpha']\phi \Rightarrow \Delta}{\Gamma, \sigma : [\alpha]\phi \Rightarrow \Delta}$ [1] ([α]2), if $\vdash_{P_{\Lambda,\Omega}} (\Gamma \Rightarrow (\alpha,\sigma) \longrightarrow (\alpha',\sigma'), \Delta)$					
$\dfrac{}{\Gamma \Rightarrow \Delta}$ [2] (*Ter*)	$\dfrac{\Im(\sigma,\phi)}{\sigma : \phi}$ [3] (*Int*)	$\dfrac{\sigma : \phi}{\sigma : [\downarrow]\phi}$ ([↓])	$\dfrac{\Gamma \Rightarrow \Delta}{Sub(\Gamma) \Rightarrow Sub(\Delta)}$ [4] (*Sub*)	$\dfrac{\neg(\sigma : \phi)}{\sigma : (\neg\phi)}$ ($\sigma\neg$)	$\dfrac{(\sigma : \phi) \wedge (\sigma : \psi)}{\sigma : (\phi \wedge \psi)}$ ($\sigma\wedge$)
$\dfrac{}{\Gamma, \phi \Rightarrow \phi, \Delta}$ (*ax*)	$\dfrac{\Gamma \Rightarrow \phi, \Delta \quad \Gamma, \phi \Rightarrow \Delta}{\Gamma \Rightarrow \Delta}$ (*Cut*)	$\dfrac{\Gamma \Rightarrow \Delta}{\Gamma, \phi \Rightarrow \Delta}$ (*Wk L*)	$\dfrac{\Gamma \Rightarrow \Delta}{\Gamma \Rightarrow \phi, \Delta}$ (*Wk R*)	$\dfrac{\phi, \phi}{\phi}$ (*Con*)	
$\dfrac{\Gamma \Rightarrow \phi, \Delta}{\Gamma, \neg\phi \Rightarrow \Delta}$ ($\neg L$)	$\dfrac{\Gamma, \phi \Rightarrow \Delta}{\Gamma \Rightarrow \neg\phi, \Delta}$ ($\neg R$)	$\dfrac{\Gamma, \phi, \psi \Rightarrow \Delta}{\Gamma, \phi \wedge \psi \Rightarrow \Delta}$ ($\wedge L$)	$\dfrac{\Gamma \Rightarrow \phi, \Delta \quad \Gamma \Rightarrow \psi, \Delta}{\Gamma \Rightarrow \phi \wedge \psi, \Delta}$ ($\wedge R$)		

[1] $\alpha \not\equiv \downarrow$. [2] for each ϕ in Γ or Δ, $\phi \in$ **Form**; $\Gamma \Rightarrow \Delta$ is valid. [3] $\phi \in$ **Form**. [4] *Sub* is by Definition 3.

Table 2. Primitive Rules of Proof System P_{DLP}

Illustration of each rule is as follows.

Rule ([α]) and rule ([α]2) ([34] displays its equivalent rule ($\langle\alpha\rangle$)) deal with dynamic parts of DL_p formulas based on program transitions. Both rules rely on the sub-proof-procedures of system $P_{\Lambda,\Omega}$. In rule ([α]), $\{...\}_{(\alpha',\sigma')\in\Phi}$ represents the collection of premises for all program states $(\alpha',\sigma') \in \Phi$. Notice that we assume that the system $P_{\Lambda,\Omega}$ only derives a finite set of transitions from (α,σ) (which is mostly the case in practice). So Φ is finite, and rule ([α]) only has a finite number of premises. When Φ is empty, the conclusion terminates. Intuitively, rule ([α]) says that to to derive $[\alpha]\phi$ under configuration σ, we derive $[\alpha']\phi$ under configuration σ' for each transition from (α,σ) to (α',σ'). Compared to rule ([α]), rule ([α]2) has only one premise for some (α',σ'). Intuitively, it says that if $[\alpha]\phi$ holds under configuration σ and if there is a transition $(\alpha,\sigma) \longrightarrow (\alpha',\sigma')$, then we have $[\alpha']\phi$ holds under σ'.

In rule (*Ter*), formula $\Gamma \Rightarrow \Delta$ is a non-dynamic formula. The introduction rule (*Int*) eliminates the label σ during the derivation. Rule ([↓]) deals with the situation when the program is a termination ↓. Its soundness is straightforward by the well-definedness of program behaviours Λ (Section 2).

Rule (*Sub*) describes a specialization process (from premise to conclusion) for DL_p formulas. For a set A of formulas, $Sub(A) =_{df} \{Sub(\phi) \mid \phi \in A\}$, with *Sub* a function defined as follows in Definition 3. Intuitively, if formula $\Gamma \Rightarrow \Delta$ is valid, then its one of special cases $Sub(\Gamma) \Rightarrow Sub(\Delta)$ through an abstract version of substitutions *Sub* on formulas is valid. Rule (*Sub*) plays an important role in constructing a bud in a cyclic preproof structure (Section 3.2). See Section 4 as an example.

Definition 3 (Substitution). *A structural function η : **TA** → **TA** is called a 'substitution', if for any evaluation $\rho \in$ **Eval**, there exists a $\rho' \in$ **Eval** such that $\rho(\eta(\phi)) \equiv \rho'(\phi)$ for each formula $\phi \in \mathbf{DL_p}$.*

Rules ($\sigma\neg$) and ($\sigma\wedge$) are for transforming labeled formulas. Their soundness are direct according to the semantics of labeled formulas (Definition 2). From rule (ax) to ($\sigma \wedge R$) are the rules inherited from traditional first-order logic. The meanings of these rules are classic and are omitted in this paper.

The soundness of the rules in P_{DLP} is stated as follows.

Theorem 1. *All rules in P_{DLP} (Table 2) are sound.*

Following the above explanations, Theorem 1 can be proved directly according to the semantics of DL_p. See Appendix A of report [34] for more details.

3.2 Construction of A Cyclic Preproof Structure in P_{DLP}

In the proof system P_{DLP}, given a sequent $\Gamma \Rightarrow \Delta$, we expect a finite proof tree to prove $\Gamma \Rightarrow \Delta$. However, a branch of a proof tree does not always terminate. Because the process of symbolically executing a program via rule ($[\alpha]$) or/and rule ($[\alpha]2$) 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 transitions: $(W, \{x \mapsto 0\}) \longrightarrow (W, \{x \mapsto 1\}) \longrightarrow ....$

In this paper, we apply the cyclic proof approach (cf. [8]) to deal with a potential infinite proof tree in DL_p. Cyclic proof is a technique to ensure a valid conclusion of a special type of infinite proof trees, called 'preproof', under some certain conditions called "soundness conditions". Below we firstly introduce a preproof structure, then based on it we propose a 'cyclic' preproof — a structure that satisfies certain soundness conditions. Section 5 will explain why cyclic preproof can be sound.

Preproof & Derivation Traces. A *preproof* (cf. [8]) is an infinite proof tree expressed as a tree structure with finite 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 N is called a *companion of* N.

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 of the preproof, where for each node pair (ν_i, ν_{i+1}) $(i \geq 1)$, ν_i is the conclusion and ν_{i+1} is a premise, of a rule.

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 starting from formula τ_1 in node ν_1 such that each pair (τ_i, τ_{i+1}) $(i \geq 1)$ is either (1) a target pair of a rule in Table 2 (except rule (Sub)); (2) or a pair $(Sub(\phi), \phi)$ of rule (Sub) for some formula ϕ; (3) or a CP pair of a rule satisfying $\tau_i \equiv \tau_{i+1}$ in Table 2.

Progressive Steps. A crucial prerequisite of defining cyclic preproofs is to introduce the notion of *(progressive) derivation traces* in a preproof of P_{DLP}.

Definition 4 (Progressive Step/Progressive Derivation Trace in DL_p). *In a preproof of 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 formula pair (τ_i, τ_{i+1}) $(1 \leq i \leq m)$ over (ν_i, ν_{i+1}) is called a "progressive step", if (τ_i, τ_{i+1}) is either (1)*

a target pair of an instance of rule ($[\alpha]$)*; (2) Or the target pair of an instance of rule* ($[\alpha]2$)*, provided with an additional condition that* $\vdash_{P_{\Lambda,\Omega}} (\Gamma \Rightarrow (\alpha', \sigma') \Downarrow, \Delta)$.

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

The additional condition of the instance of rule ($[\alpha]2$) is the key to prove the corresponding case in Lemma 1 of [34].

Cyclic Preproofs. We build a sound cyclic preproof structure as the following definition.

Definition 5 (A Cyclic Preproof for DL_p). *In P_{DLP}, a preproof is a 'cyclic' one, if there exists a progressive trace over every derivation path.*

Theorem 2 (Soundness of the Cyclic Preproof for DL_p). *Provided that $P_{\Lambda,\Omega}$ is sound, a cyclic preproof of P_{DLP} has a sound conclusion.*

Theorem 2 will be analyzed in Section 5.

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 sets **Prog**, **Conf** and **Form**. 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

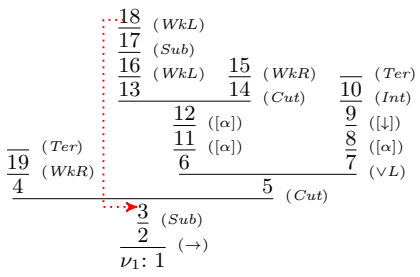
We give an example to show how a DL_p formula can be derived according to rules in Table 2. We prove the property given in Example 3, stated as the following sequent

$$\nu_1 =_{df} \cdot \Rightarrow v \geq 0 \rightarrow \sigma_1 : [WP](s = ((v+1)v)/2).$$

Table 3 shows the derivation. We omit all sub-proof-procedures in the proof system $P_{\Lambda_{WP},\Omega_{WP}}$ derived by the inference rules in Table 1 when deriving via rule ($[\alpha]$). We write term $t(x)$ if x is a variable that has free occurrences in t. Non-primitive rules ($\rightarrow R$) and ($\vee L$) can be derived by the rules for $\neg$ and $\wedge$ accordingly. The derivation from sequent 2 to 3 is according to rule $\dfrac{\Gamma \Rightarrow \Delta}{\Gamma[e/x] \Rightarrow \Delta[e/x]}\ ^{(Sub)}$, where function $(\cdot)[e/x]$ is an instantiation of abstract substitution defined in Definition 3. Informally, for any formula ϕ, $\phi[e/x]$ substitutes each free variable x of ϕ with term e. We observe that sequent 2 can be writen as:

$$(v - m \geq 0)[0/m] \Rightarrow (\sigma_3(v) : [WP]\phi_1)[0/m],$$

as a special case of sequent 3. Intuitively, sequent 3 captures the situation after the mth loop ($m \geq 0$) of program *WP*. This step is crucial as starting from sequent 3, we can find a bud node — 18 — that is identical to node 3. The derivation from sequent 3 to {4, 5} provides a lemma: $v - m > 0 \vee v - m \leq 0$, which is obvious valid. Sequent 16 indicates the end of the $(m+1)$th loop of program *WP*. From node 12 to 16, we rewrite the formulas on the left side into



Definitions of other symbols:
$WP =_{df} \{while\ (n > 0)\ do\ s := s + n\ ;\ n := n - 1\ end\}$
$\phi_1 =_{df} (s = ((v+1)v)/2)$
$\sigma_1(v) =_{df} \{n \mapsto v, s \mapsto 0\}$
$\sigma_2(v) =_{df} \{n \mapsto v, s \mapsto 0\}$
$\sigma_3(v) =_{df} \{n \mapsto v - m, s \mapsto (2v - m + 1)m/2\}$
$\sigma_4(v) =_{df} \{n \mapsto v - m, s \mapsto (2v - (m+1) + 1)(m+1)/2\}$
$\sigma_5(v) =_{df} \{n \mapsto v - (m+1), s \mapsto (2v - (m+1) + 1)(m+1)/2\}$

1:	$\cdot$	$\Rightarrow$	$v \geq 0 \to \sigma_1(v) : [WP]\phi_1$
2:	$v \geq 0$	$\Rightarrow$	$\sigma_1(v) : [WP]\phi_1$
3:	$v - m \geq 0$	$\Rightarrow$	$\underline{\sigma_3(v) : [WP]\phi_1}$
4:	$v - m \geq 0$	$\Rightarrow$	$\sigma_3(v) : [WP]\phi_1, v - m > 0 \vee v - m \leq 0$
19:	$v - m \geq 0$	$\Rightarrow$	$v - m > 0 \vee v - m \leq 0$
5:	$v - m \geq 0, v - m > 0 \vee v - m \leq 0$	$\Rightarrow$	$\underline{\sigma_3(v) : [WP]\phi_1}$
6:	$v - m \geq 0, v - m > 0$	$\Rightarrow$	$\underline{\sigma_3(v) : [WP]\phi_1}$
11:	$v - m \geq 0, v - m > 0$	$\Rightarrow$	$\underline{\sigma_4(v) : [n := n - 1\ ;\ WP]\phi_1}$
12:	$v - m \geq 0, v - m > 0$	$\Rightarrow$	$\underline{\sigma_5(v) : [WP]\phi_1}$
13:	$\left\{\begin{array}{l} v - m \geq 0, v - m > 0, \\ v - (m+1) \geq -1, v - (m+1) \geq 0 \end{array}\right\}$	$\Rightarrow$	$\underline{\sigma_5(v) : [WP]\phi_1}$
16:	$v - (m+1) \geq -1, v - (m+1) \geq 0$	$\Rightarrow$	$\underline{\sigma_5(v) : [WP]\phi_1}$
17:	$v - m \geq -1, v - m \geq 0$	$\Rightarrow$	$\underline{\sigma_3(v) : [WP]\phi_1}$
18:	$v - m \geq 0$	$\Rightarrow$	$\underline{\sigma_3(v) : [WP]\phi_1}$
14:	$v - m \geq 0, v - m > 0$	$\Rightarrow$	$\sigma_5(v) : [WP]\phi_1, v - (m+1) \geq -1, v - (m+1) \geq 0$
15:	$v - m \geq 0, v - m > 0$	$\Rightarrow$	$v - (m+1) \geq -1, v - (m+1) \geq 0$
7:	$v - m \geq 0, v - m \leq 0$	$\Rightarrow$	$\sigma_3(v) : [WP]\phi_1$
8:	$v - m \geq 0, v - m \leq 0$	$\Rightarrow$	$\sigma_3(v) : [\downarrow]\phi_1$
9:	$v - m \geq 0, v - m \leq 0$	$\Rightarrow$	$\sigma_3(v) : (s = ((v+1)v)/2)$
10:	$v - m \geq 0, v - m \leq 0$	$\Rightarrow$	$((2v - m + 1)m)/2 = ((v+1)v)/2)$

Table 3. Derivations of Property ν_1

an obvious logical equivalent form in order to apply rule (*Sub*). From sequent 16 to 17 rule (*Sub*) is applied, with 16 can be written as:

$$(v - m \geq -1)[m + 1/m], (v - m \geq 0)[m + 1/m] \Rightarrow (\sigma_3 : [WP]\phi_1)[m + 1/m].$$

The whole proof tree is a cyclic preproof because the only derivation path: $1, 2, 3, 5, 6, 11, 12, 13, 16, 17, 18, 3, ...$ has a progressive derivation trace whose elements are underlined in Table 3.

Unlike traditional verifications of dynamic logic and Hoare logic, the derivations shown in Table 3 rely solely on the symbolic executions of while programs. In this approach, verifying a while loop reduces to selecting an appropriate configuration using rule (*Sub*). This eliminates the need for a dedicated rule to dissolve the while structure, as is typically required in classical Hoare logic.

Mechanization of DL_p. To further demonstrate its applicability we mechanize the theory of DL_p in Coq. We adopt the so-called "deep embedding" and currently have implemented the syntactic structure of DL_p as Coq objects and

managed to perform the deduction of each rule in proof system P_{DLP}. The source code can be found in [1]. The semantics of DL_p and the soundness of DL_p has not been validated in Coq yet, which we attribute to one of our future work.

5 Soundness Analysis of Cyclic Proof System P_{DLP}

In this section, we analyze the soundness of the cyclic proof system P_{DLP} as stated as Theorem 2. By following a similar idea behind [7], we manage to prove the soundness for formulas in which the target formula $\sigma : [\alpha]\phi$ has only a single dynamic part $[\alpha]$. This case, however, already encompasses the full expressiveness of Hoare logic and is adequate for most practical applications of program verifications. Below we illustrate the main idea of the proof using the example given in Section 4. A complete proof can be found in [34].

Well-foundedness. 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. $\preceq$.

Main Idea of Soundness Proof. We look for a type of well-founded sets under some partial-order relation $\preceq$ which are related to the truth values of dynamic formulas. For any dynamic formula τ, let its corresponding well-founded set be $C(\tau)$. This type of well-founded sets satisfies that for any pair (τ, τ') of a derivation trace in a derivation of DL_p formulas using rules in Table 2, we have $C(\tau) \succeq C(\tau')$; Moreover, if the derivation from τ to τ' is progressive (Definition 4), then $C(\tau) \succ C(\tau')$. This property is stated as Lemma 1 of [34].

The proof is conducted by contradiction. If the conclusion of a cyclic preproof is invalid, then there exists an *invalid derivation path* in which each node is invalid, and by Definition 5, there exists a progressive trace: $\tau_1\tau_2...\tau_n$ over the tail of the derivation path. By the property above, we can then obtain a sequence of well-founded sets: $C(\tau_1) \succeq C(\tau_2) \succeq ... \succeq C(\tau_n) \succeq ...$, in which due to the progressiveness of the derivation trace there are an infinite number of relation $\succ$ among these $\succeq$s. This violates the well-foundedness itself introduce above.

Illustrations in Our Example. In the example of *While* programs shown in this paper, given a formula $\sigma : [\alpha]\phi$, for any evaluation ρ, we consider a "counter-example" set $CT(\rho, \sigma : [\alpha]\phi)$ defined as the set of minimum paths of α that make $\sigma : [\alpha]\phi$ invalid, which is formally defined as follows:

$$CT(\rho, \sigma : [\alpha]\phi) =_{df} \{\rho(\alpha, \sigma)...(\downarrow, \sigma') \mid \rho \models \Gamma, \rho \not\models \sigma' : \phi\}.$$

Set $CT(\rho, \sigma : [\alpha]\phi)$ for any while program α is always finite. A relation $\preceq_m$ between two sets CT_1 and CT_2 is defined such that $CT_1 \preceq CT_2$ if each path of CT_1 is a suffix of a path in CT_2. And it is easy to prove that $\preceq_m$ is well-founded.

In the derivation shown in Section 4, if we assume the conclusion ν_1 is invalid, then by the soundness of each rule of Table 2, we can obtain (the only) one invalid derivation path: 1, 2, 3, 6, 11, 12, 13, 16, 17, 18, 1, ..., in which the derivation trace underlined in Table 2 is progressive. Starting from an evaluation ρ_0 that makes node 1 invalid, for each pair (τ, τ') of the derivation trace, we can prove

that $CT(\rho, \tau) \succeq CT(\rho', \tau')$ for some ρ and ρ'. Especially, if the rule is $([\alpha])$, e.g., the pair over nodes (11, 12), we can prove that $CT(\rho, \tau) \succ CT(\rho', \tau')$, because each path of $CT(\rho', \tau')$, according to the meaning of rule $([\alpha])$, turns out to be a proper suffix of a path of $CT(\rho, \tau)$. This progressive derivation trace leads to the contradiction of the well-foundedness of relation $\preceq_m$.

In the proof given in [34], more restrictions on programs are needed to obtain a finite well-founded set C. One can find more details of the proof in [34].

6 Related Work

Previous work such as [29,30,31,9] in last decade addressed reasoning based on program behaviours using theories based on rewriting logic [21]. Matching logic [29] describes reachability between patterns, where a reachability rule $\varphi \Rightarrow \varphi'$ captures whether a pattern φ' is reachable from another pattern φ in a given pattern reachability system. Based on matching logic, one-path and all-paths reachability logics [30,31] were developed by enhancing the expressive power of the reachability rule. A more powerful matching μ-logic [9] was proposed by adding a least fixpoint μ-binder to matching logic.

Compared to these theories, the forms of dynamic logical formulas allow a direct description (without further encoding) of program properties, provided with the support of a proof system for efficiently deriving modalities. In terms of expressiveness, the semantics of modality $[\cdot]$ in dynamic logic cannot be fully captured by matching logic and one-path reachability logic, if the programs are non-deterministic.

[22] and [20] proposed a type of general program verification frameworks based on coinduction. A specification like $\sigma : [\alpha]\phi$ in DL_p was captured by a more general structure over mathematical sets. The deduction of a specification was realized through deductions over set relations, making use of a type of fixpoint theories over sets. While set theory may permit greater generality, logical forms like $[\alpha]\phi$ offer a clearer separation between programs and properties, and enables a more structured and manageable deduction process. In terms of expressiveness, the meaning of modality $\langle\cdot\rangle$ in DL_p cannot be directly expressed in the framework of [22].

The structure 'updates' adopted in work [26,3,4] are "delay substitutions" of variables and terms. It is a special case of the configurations in DL_p, if we take a configuration as a meta variable without explicit structures.

Cyclic proof approach firstly arose in [32] and was later developed in different logics [8,7]. [19] proposed a complete cyclic proof system for μ-calculus, which subsumes PDL [11] in its expressiveness. In [10], the authors proposed a complete labelled cyclic proof system for PDL. Both μ-calculus and PDL, as mentioned in Section 1, are logics designed for dissolving regular expressions as their program models. The labelled form of DL_p formula are alike in [10], but of a more general meaning and can be of arbitrary terms.

There has been some other work for generalizing the theories of dynamic logics, such as [23,17]. However, none of them allow general structures as in

DL_p, nor adopt a similar approach for reasoning about programs. The logic proposed in [23], though for a general structure of monads, is still in explicit forms, thus is different from the design philosophy of DL_p. [17] modifies PDL by generalizing regular expressions as arbitrary programs. However, there the program behaviours are captured by pure program transitions with abstract actions, e.g. $\alpha \xrightarrow{a} \alpha'$. And yet no proof systems have been proposed and analyzed for the logic.

7 Conclusion & Future Work

As a summery, we claim that the theory of DL_p hopefully will become a candidate for unifying different dynamic-logic theories, to provide a convenient framework for reasoning based on symbolic executions directly. This is particularly important and has potentially many applications, as most of computer programs have operational semantics in their nature. Although an independent structure not built upon any other logical theories, DL_p actually provides a promising method for fast-prototyping program-verification infrastructures in general theorem provers like Coq or Isabelle. More investigations on this issue will be carried out in future.

Acknowledgment. This work is partially supported by the projects of National Science Foundation of China No. 62102329 and No. 62272397.

References

1. `https://github.com/yrz5a/Coq-DLp.git`
2. Appel, A.W., Dockins, R., et al.: Program Logics for Certified Compilers. Cambridge University Press (2014)
3. Beckert, B., Bruns, D.: Dynamic logic with trace semantics. In: CADE 2013. pp. 315–329. Springer Berlin Heidelberg (2013)
4. Beckert, B., Klebanov, V., Weiß, B.: Dynamic Logic for Java, pp. 49–106. Springer International Publishing (2016)
5. 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)
6. Berry, G., Gonthier, G.: The Esterel synchronous programming language: design, semantics, implementation. Science of Computer Programming **19**(2), 87 – 152 (1992)
7. Brotherston, J., Bornat, R., Calcagno, C.: Cyclic proofs of program termination in separation logic. SIGPLAN Not. **43**(1), 101–112 (2008)
8. Brotherston, J., Simpson, A.: Complete sequent calculi for induction and infinite descent. In: LICS 2007. pp. 51–62 (2007)
9. Chen, X., Rosu, G.: Matching *mu*-logic. In: LICS 2019. pp. 1–13. IEEE Computer Society (jun 2019)
10. 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)
11. Fischer, M.J., Ladner, R.E.: Propositional dynamic logic of regular programs. Journal of Computer and System Sciences **18**(2), 194–211 (1979)

12. Gentzen, G.: Untersuchungen über das logische Schließen. Ph.D. thesis, NA Göttingen (1934)
13. Gesell, M., Schneider, K.: A Hoare calculus for the verification of synchronous languages. In: PLPV 2012. p. 37–48. Association for Computing Machinery (2012)
14. Goodfellow, I.J., Bengio, Y., Courville, A.: Deep Learning. MIT Press (2016)
15. Harel, D.: First-Order Dynamic Logic, Lecture Notes in Computer Science (LNCS), vol. 68. Springer (1979)
16. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. MIT Press (2000)
17. Hennicker, R., Wirsing, M.: A Generic Dynamic Logic with Applications to Interaction-Based Systems, pp. 172–187. Springer International Publishing (2019)
18. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
19. Jungteerapanich, N.: A tableau system for the modal μ-calculus. In: TABLEAUX 2009. pp. 220–234. Springer Berlin Heidelberg (2009)
20. 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)
21. Meseguer, J.: Twenty years of rewriting logic. The Journal of Logic and Algebraic Programming **81**(7), 721–781 (2012)
22. Moore, B., Peña, L., Rosu, G.: Program verification by coinduction. In: ESOP 2018. pp. 589–618. Springer International Publishing (2018)
23. 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)
24. O'Hearn, P.W.: Incorrectness logic. Proc. ACM Program. Lang. **4**(POPL) (2019)
25. 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)
26. Platzer, A.: Differential dynamic logic for verifying parametric hybrid systems. In: TABLEAUX 2007. pp. 216–232. Springer Berlin Heidelberg (2007)
27. Platzer, A.: Logical Foundations of Cyber-Physical Systems. Springer (2018)
28. Plotkin, G.D.: A structural approach to operational semantics. Tech. Rep. DAIMI FN-19, University of Aarhus (1981)
29. Roşu, G., Ştefănescu, A.: Towards a unified theory of operational and axiomatic semantics. In: ICALP 2012. pp. 351–363. Springer Berlin Heidelberg (2012)
30. Rosu, G., Stefanescu, A., Ciobâcá, S., Moore, B.M.: One-path reachability logic. In: LICS 2013. pp. 358–367 (2013)
31. Ştefănescu, A., Ciobâcă, Ş., et al.: All-path reachability logic. In: RTA-TLCA 2014. pp. 425–440. Springer International Publishing (2014)
32. Stirling, C., Walker, D.: Local model checking in the modal mu-calculus. Theor. Comput. Sci. **89**(1), 161–177 (1991)
33. Yang, Z., Hu, K., et al.: From AADL to timed abstract state machines: A verified model transformation. Journal of Systems and Software **93**, 42–68 (2014)
34. Zhang, Y.: Parameterized dynamic logic — towards a cyclic logical framework for general program specification and verification (2024), `https://arxiv.org/abs/2404.18098v4`
35. Zhang, Y., Mallet, F., Liu, Z.: A dynamic logic for verification of synchronous models based on theorem proving. Front. Comput. Sci. **16**(4) (2022)