

On A Parameterized Theory of Dynamic Logic for Operationally-based Programs

YUANRUI ZHANG, College of Software, Nanjing University of Aeronautics and Astronautics, China

Applying dynamic logics to program verifications is a challenge, because their axiomatic rules for regular expressions can be difficult to be adapted to different program models. We present a novel dynamic logic, called DL_p, which supports reasoning based on programs' operational semantics. For those programs whose transitional behaviours are their standard or natural semantics, DL_p makes their verifications easier since one can directly apply the program transitions for reasoning, without the need of re-designing and validating new rules as in most other dynamic logics. DL_p is parametric. It provides a model-independent framework consisting of a relatively small set of inference rules, which depends on a given set of trustworthy rules for the operational semantics. These features of DL_p let multiple models easily compared in its framework and makes it compatible with existing dynamic-logic theories. DL_p supports cyclic reasoning, providing an incremental derivation process for recursive programs, making it more convenient to reason about without prior program transformations. We analyze and prove the soundness and completeness of DL_p under certain conditions. Several case studies illustrate the features of DL_p and fully demonstrate its potential usage.

CCS Concepts: • **Theory of computation** → **Modal and temporal logics; Hoare logic; Logic and verification; Proof theory.**

Additional Key Words and Phrases: Dynamic Logic, Program Deduction, Verification Framework, Cyclic Proof, Operational Semantics, Symbolic Execution

ACM Reference Format:

Yuanrui Zhang. 2018. On A Parameterized Theory of Dynamic Logic for Operationally-based Programs. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX)*. ACM, New York, NY, USA, 39 pages. <https://doi.org/XXXXXXX.XXXXXXX>

1 Introduction

Dynamic logic [29] has proven to be a valuable program logic for specifying and reasoning about different types of programs. As a “multi-modal” logic that integrates both programs and formulas into a single form, it is more expressive than traditional Hoare logic [31] (cf. [4]). Dynamic logic has been successfully applied to domains such as process algebras [8], programming languages [7], synchronous systems [62, 63], hybrid systems [44, 45] and probabilistic systems [35, 43]. These theories have inspired the development of related verification tools for safety-critical systems, such as KIV [50], KeY [54], KeYmaera [46], and the tools developed in [19, 63]. Recent work in dynamic logic features a variety of extensions designed to tackle modern problems, including guaranteeing the correctness of blockchain protocols [32], formalizing hyper-properties [25], and verifying quantum computations [19, 58]. It also has attracted attention as a promising framework for incorrectness reasoning, as explored recently in [42, 64].

Author's Contact Information: Yuanrui Zhang, yuanruizhang@nuaa.edu.cn, College of Software, Nanjing University of Aeronautics and Astronautics, Nanjing, Jiangsu, China.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.

Manuscript submitted to ACM

Manuscript submitted to ACM

1

The theories of most dynamic logics, as well as Hoare-style logics, are built up based on the denotational semantics of programs: The behaviour of a program is interpreted as a mathematical object (e.g. a set of traces), and a set of inference rules are constructed to match this object. Adapting these theories to other programs can be difficult. This is true especially for the programming languages such as Java, C and Esterel [11], whose semantics is very complex. Building logic theories for them requires carefully designing a large set of rules specific in their program domains. Moreover, these rules are often error prone, thus requiring validation of their soundness (and even completeness), which can also be costly. For example, in KeY [54], to apply first-order dynamic logic [49] (FODL) to the verification of Java programs, more than 500 inference rules are proposed for the primitives of Java (cf. [41]). Their correctness is hard to be guaranteed. Another example is that the tool Verifiable C [5] spends nearly 40,000 lines of Rocq code to define and validate its logic theory for C based on separation logic [51].

Another main issue is that to reason about some types of programs, one has to first transform them into some sorts of “standard forms”, in order to apply suitable axiomatic rules. These beforehand transformations are unnecessary and can be expensive. And they usually mean breaking the original program structures and thus can cause loss of program information. A typical example is imperative synchronous programming languages such as Esterel [11] or Quartz [55]. In [24], it shows that how a synchronous program must be transformed into a so-called “STA program” in order to apply the right Hoare-logic rules to it.

Different from denotational semantics, structural operational semantics [47] describes how a program is transitioned to another program under some configuration. It is the standard semantics for concurrent models, such as CCS [38] and π -calculus [39]. For executable programs such as those mentioned above, providing an operational semantics is straightforward, since their executions are intended to directly transform program configurations. For this reason, in most cases, the operational semantics can be trusted as given, without additional validations (cf., e.g., [14, 21]).

In this paper, we propose a dynamic-logic-based theory aimed for easing the reasoning of those programs whose operational semantics is in their nature. We propose a so-called *parameterized dynamic logic*, abbreviated as DLp¹. It supports a directly reasoning based on operational semantics. Unlike previous work such as [8, 9] which focus on particular calculi, our framework is parametric and can be adapted to arbitrary programs and formulas. We present a proof system for DLp based on a cyclic proof approach (cf. [16]). It provides a set of “kernel rules” for deriving DLp formulas, accompanied with an assumed set of rules for programs’ transitional behaviours. We study and prove the soundness and completeness of DLp under a general setting.

Compared to the traditional approaches based on dynamic logics, DLp has several advantages: (1) For those programs for which the operational semantics is easy to obtain and can be trusted, it reduces the burden of the target-model adaptations and consistency validations for unreliable rules. Compared to the previous work (like [7]), our set of “kernel rules” is very small. (2) Its parameterization of formulas provides a model-independent framework, in which different program theories can be easily embedded through a lifting process, and multiple models can be easily compared. (3) Its support of cyclic reasoning is a natural solution for infinite symbolic executions caused by recursive programs, which also allows an incremental derivation process by avoiding prior program transformations for certain types of “non-standard” program models.

Previous work mostly related to ours have addressed this issue in different mathematical logics or theories (e.g. [2, 17, 30, 36, 40, 52, 53, 56, 60], see Section 8 for a detailed comparison). Except a few [2, 30, 60], most of them has not yet

¹To avoid the conflict with the famous Propositional Dynamic Logic (PDL) [23]

concerned with an efficient logical calculus for deriving dynamic-logic formulas. To our best knowledge, DLp is the first dynamic logic to provide a cyclic verification framework for direct operationally-based reasoning of different programs.

This paper is a non-trivial extension of the previous work [61], from which we take an entire different method to build up the theory of DLp that is based on Kripke structures and is independent from explicit signatures. In this work, we further make a full analysis of the soundness and completeness of DLp, as well as a much richer cases analysis for different features of DLp.

2 An Overview

In dynamic logic, a *dynamic formula* is of the form: $[\alpha]\phi$ (cf. [29]), where $[\cdot]$ is a modal operator, α is a program model (or simply “program”), ϕ is a logical formula. Intuitively, it means that after the terminations of all executions of program α , formula ϕ holds. When α is deterministic, formula $\phi \rightarrow [\alpha]\psi$ exactly captures the partial correctness of the triple $\{\phi\}\alpha\{\psi\}$ in Hoare-style logics (e.g. [31, 51]). $\phi \rightarrow \langle\alpha\rangle\psi$ captures the total correctness of $\{\phi\}\alpha\{\psi\}$, with $\langle\cdot\rangle$ the dual modal operator of $[\cdot]$. $\langle\alpha\rangle\psi$ is defined such that $\langle\alpha\rangle\psi = \neg[\alpha]\neg\psi$, meaning that there exists an execution of α satisfying that it terminates and after its termination, formula ψ holds. As a combination of both programs and formulas, a dynamic formula allows multiple and nested modalities in forms like $[\alpha]\phi \rightarrow \langle\beta\rangle\psi$, $[\alpha]\langle\beta\rangle\phi$, $[\alpha](\phi \rightarrow \langle\beta\rangle\psi)$, etc., making it strictly more expressive than Hoare logic (cf. [4]).

The rest of this section gives an outline of the main work on DLp, focusing on the main ideas illustrated through examples. They are introduced in details in the following sections of this paper.

2.1 Labeling and Parameterization of Dynamic Logic

In order to directly reason about programs via their operational semantics, in DLp, we introduce a “label” σ (Definition 4.8) to capture explicit program structures as the current program configurations for symbolic executions. σ attaches a dynamic formula $[\alpha]\phi$, yielding a labeled formula of the form $\sigma : [\alpha]\phi$. Intuitively, it means that under configuration σ , program α can be executed and formula ϕ holds after all terminating executions of α . The introduction of labels allows us to derive labeled formulas $\sigma : [\alpha]\phi$ by the following program transitions:

$$(\alpha, \sigma) \longrightarrow (\alpha', \sigma'),$$

using inference rules conceptually explained as the form:

$$\frac{\sigma' : [\alpha']\phi, \text{ for all } (\alpha', \sigma') \text{ such that } (\alpha, \sigma) \longrightarrow (\alpha', \sigma')}{\sigma : [\alpha]\phi} \quad ([\alpha]).$$

These rules (corresponding to the rules $([\alpha]R)$ and $([\alpha]L)$ in Table 2) reduce the deduction of $\sigma : [\alpha]\phi$ to the deductions of all successor formulas $\sigma' : [\alpha']\phi$ corresponding to the one-step program transitions.

Due to the universal form of program transitions, this framework applies for arbitrary program models and configurations. Consequently, in a labeled formula $\sigma : [\alpha]\phi$ of DLp, we parameterize the program α , the logical formula ϕ and the label σ , to allow them to have any algebraic structures. $\sigma : [\alpha]\phi$ turns out to be a more general form than $[\alpha]\phi$. When σ is “free” (cf. Definition 5.8) w.r.t. $[\alpha]\phi$, $\sigma : [\alpha]\phi$ has the same meaning as $[\alpha]\phi$.

Consider a formula $\phi_1 =_{df} (x \geq 0 \rightarrow [x := x + 1]x > 0)$ in FODL [49], 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 the assignment rule:

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

on the part $[x := x + 1]x > 0$. It substitutes x of $x > 0$ with $x + 1$, yielding expression $x + 1 > 0$. After the derivation we obtain formula $\phi'_1 =_{df} (x \geq 0 \rightarrow x + 1 > 0)$, which is true for any $x \in \mathbb{Z}$.

In DLp, on the other hand, we can express ϕ_1 as an equivalent labeled formula: $\psi_1 =_{df} (t \geq 0 \rightarrow \{x \mapsto t\} : [x := x + 1]x > 0)$, where the label $\{x \mapsto t\}$ means that variable x stores value t (with t a fresh variable). With the program configurations explicitly showing up, to derive formula ψ_1 , we instead directly apply the above rule ($[\alpha]$) on the part $\{x \mapsto t\} : [x := x + 1]x > 0$ according to the program transition

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

which assigns the value $t + 1$ to x afterwards. Here \downarrow indicates a program termination (cf. Definition 4.1). After the derivation, we obtain the formula $\psi'_1 =_{df} (t \geq 0 \rightarrow \{x \mapsto t + 1\} : x > 0)$, where formula $\{x \mapsto t + 1\} : x > 0$ exactly means $t + 1 > 0$ if we replace x with its current value $t + 1$ in formula $x > 0$. So from ψ'_1 , we obtain formula $t \geq 0 \rightarrow t + 1 > 0$, which is exactly formula ϕ'_1 (modulo free-variable renaming).

From this example, we see that the above rule ($[\alpha]$) can be directly applied to other languages (by just choosing a different set of program transitions) while rule $(op \ x := e)$ may not. It cannot be applied to, e.g., a Java statement $x := new \ C(...)$, which creates a new object of class C (cf. [7]). Throughout this paper (from Example 4.4 - 5.2, in Section 6.1 and 6.3), we show that how DLp can be adapted to different theories of programs through two instantiations of DLp: DLp-WP and DLp-FODL. Section 6.3 also displays the capability of DLp to derive multiple program models in a single framework.

Section 6.4 and Appendix C further give two instantiations: DLp-PL and DLp-SP separately to show that in DLp not only static properties (i.e. the properties holding on a state) can be expressed, but also more complex properties, like temporal properties and spatial properties.

The entire process of labeling and parameterization is fully introduced in Section 4. In Section 5.3, a proof system Pr_{dlp} for DLp is built.

2.2 Lifting Process and Compatibility of DLp

As a dynamic logic extended with the extra structure labels, DLp is compatible with the existing theories of dynamic logics in the sense that every inference rule for non-labeled dynamic formulas can be lifted as a rule for their labeled counterparts in DLp. Section 5.5 discusses this technique in detail, where we introduce a notion called “free labels” (Definition 5.8), and show that attaching a free label to a formula does not affect the validity of this formula (Theorem 5.10).

For instance, from the rule $(op \ x := e)$ above, one can obtain a sound lifted rule by attaching to each formula the label $\{x \mapsto t\}$:

$$\frac{\{x \mapsto t\} : \phi[x/e]}{\{x \mapsto t\} : [x := e]\phi} \quad (lf \ (x := e))$$

$\{x \mapsto t\}$ is free as t is a fresh variable. Trivially, replacing any free occurrence of variable x with variable t in the formulas $[x := e]\phi$ and $\phi[x/e]$ does not change their meanings. From the formula ψ_1 above, by applying the rule $(lf \ (x := e))$ on the part $\{x \mapsto t\} : [x := x + 1]x > 0$, we obtain the formula $\psi''_1 =_{df} t \geq 0 \rightarrow \{x \mapsto t\} : x + 1 > 0$. It is just ψ'_1 we have seen above if we replace x of $x + 1 > 0$ with its current value t .

Lifting process provides a type of flexibility by directly making use of the rules special in different domains. In Section 6.2, we illustrate in detail how this technique can be beneficial during derivations.

2.3 Cyclic Reasoning of DLp Formulas

In an ordinary deductive procedure we usually expect a finite proof tree. However, in the proof system Pr_{dlp} of 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, in the instantiated theory DLp-WP of DLp (cf. Example 4.4 - 5.2), a while program

$$W \equiv_{df} \text{while true do } x := x + 1 \text{ end}$$

proceeds infinitely as the following program transitions:

$$(W, \{x \mapsto 0\}) \longrightarrow (W, \{x \mapsto 1\}) \longrightarrow \dots$$

This yields the following infinite derivation branch in DLp when deriving, for example, a formula $\{x \mapsto 1\} : [W]\phi$:

$$\frac{\dots}{\{x \mapsto n\} : [W]\phi} \quad \vdots \quad \frac{\dots}{\{x \mapsto 2\} : [W]\phi} \quad \begin{matrix} ([\alpha]) \\ ([\alpha]) \end{matrix} \quad \frac{\{x \mapsto 2\} : [W]\phi}{\{x \mapsto 1\} : [W]\phi}.$$

To solve this problem, we propose a cyclic proof system for DLp (Section 5.4). Cyclic proof approach (cf. [16]) is a technique to admit a certain type of infinite deductions, called “cyclic proofs” (cf. Section 5.2). A cyclic proof is a finite proof tree augmented with some non-terminating leaf nodes, called “buds”, which are identical to some of their ancestors. Call a bud and one of its identical ancestors a “back-link”.

We propose a cyclic derivation approach special for DLp. This mainly consists of the following two steps.

In the first step, we construct a cyclic structure by identifying suitable buds and back-links, where the most critical work is to design the substitution rule (*Sub*) of labels (Definition 4.12). For example, by performing the substitution rule (*Sub*) given in Section 6.1 on the labels $\{x \mapsto 1\}$ and $\{x \mapsto t + 1\}$, we can obtain a cyclic derivation for the formula $\{x \mapsto 1\} : [W]\phi$ on the left below:

$$\frac{\frac{2 : \{x \mapsto t\} : [W]\phi}{\{x \mapsto t + 1\} : [W]\phi} \quad \begin{matrix} (Sub) \\ ([\alpha]) \end{matrix}}{\frac{1 : \{x \mapsto t\} : [W]\phi}{\{x \mapsto 1\} : [W]\phi} \quad (Sub)} \quad , \quad \frac{\frac{\{x \mapsto t\} : \langle W \rangle \phi}{\{x \mapsto t + 1\} : \langle W \rangle \phi} \quad \begin{matrix} (Sub) \\ ([\alpha]) \end{matrix}}{\frac{\{x \mapsto t\} : \langle W \rangle \phi}{\{x \mapsto 1\} : \langle W \rangle \phi} \quad (Sub)}$$

where node 2 is a bud and it back-links to node 1. t is a fresh variable w.r.t. x , W and ϕ . The label $\{x \mapsto 1\}$ equals to $\{x \mapsto t\}[1/t]$ (i.e. the label obtained by substituting t with 1) and the label $\{x \mapsto t + 1\}$ equals to $\{x \mapsto t\}[t + 1/t]$ (i.e. the label obtained by substituting t with expression $t + 1$).

However, not all cyclic structures are cyclic proofs. Consider the cyclic derivation on the above right for formula $\{x \mapsto 1\} : \langle W \rangle \phi$. According to the semantics of modality $\langle \cdot \rangle$ (cf. Section 4.1), $\{x \mapsto 1\} : \langle W \rangle \phi$ is invalid for any formula ϕ because W never terminates. Therefore, in the second step, we need to check whether these cyclic structures are legal cyclic proofs, where the key step is to define suitable “progressive derivation traces” special for system Pr_{dlp} (Definition 5.5).

We give two examples in Section 6.1 and Appendix B respectively to show how cyclic reasoning in DLp can be carried out and how we can benefit from the incremental reasoning of recursive programs by cyclic graphs. Especially,

the example given in Appendix B is an Esterel program, from which we can see that how cyclic reasoning based on operational semantics can prevent extra prior program transformations in synchronous languages.

2.4 Soundness and Completeness of DLp

We analyze and prove the soundness and completeness of DLp w.r.t. arbitrary programs and formulas under certain restriction conditions (Section 7).

The soundness of DLp states that a cyclic proof always leads to a valid conclusion. In Section 7.1, we prove it under a condition (Definition 7.2) that restricts how a program can terminate. Even though, the types of restricted programs is still very rich, enough to include all deterministic programming languages (cf. Section 7.1).

The idea of proving the soundness is by contradiction (cf. [15]). We assume the conclusion, e.g. $\{x \mapsto 1\} : [W]\phi$, is invalid, then it leads a sequence of invalid formulas in some proof branch, e.g. $\{x \mapsto 1\} : [W]\phi$, $\{x \mapsto t\} : [W]\phi$, $\{x \mapsto t + 1\} : [W]\phi, \dots$ in the above derivation. This sequence of invalid formulas then causes the violation of a well-founded set (Definition 7.4) of a type of metrics (Definition 7.8) that relate these invalid formulas. More details is given in Section 7.1.

The completeness of DLp states that for any valid labeled dynamic formula, there is a cyclic proof for it. We prove the completeness of DLp under a sufficient assumption about the so-called “loop programs” (Definition 7.13, 7.14). The main idea and the details of the proof are given in Section 7.2 and Appendix A. This completeness result is useful because the restriction condition is general: any instantiation of DLp is complete once its program models satisfy this condition.

2.5 Main Contributions & Content Structure

The main contributions of this paper can be summarized as follows:

- We define the syntax and semantics of DLp formulas.
- We construct a labeled proof system and develop a lifting process for DLp.
- We propose a cyclic proof approach tailored for DLp.
- We analyze and prove the soundness and completeness of DLp under certain conditions.

The rest of the paper is organized as follows. Section 3 gives a brief introduction to PDL and FODL, necessary for understanding the main content. In Section 4, we define the syntax and semantics of DLp. In Section 5, we propose a cyclic proof system for DLp. In Section 6, we analyze some case studies. Section 7 analyzes the soundness and completeness of DLp. Section 8 introduces related work, while Section 9 makes a conclusion and discusses about future work.

3 Prerequisites : PDL & FODL

In propositional dynamic logic (PDL) [23], the syntax of a formula ϕ is given by simultaneous inductions on both programs and formulas as follows in BNF form:

$$\begin{aligned}\alpha &=_{df} a \mid \phi? \mid \alpha; \alpha \mid \alpha \cup \alpha \mid \alpha^*, \\ \phi &=_{df} p \mid \neg\phi \mid \phi \wedge \phi \mid [\alpha]\phi.\end{aligned}$$

In the above definition, α is a regular expression with tests, often called a *regular program*. $a \in A$ is an atomic action of a symbolic set A . $\phi?$ is a test. If ϕ is true, then the program proceeds, otherwise, the program halts; $\alpha; \beta$ is a sequential program, meaning that after program α terminates, β proceeds. $\alpha \cup \beta$ is a choice program, it means that either α or β

proceeds non-deterministically. α^* is a star program, which means that α proceeds for an arbitrary number $n \geq 0$ of times. p is an atomic formula, including the boolean *true*. We call a formula having the modality $[\cdot]$ a *dynamic formula*. Intuitively, formula $[\alpha]\phi$ means that after all executions of program α , formula ϕ holds.

The semantics of PDL is given based on a Kripke structure (cf. [29]) $M = (S, \rightarrow, I)$, where S is a set of *worlds*; $\rightarrow \subseteq S \times A \times S$ is a set of transitions labeled by atomic programs; $I : P \rightarrow \mathcal{P}(S)$ interprets each atomic PDL formula of set P to a set of worlds.

Given a Kripke structure M , the semantics of PDL is based on the denotational semantics $[\cdot]$ of regular programs, given as a satisfaction relation $M, w \models \phi$ between a world w and a PDL formula ϕ . It is defined as follows by simultaneous inductions on both programs and formulas:

- a. $[[a]] =_{df} \{(w, w') \mid w \xrightarrow{a} w' \text{ on } M\};$
 - b. $[[\phi?]] =_{df} \{(w, w) \mid M, w \models \phi\};$
 - c. $[[\alpha; \beta]] =_{df} \{(w, w') \mid \exists w''. (w, w'') \in [[\alpha]] \wedge (w'', w') \in [[\beta]]\};$
 - d. $[[\alpha \cup \beta]] =_{df} [[\alpha]] \cup [[\beta]];$
 - e. $[[\alpha^*]] =_{df} \bigcup_{n=0}^{\infty} [[\alpha^n]]$, where $\alpha^0 =_{df} \text{true?}$, $\alpha^n =_{df} \alpha; \alpha^{n-1}$ for any $n \geq 1$.
1. $M, w \models p$, if $w \in I(p)$;
 2. $M, w \models \neg\phi$, if $M, w \not\models \phi$;
 3. $M, w \models \phi \wedge \psi$, if $M, w \models \phi$ and $M, w \models \psi$;
 4. $M, w \models [\alpha]\phi$, if for all $(w, w') \in [[\alpha]]$, $M, w' \models \phi$.

PDL studies the formulas that are valid w.r.t. all Kripke structures. Its proof system is complete (cf. [29]).

First-order dynamic logic (FODL) [49] is obtained from PDL by specializing the atomic actions a and atomic formulas p in PDL in some special domains. In FODL, an atomic action is an assignment of the form $x := e$, where $x \in \text{Var}_{fodl}$ is a variable and e is an expression. Usually, we consider e as an arithmetical expression of integer domain \mathbb{Z} , e.g., $x + 5$ and $x - 2 * y$, where $x, y \in \text{Var}_{fodl}$. $+, -, *, /$ are the usual arithmetical operators. An atomic formula is an arithmetical relation $e_1 \bowtie e_2$ with $\bowtie \in \{=, <, \leq, >, \geq\}$, such as $x + 5 < 0$ and $x - 2 * y = 1$. The non-dynamic formulas in FODL are thus the usual arithmetical first-order formulas linked by the logical connectives \neg, \wedge and the quantifier \forall .

In FODL, a *state* $w : \text{Var}_{fodl} \rightarrow \mathbb{Z}$ maps each variable to an integer. For an expression e , $w(e)$ returns the value obtained by replacing all the free occurrences of each variable x in e with the value $w(x)$. The Kripke structure $M_{fodl} = (S_{fodl}, \xrightarrow{x:=e}, I_{fodl})$ of FODL is defined such that S_{fodl} is the set of all states w.r.t. Var_{fodl} and \mathbb{Z} . For each assignment $x := e$, $w \xrightarrow{x:=e} w'$ is a relation on M_{fodl} iff $w' = w[x \mapsto e]$, where $w[x \mapsto e]$ returns a state that maps x to value $w(e)$ and maps other variables to the value the same as w . I_{fodl} interprets each atomic FODL formula as the set of states in which it is satisfied. Or formally, for a state $w \in I_{fodl}(e_1 \bowtie e_2)$, $w(e_1) \bowtie w(e_2)$ is true. Based on these, the semantics of FODL can be defined in a similar way as shown above. One can refer to [29] for a more formal definition of FODL.

FODL forms the language basis of many existing dynamic-logic theories [7, 8, 22, 35, 43–45, 62, 63] as mentioned in Section 1. Some are the extensions of FODL by adding new primitives, while the others can be expressed by FODL. For example, for the traditional while programs α :

$$\alpha =_{df} x := e \mid \alpha; \alpha \mid \text{if } \phi \text{ then } \alpha \text{ else } \beta \text{ end} \mid \text{while } \phi \text{ do } \alpha \text{ end},$$

their special statements can be captured by FODL as follows (cf. [29]):

$$\begin{aligned} \text{if } \phi \text{ then } \alpha \text{ else } \beta \text{ end} &=_{df} \phi?; \alpha \cup \neg\phi?; \beta, \\ \text{while } \phi \text{ do } \alpha \text{ end} &=_{df} (\phi?; \alpha)^*; \neg\phi?. \end{aligned}$$

4 Dynamic Logic DLp

4.1 Syntax and Semantics of DLp Formulas

The theory of DLp extends PDL by permitting the program α and formula ϕ in modalities $[\alpha]\phi$ to take arbitrary forms, only subject to some restriction conditions when discussing its soundness and completeness in Section 7.

In the relations defined in this section, we use \cdot to express an ignored object whose content does not really matter. For example, we may write $w \rightarrow \cdot$, $w \xrightarrow{\alpha/\cdot} w'$, $(\alpha, \sigma) \rightarrow (\alpha', \cdot)$ and so on.

Definition 4.1 (Programs & Formulas). In DLp we assume two pre-defined disjoint sets \mathbf{P} and \mathbf{F} . \mathbf{P} is a set of programs, in which we distinguish a special program $\downarrow \in \mathbf{P}$ called the “terminal program”. \mathbf{F} is a set of formulas.

Definition 4.2 (DLp Formulas). A dynamic logical formula ϕ w.r.t. the parameters \mathbf{P} and \mathbf{F} , called a “parameterized dynamic logic” (DLp) formula, is defined as follows in BNF form:

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

where $F \in \mathbf{F}$, $\alpha \in \mathbf{P}$; $[\cdot]$ is a new operator that does not appear in any formula of \mathbf{F} .

We denote the set of DLp formulas as \mathfrak{F}_{dlp} .

A DLp formula is called a *dynamic formula* if it contains a modality $[\cdot]$ within it. Intuitively, formula $[\alpha]\phi$ means that after the terminations of 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 \rightarrow can be expressed by formulas with \neg and \wedge accordingly.

Following the convention of defining a dynamic logic (cf. [29]), we introduce a novel Kripke structure to capture the parameterized program behaviours in \mathbf{P} .

Definition 4.3 (Program-labeled Kripke Structures). A “program-labeled” Kripke (PLK) 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 \mathcal{S} is a set of worlds; $\longrightarrow \subseteq \mathcal{S} \times (\mathbf{P} \times \mathbf{P}) \times \mathcal{S}$ is a set of relations labeled by program pairs, in the form of $w_1 \xrightarrow{\alpha/\alpha'} w_2$ for some $w_1, w_2 \in \mathcal{S}$, $\alpha, \alpha' \in \mathbf{P}$; $\mathcal{I} : \mathbf{F} \rightarrow \mathcal{P}(\mathcal{S})$ is an interpretation of formulas in \mathbf{F} on the power set of worlds. Moreover, $K(\mathbf{P}, \mathbf{F})$ satisfies that $w \not\xrightarrow{\downarrow/\alpha} \cdot$ for any $w \in \mathcal{S}$ and $\alpha \in \mathbf{P}$.

Definition 4.3 differs from the Kripke structures M of PDL (Section 3) in the following aspects: (1) It introduces a program-labeled relation of the form: $w_1 \xrightarrow{\alpha/\alpha'} w_2$; (2) It introduces an additional condition for the terminal program \downarrow . The program-labeled relations describe programs’ transitional behaviours, which is usually captured by their operational semantics (as we see in Section 5.3). This is unlike the relations in M , where a relation only captures the behaviours of an atomic program. Intuitively, $w_1 \xrightarrow{\alpha/\alpha'} w_2$ means that from world w_1 , program α is transitioned to program α' , ending with world w_2 . The condition for \downarrow exactly captures the meaning of program termination.

Below in this paper, **our discussion is always based on an assumed PLK structure namely $\mathcal{K}(\mathbf{P}, \mathbf{F}) = (\mathcal{S}, \longrightarrow, \mathcal{I})$.**

Starting from Example 4.4 below, through Example 4.9, 4.10 and 5.2 we gradually instantiate the theory DLp in the setting of the special theory FODL, where we restrict the program models of FODL to a simpler one – the while programs. To do this, we give explicit definitions for the parameters $\mathbf{P}, \mathbf{F}, \mathbf{L}, \mathbf{M}$ and the proof system \mathbf{Pr}_{op} for the operational semantics in DLp (\mathbf{L}, \mathbf{M} and \mathbf{Pr}_{op} are introduced below). The logical theory after instantiated is called DLp-WP.

Example 4.4 (An Instantiation of Programs and Formulas). Consider instantiating \mathbf{P} by the set of while programs defined in Section 3, namely \mathbf{P}_W . Consider a program WP in \mathbf{P}_W :

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

Given an initial value of variables n and s , program WP computes the sum from n to 1 stored in the variable s . The PLK structure $\mathcal{K}_W = (\mathcal{S}_W, \longrightarrow_W, \mathcal{I}_W)$ of while programs satisfies that $\mathcal{S}_W = S_{fodl}$ and $\mathcal{I}_W = I_{fodl}$. \longrightarrow_W describes the transitional behaviours of while programs, captured by the operational semantics of \mathbf{P}_W (see Table 1 in Section 5). \longrightarrow_W coincides with \longrightarrow_{fodl} on atomic programs. For example, a relation $w \xrightarrow{x:=x+1/\downarrow} w[x \mapsto w(x) + 1]$ is on \mathcal{K}_W iff a relation $w \xrightarrow{x:=x+1} w[x \mapsto w(x) + 1]$ is on M_{fodl} .

We instantiate \mathbf{F} by the arithmetic first-order formulas in integer domain \mathbb{Z} (Section 3), namely \mathbf{F}_{af0} .

Definition 4.5 (Execution Paths). An “execution path” on \mathcal{K} is a finite sequence of relations on \longrightarrow : $w_1 \xrightarrow{\alpha_1/\beta_1} \dots \xrightarrow{\alpha_n/\beta_n} w_{n+1}$ ($n \geq 0$) satisfying that $\beta_n \in \{\downarrow\}$, and $\beta_i = \alpha_{i+1} \notin \{\downarrow\}$ for all $1 \leq i < n$.

In Definition 4.5, the execution path is sometimes simply written as a sequence of worlds: $w_1 \dots w_{n+1}$. When $n = 0$, the execution path is a single world w_1 (without any relations on \longrightarrow).

Given a path tr , we often use tr_b and tr_e to denote its first and last element (if there is). For two paths $tr_1 =_{df} w_1 \dots w_n$ and $tr_2 =_{df} w'_1 w'_2 \dots w'_m \dots$ ($n, m \geq 0$), tr_1 is finite. The *concatenation* $tr_1 \cdot tr_2$ is defined as the path: $w_1 \dots w_n w'_1 \dots w'_m \dots$, if $w_n = w'_1$ holds. We use relation $tr_1 \preceq_s tr_2$ to represent that tr_1 is a suffix of tr_2 . Write $tr_1 \prec_s tr_2$ if tr_1 is a proper suffix of tr_2 .

Definition 4.6 (Semantics of DLp Formulas). Given a DLp formula ϕ , the satisfaction of ϕ by a world $w \in \mathcal{S}$ under \mathcal{K} , denoted by $\mathcal{K}, w \models \phi$, is inductively defined as follows:

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

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

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

Compared to the semantics of PDL (Section 3), where to capture the semantics of a regular program one only has to record the beginning and ending worlds, we have to record the whole execution path from the beginning to the ending node. This is because the semantics of a program in DLp is operational, not denotational defined according to its syntactic structures.

Example 4.7 (DLp Specifications). A property of program WP (Example 4.4) 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 the execution of WP , s equals to $((N + 1)N)/2$, which is the sum of $1 + 2 + \dots + N$, with N a free variable in \mathbb{Z} . We prove an equivalent labeled version of this formula in DLp in Section 6.1.

4.2 Labeled DLp Formulas

Definition 4.8 (Labels & Label Mappings). In DLp, we assume two pre-defined sets \mathbf{L} and \mathbf{M} . \mathbf{L} is a set of “labels”. $\mathbf{M} \subseteq \mathbf{L} \rightarrow \mathcal{S}$ is a set of label mappings. Each mapping $\mathbf{m} \in \mathbf{M}$ maps a label of \mathbf{L} to a world of set \mathcal{S} .

Labels usually denote the explicit data structures that capture program configurations, for example, storage, heaps, substitutions, etc. Label mappings associate labels with the worlds, acting as the semantic functions of the labels.

Example 4.9 (An Instantiation of Labels). In while programs, we consider a type of labels namely \mathbf{L}_W that capture the meaning of the program configurations of the form:

$$\{x_1 \mapsto e_1, \dots, x_n \mapsto e_n\} \ (n \geq 0)$$

where each variable $x_i \in \text{Var}_W$ stores a unique value of arithmetic expression e_i ($1 \leq i \leq n$). To make it simple, we restrict that variables x_1, \dots, x_n must appear in the discussed programs and any free variable in e_1, \dots, e_n cannot be any of x_1, \dots, x_n . For any expression e , $\sigma(e)$ returns an expression by replacing each free variable x_i in e with its expression e_i in σ . A “configuration update” σ_e^x returns a configuration that stores variable x as a value of expression $\sigma(e)$, while storing other variables as the same value as σ .

For example, in program WP (Example 4.4), $\{n \mapsto N, s \mapsto 0\}$ can be a configuration that maps n to value N (as a free variable) and s to 0.

Example 4.10 (An Instantiation of Label Mappings). In while programs, we consider a set \mathbf{M}_W of label mappings. $\mathbf{M}_W \subseteq \mathbf{L}_W \rightarrow \mathcal{S}_W$. Each label mapping in \mathbf{M}_W is associated to a world, denoted by \mathbf{m}_w for some $w \in \mathcal{S}_W$. Given a configuration σ that captures the meaning of $\{x_1 \mapsto e_1, \dots, x_n \mapsto e_n\}$ ($n \geq 1$), $\mathbf{m}_w(\sigma)$ is defined as a world such that

- (1) $\mathbf{m}_w(\sigma)(x_i) = w(e_i)$ for each x_i ($1 \leq i \leq n$);
- (2) $\mathbf{m}_w(\sigma)(y) = w(y)$ for other variable $y \in \text{Var}_W$.

Where as explained in Section 3, $w(e_i)$ returns a value by substituting each free occurrences of variable x of e_i with the value $w(x)$.

For example, let w be a world with $w(N) = 5$, then we have $\mathbf{m}_w(\{n \mapsto N, s \mapsto 0\})(n) = w(N) = 5$, $\mathbf{m}_w(\{n \mapsto N, s \mapsto 0\})(s) = 0$, and $\mathbf{m}_w(\{n \mapsto N, s \mapsto 0\})(y) = w(y)$ for any other variable $y \notin \{n, s\}$.

Definition 4.11 (Labeled DLp Formulas). A “labeled formula” in DLp belongs to one of the following types of formulas defined as follows:

$$\phi =_{df} \sigma : \psi \mid (\alpha, \sigma) \longrightarrow (\alpha', \sigma') \mid \sigma \Downarrow \alpha,$$

where $\sigma, \sigma' \in \mathbf{L}$, $\alpha, \alpha' \in \mathbf{P}$, $\psi \in \mathfrak{F}_{dlp}$.

We use \mathfrak{F}_{dlp} , \mathfrak{F}_{pt} and \mathfrak{F}_{ter} to represent the sets of labeled formulas of the forms: $\sigma : \psi, (\alpha, \sigma) \longrightarrow (\alpha', \sigma')$ and $\sigma \Downarrow \alpha$ respectively. We often use τ to represent a labeled formula in $\mathfrak{F}_{dlp} \cup \mathfrak{F}_{pt} \cup \mathfrak{F}_{ter}$.

In DLp, relation $(\alpha, \sigma) \longrightarrow (\alpha', \sigma')$ is called a *program transition*, which indicates an execution from a so-called *program state* (α, σ) to another program state (α', σ') . Relation $\sigma \Downarrow \alpha$ is called a *program termination*, which describes the termination of a program α under a label σ .

Definition 4.12 (Substitution of Labels). A “substitution” $\eta : \mathbf{L} \rightarrow \mathbf{L}$ is a function on \mathbf{L} satisfying that for any label mapping $\mathbf{m} \in \mathbf{M}$, there exists a label mapping $\mathbf{m}'(\mathbf{m}, \eta)$ (determined only by \mathbf{m} and η) such that $\mathbf{m}'(\sigma) = \mathbf{m}(\eta(\sigma))$ for all labels $\sigma \in \mathbf{L}$.

Definition 4.12 is used in the rule (Sub) (Table 2) and in the proof of soundness of rules Pr_{dlp} and the cyclic proof system of DLp.

Definition 4.13 (Semantics of Labeled DLp Formulas). Given a label mapping $\mathbf{m} \in \mathbf{M}$ and a labeled formula $\tau \in \mathcal{F}_{dlp} \cup \mathcal{F}_{pt} \cup \mathcal{F}_{ter}$, the satisfaction relation $\mathcal{K}, \mathbf{M}, \mathbf{m} \models \tau$ of a formula τ by \mathcal{K}, \mathbf{M} and \mathbf{m} (simply $\mathbf{m} \models \tau$) is defined as follows according to the different cases of τ :

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

A formula $\tau \in \mathcal{F}_{dlp} \cup \mathcal{F}_{pt} \cup \mathcal{F}_{ter}$ is *valid*, denoted by $\mathcal{K} \models \tau$ (or simply $\models \tau$), if $\mathcal{K}, \mathbf{M}, \mathbf{m} \models \tau$ for all $\mathbf{m} \in \mathbf{M}$.

5 A Cyclic Proof System for DLp

We propose a cyclic proof system for DLp. We firstly propose a labeled proof system Pr_{dlp} to support reasoning based on operational semantics (Section 5.3). Then we construct a cyclic proof structure for system Pr_{dlp} , which support deriving infinite proof trees under certain conditions (Section 5.4). Section 5.1 and 5.2 introduce the notions of labeled sequent calculus and cyclic proof respectively.

5.1 Labeled Sequent Calculus

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 *labeled sequent* is a sequent in which each formula is a labeled formula in $\mathcal{F}_{dlp} \cup \mathcal{F}_{pt} \cup \mathcal{F}_{ter}$.

According to the meaning of a sequent above, a labeled sequent $\Gamma \Rightarrow \Delta$ is *valid*, if for every $\mathbf{m} \in \mathbf{M}$, $\mathbf{m} \models \tau$ for all $\tau \in \Gamma$ implies $\mathbf{m} \models \tau'$ for some $\tau' \in \Delta$. For a multi-set Γ of formulas, we write $\mathbf{m} \models \Gamma$ to mean that $\mathbf{m} \models \tau$ for all $\tau \in \Gamma$.

5.2 Proofs & Preproofs & Cyclic Proofs

An *inference rule* is of the form $\frac{\nu_1 \quad \dots \quad \nu_n}{\nu}$, where each of ν, ν_i ($1 \leq i \leq n$) is also called a *node*. Each of ν_1, \dots, ν_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 ν_1, \dots, ν_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 .

In this paper, we use a double-lined inference form:

$$\frac{\phi_1 \quad \dots \quad \phi_n}{\phi}$$

to represent both rules

$$\frac{\Gamma \Rightarrow \phi_1, \Delta \quad \dots \quad \Gamma \Rightarrow \phi_n, \Delta}{\Gamma \Rightarrow \phi, \Delta} \quad \text{and} \quad \frac{\Gamma, \phi_1 \Rightarrow \Delta \quad \dots \quad \Gamma, \phi_n \Rightarrow \Delta}{\Gamma, \phi \Rightarrow \Delta},$$

provided any context Γ and Δ .

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. [16]), 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. A bud and one of its identical ancestors together is called a *back-link*. A *derivation path* in a preproof is an infinite sequence of nodes $v_1 v_2 \dots v_m \dots$ ($m \geq 1$) starting from the root node v_1 , where each node pair (v_i, v_{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 5.5 later for DLp), over every derivation path.

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

5.3 A Proof System for DLp

The labeled proof system Pr_{dlp} for DLp consists of two parts: a finite set Pr_{ldlp} of *kernel rules* for deriving DLp formulas, as listed in Table 2, and a finite set \mathbf{Pr}_{op} of the rules, as a parameter of DLp, for capturing the operational semantics of the programs in \mathbf{P} . The rules in Pr_{ldlp} do not rely on the explicit structures of the programs in \mathbf{P} , but depend on the derivations of program transitions according to \mathbf{Pr}_{op} as their side-conditions. Pr_{dlp} provides a universal logical framework but modulo different program theories: i.e., \mathbf{Pr}_{op} .

The content of \mathbf{Pr}_{op} depends on the explicit structures of the programs in \mathbf{P} and can vary from case to case. Through the rules in \mathbf{Pr}_{op} , in the system Pr_{dlp} we can derive the program transitions \mathfrak{F}_{pt} and terminations \mathfrak{F}_{ter} respectively. However, to coincide with the PLK structure $\mathcal{K}(\mathbf{P}, \mathbf{F})$ as well as to fulfill the soundness and completeness of DLp, we need to make the following assumptions on \mathbf{Pr}_{op} as described in the next definition.

Definition 5.1 (Assumptions on Set \mathbf{Pr}_{op} (Pr_{dlp})). The parameter \mathbf{Pr}_{op} (as a part of Pr_{dlp}) satisfies that

1. *Coincidence with $\mathcal{K}(\mathbf{P}, \mathbf{F})$.* For any $\mathbf{m} \in \mathbf{M}$, Γ such that $\mathbf{m} \models \Gamma$, and for any $\sigma \in \mathbf{L}$, if $\mathbf{m}(\sigma) \xrightarrow{\alpha/\alpha'} w$ is a relation on \mathcal{K} for some $\alpha, \alpha' \in \mathbf{P}$ and $w \in \mathcal{S}$, then there exists a label $\sigma' \in \mathbf{L}$ such that $\mathbf{m}(\sigma') = w$ and $\models (\Gamma \Rightarrow (\alpha, \sigma) \longrightarrow (\alpha', \sigma'))$.
2. *Soundness w.r.t. \mathfrak{F}_{pt} and \mathfrak{F}_{ter} .* For any derivation $Pr_{dlp} \vdash (\Gamma \Rightarrow (\alpha, \sigma) \longrightarrow (\alpha', \sigma'))$ (resp. $Pr_{dlp} \vdash (\Gamma \Rightarrow \sigma \Downarrow \alpha)$) in system Pr_{dlp} , $\models (\Gamma \Rightarrow (\alpha, \sigma) \longrightarrow (\alpha', \sigma'))$ (resp. $\models (\Gamma \Rightarrow \sigma \Downarrow \alpha)$).
3. *Completeness w.r.t. \mathfrak{F}_{pt} and \mathfrak{F}_{ter} .* For any valid sequent $\Gamma \Rightarrow (\alpha, \sigma) \longrightarrow (\alpha', \sigma')$ (resp. $\Gamma \Rightarrow \sigma \Downarrow \alpha$), $Pr_{dlp} \vdash (\Gamma \Rightarrow (\alpha, \sigma) \longrightarrow (\alpha', \sigma'))$ (resp. $Pr_{dlp} \vdash (\Gamma \Rightarrow \sigma \Downarrow \alpha)$).
4. *Simple Conditions.* For any valid sequent $\Gamma \Rightarrow (\alpha, \sigma) \longrightarrow (\alpha', \sigma')$, there is a context Γ' in which all formulas are non-dynamic ones, such that $\models (\Gamma' \Rightarrow (\alpha, \sigma) \longrightarrow (\alpha', \sigma'))$ and $\models (\Gamma' \Rightarrow \Gamma')$.

Intuitively, the coincidence with $\mathcal{K}(\mathbf{L}, \mathbf{F})$ says that the relations on \mathcal{K} between worlds (, once their starting nodes can be captured by labels,) can be expressed by the program transitions as labeled formulas of DLp. It is essential for proving the soundness of the rules in Pr_{ldlp} (see the proof of Theorem 5.3 in Appendix A). The soundness and completeness w.r.t. \mathfrak{F}_{pt} and \mathfrak{F}_{ter} are required for proving Theorem 5.3, 7.3 and 7.15. The soundness condition says that the proof system \mathbf{Pr}_{op} derives no more than the program transitions that are valid on \mathcal{K} , while the completeness condition says that \mathbf{Pr}_{op} is enough for deriving all those behaviours. The assumption “simple conditions” is needed in our proof of the conditional completeness of DLp (Theorem 7.15). It means that the conditions for program transitions do not depend on the behaviours of other programs. This is usually the case for the program languages in practice. However, this

$\frac{}{\Gamma \Rightarrow (x := e, \sigma) \rightarrow (\downarrow, \sigma_e^x), \Delta} \quad (x:=e)$	
$\frac{}{\Gamma \Rightarrow (\alpha_1, \sigma) \rightarrow (\downarrow, \sigma'), \Delta} \quad (;\downarrow)$	$\frac{\Gamma \Rightarrow (\alpha_1, \sigma) \rightarrow (\alpha'_1, \sigma'), \Delta}{\Gamma \Rightarrow (\alpha_1; \alpha_2, \sigma) \rightarrow (\alpha'_1; \alpha_2, \sigma'), \Delta} \quad (;)$
$\frac{\Gamma \Rightarrow (\alpha_1, \sigma) \rightarrow (\downarrow, \sigma'), \Delta}{\Gamma \Rightarrow (\alpha_1; \alpha_2, \sigma) \rightarrow (\alpha_2, \sigma'), \Delta} \quad (;\downarrow)$	$\frac{\Gamma \Rightarrow (\alpha_1, \sigma) \rightarrow (\alpha'_1, \sigma'), \Delta \quad \Gamma \Rightarrow \sigma : \phi, \Delta}{\Gamma \Rightarrow (\text{if } \phi \text{ then } \alpha_1 \text{ else } \alpha_2 \text{ end}, \sigma) \rightarrow (\alpha'_1, \sigma'), \Delta} \quad (\text{ite1})$
$\frac{\Gamma \Rightarrow (\alpha_2, \sigma) \rightarrow (\alpha'_2, \sigma'), \Delta \quad \Gamma \Rightarrow \sigma : \neg \phi, \Delta}{\Gamma \Rightarrow (\text{if } \phi \text{ then } \alpha_1 \text{ else } \alpha_2 \text{ end}, \sigma) \rightarrow (\alpha'_2, \sigma'), \Delta} \quad (\text{ite2})$	
$\frac{\Gamma, \sigma : \phi \Rightarrow (\alpha, \sigma) \rightarrow (\alpha', \sigma'), \Delta \quad \Gamma \Rightarrow \phi : \sigma, \Delta}{\Gamma \Rightarrow (\text{while } \phi \text{ do } \alpha \text{ end}, \sigma) \rightarrow (\alpha', \text{while } \phi \text{ do } \alpha \text{ end}, \sigma'), \Delta} \quad (\text{wh1})$	
$\frac{\Gamma, \sigma : \phi \Rightarrow (\alpha, \sigma) \rightarrow (\downarrow, \sigma'), \Delta \quad \Gamma \Rightarrow \sigma : \phi, \Delta}{\Gamma \Rightarrow (\text{while } \phi \text{ do } \alpha \text{ end}, \sigma) \rightarrow (\text{while } \phi \text{ do } \alpha \text{ end}, \sigma'), \Delta} \quad (\text{wh1}\downarrow)$	
$\frac{\Gamma \Rightarrow \sigma : \neg \phi, \Delta}{\Gamma \Rightarrow (\text{while } \phi \text{ do } \alpha \text{ end}, \sigma) \rightarrow (\downarrow, \sigma), \Delta} \quad (\text{wh2})$	

Table 1. Partial Rules of $(\mathbf{Pr}_{op})_W$ for Program Transitions of While Programs

$\frac{\{\Gamma \Rightarrow \sigma' : [\alpha']\phi, \Delta\}_{(\alpha', \sigma') \in \Phi}}{\Gamma \Rightarrow \sigma : [\alpha]\phi, \Delta} \quad 1 \text{ } ([\alpha]R), \text{ where } \Phi =_{df} \{(\alpha', \sigma') \mid Pr_{dlp} \vdash (\Gamma \Rightarrow (\alpha, \sigma) \rightarrow (\alpha', \sigma'), \Delta)\}$	
$\frac{\Gamma, \sigma' : [\alpha']\phi \Rightarrow \Delta}{\Gamma, \sigma : [\alpha]\phi \Rightarrow \Delta} \quad 1 \text{ } ([\alpha]L), \text{ if } Pr_{dlp} \vdash (\Gamma \Rightarrow (\alpha, \sigma) \rightarrow (\alpha', \sigma'), \Delta)$	
$\frac{\sigma : \phi}{\sigma : [\downarrow]\phi} \quad ([\downarrow])$	$\frac{}{\Gamma \Rightarrow \Delta} \quad 2 \text{ } (Ter)$
$\frac{\Gamma \Rightarrow \Delta}{Sub(\Gamma) \Rightarrow Sub(\Delta)} \quad 3 \text{ } (Sub)$	$\frac{}{\Gamma, \sigma : \phi \Rightarrow \sigma : \phi, \Delta} \quad (ax)$
$\frac{\Gamma \Rightarrow \sigma : \phi \quad \Gamma, \sigma : \phi \Rightarrow \Delta}{\Gamma \Rightarrow \Delta} \quad (Cut)$	$\frac{\Gamma \Rightarrow \Delta}{\Gamma \Rightarrow \sigma : \phi, \Delta} \quad (WkR)$
$\frac{\Gamma \Rightarrow \Delta}{\Gamma, \sigma : \phi \Rightarrow \Delta} \quad (WkL)$	$\frac{\sigma : \phi, \sigma : \phi}{\sigma : \phi} \quad (Con)$
$\frac{\Gamma, \sigma : \phi \Rightarrow \Delta}{\Gamma \Rightarrow \sigma : \neg \phi, \Delta} \quad (\neg R)$	$\frac{\Gamma \Rightarrow \sigma : \phi, \Delta}{\Gamma, \sigma : \neg \phi \Rightarrow \Delta} \quad (\neg L)$
$\frac{\Gamma \Rightarrow \sigma : \phi, \Delta \quad \Gamma \Rightarrow \sigma : \psi, \Delta}{\Gamma \Rightarrow \sigma : \phi \wedge \psi, \Delta} \quad (\wedge R)$	$\frac{\Gamma, \sigma : \phi, \sigma : \psi \Rightarrow \Delta}{\Gamma, \sigma : \phi \wedge \psi \Rightarrow \Delta} \quad (\wedge L)$

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 4.12.

Table 2. Rules Pr_{dlp} for the Proof System of DLp

assumption makes DLp unable to instantiate the dynamic logics with “rich tests” (cf. [29]), where a test can be a dynamic formula itself, for example a test $[x := e]\phi?$.

Example 5.2 (An Instantiation of \mathbf{Pr}_{op}). Table 1 displays a set of partial rules of $(\mathbf{Pr}_{op})_W$ for describing the operational semantics of while programs, where we omit the rules for deriving program terminations. In Table 1, σ_e^x is defined in Example 4.9.

In practice, we usually think that the rules \mathbf{Pr}_{op} faithfully commit the operational semantics of the programs. So in this manner we simply trust the assumptions 1, 2, 3 of Definition 5.1 without proving.

Through the rules in Pr_{dlp} , a labeled DLp 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.

Rules $([\alpha]R)$ and $([\alpha]L)$ reason about dynamic parts of labeled DLp formulas. Both rules rely on side deductions: “ $Pr_{dlp} \vdash (\Gamma \Rightarrow (\alpha, \sigma) \rightarrow (\alpha', \sigma'), \Delta)$ ” as sub-proof procedures of program transitions. In rule $([\alpha]R)$, $\{\dots\}_{(\alpha', \sigma') \in \Phi}$ represents the collection of premises for all program states $(\alpha', \sigma') \in \Phi$. By the finiteness of system Pr_{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 terminal one \downarrow . Its soundness is straightforward by the semantics of \downarrow in Definition 4.3.

Rule (Ter) indicates that one proof branch terminates when a sequent $\Gamma \Rightarrow \Delta$ is valid in which all labeled formulas are non-dynamic ones.

Rule (Sub) describes a specialization process for labeled dynamic formulas. For a set A of labeled formulas, $Sub(A) =_{df} \{Sub(\tau) \mid \tau \in A\}$, with Sub a substitution (Definition 4.12). 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 5.4). See Section 6.1 for more details.

Rules from (ax) to $(\wedge L)$ are the “labeled versions” of the corresponding rules inherited from traditional first-order logic. Their meanings are classical and we omit their discussions here.

THEOREM 5.3. *Each rule from Pr_{dlp} in Table 2 is sound.*

Following the above explanations, Theorem 5.3 can be proved according to the semantics of labeled DLp formulas under the assumption of Definition 5.1. See Appendix A for more details.

5.4 Construction of a Cyclic Proof Structure for DLp

We build a cyclic labeled proof system for DLp, in order to recognize and admit potential infinite derivations as the example shown in Section 2. Based on the notion of preproofs (Section 5.2), we build a cyclic proof structure for system Pr_{dlp} , where the key part is to introduce the notion of progressive derivation traces in DLp (Definition 5.5).

Next we first introduce the notion of progressive derivation traces for DLp, then we define the cyclic proof structure for DLp as a special case of the notion already given in Section 5.2.

Definition 5.4 (Derivation Traces). A “derivation trace” over a derivation path $\mu_1\mu_2\dots\mu_k v_1 v_2\dots v_m\dots$ ($k \geq 0, m \geq 1$) is an infinite sequence $\tau_1\tau_2\dots\tau_m\dots$ of formulas with each formula τ_i ($1 \leq i \leq m$) in node v_i . Each CP pair (τ_i, τ_{i+1}) ($i \geq 1$) of derivation (v_i, v_{i+1}) satisfies special conditions as follows according to (v_i, v_{i+1}) being the different instances of rules from Pr_{dlp} :

1. If (v_i, v_{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 (v_i, v_{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 (v_i, v_{i+1}) is an instance of other rules, then $\tau_i = \tau_{i+1}$.

Below an expression $n :: O$ means that we use name n to denote the object O .

Definition 5.5 (Progressive Derivation Traces). In a preproof of system Pr_{dlp} , given a derivation trace $\tau_1\tau_2\dots\tau_m\dots$ over a derivation path $\dots v_1 v_2\dots v_m\dots$ ($m \geq 1$) starting from τ_1 in node v_1 , a CP pair (τ_i, τ_{i+1}) ($1 \leq i \leq m$) of derivation (v_i, v_{i+1})

is called a “progressive step”, if (τ_i, τ_{i+1}) is the following CP pair of an instance of rule $([\alpha]R)$:

$$\frac{\dots \quad v_{i+1} :: (\Gamma \Rightarrow \tau_{i+1} :: (\sigma' : [\alpha']\phi), \Delta) \quad \dots}{v_i :: (\Gamma \Rightarrow \tau_i :: (\sigma : [\alpha]\phi), \Delta),} \quad ([\alpha]R);$$

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

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

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

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 7.9 (see Appendix A).

Theorem 5.3 shows that each rule of Pr_{dlp} is sound. But that does not mean that the proof system Pr_{dlp} is sound, because we need to make sure that each cyclic proof also leads to a valid conclusion. The soundness of the system Pr_{dlp} is fully discussed in Section 7.

5.5 Lifting Rules From Dynamic Logic Theories

We introduce a technique of lifting the rules from particular dynamic-logic theories, e.g. FODL [49], to the labeled ones in DLp. It makes possible for embedding existing dynamic-logic theories into DLp without losing their abilities of deriving based on programs’ syntactic structures. This in turn facilitates deriving DLp formulas in particular program domains by making use of special inference rules. Below, we propose a lifting process for general inference rules under a certain condition of labels (Proposition 5.10). One example of the applications of this technique is given in Section 6.2.

We first introduce the concept of *free labels*, as a sufficient condition for the labels to carry out the lifting. Then we introduce the lifting process as Proposition 5.10.

Definition 5.6 (Effect Equivalence). Two worlds $w, w' \in \mathcal{S}$ “have the same effect” w.r.t a set of unlabeled formulas $A \subseteq \mathfrak{F}_{dlp}$, denoted by $w =_A w'$, if for any $\phi \in A$, $w \models \phi$ iff $w' \models \phi$.

Example 5.7. In DLp-WP, consider two worlds $w, w' \in \mathcal{S}_W$ (which are two mappings from Var_{fodl} to \mathbb{Z}) satisfying that $w(x) = 1, w(y) = 1$ and $w'(x) = 2, w'(y) = 1$ for some $x, y \in Var_{fodl}$. Let formula $\phi = (x + y > 1) \in \mathcal{F}_{afo}$. Then $w =_{\{\phi\}} w'$, although $w \neq w'$.

Definition 5.8 (Free Labels). A label $\sigma \in \mathcal{L}$ is called “free” w.r.t a set A of formulas if for any world $w \in \mathcal{S}$, there exists a label mapping $\mathfrak{m} \in \mathcal{M}$ such that

$$w =_A \mathfrak{m}(\sigma).$$

We denote the set of all free labels w.r.t. A as $free(\mathcal{L}, A)$.

Intuitively, the freedom of a label σ w.r.t. a set A of formulas means that σ is general enough so that it does not have an impact on the validity of the formulas in A .

Example 5.9. In DLp-WP, let $\sigma = \{x \mapsto t + 1\} \in \mathcal{L}_W$ (with t a variable). σ is free w.r.t. $\{\phi\}$ for $\phi = (x + y > 1)$ (Example 5.7). Because for any world $w \in \mathcal{S}_W$, let $w' =_{df} w[t \mapsto x - 1]$, then we have $w =_{\{\phi\}} \mathfrak{m}_{w'}(\sigma)$, since $w(x) = \mathfrak{m}_{w'}(\sigma)(x)$ and $w(y) = \mathfrak{m}_{w'}(\sigma)(y)$. On the other hand, let $\sigma' = \{x \mapsto 0, y \mapsto 0\}$, then σ' is not free w.r.t. $\{\phi\}$. Because for any $w \in \mathcal{S}_W$, $\mathfrak{m}_w(\sigma')(x) = \mathfrak{m}_w(\sigma')(y) = 0$, so $\mathfrak{m}_w(\sigma') \not\models \phi$.

We see that compared to σ , σ' is too “explicit” so that it affects the validity of formula ϕ .

For a set A of unlabeled formulas, we write $\sigma : A$ to mean the set of labeled formulas $\{\sigma : \phi \mid \phi \in A\}$.

PROPOSITION 5.10 (LIFTING PROCESS). *Given a sound rule of the form*

$$\frac{\Gamma_1 \Rightarrow \Delta_1 \quad \dots \quad \Gamma_n \Rightarrow \Delta_n}{\Gamma \Rightarrow \Delta}, \quad n \geq 1,$$

in which all formulas are unlabeled, then the rule

$$\frac{\sigma : \Gamma_1 \Rightarrow \sigma : \Delta_1 \quad \dots \quad \sigma : \Gamma_n \Rightarrow \sigma : \Delta_n}{\sigma : \Gamma \Rightarrow \sigma : \Delta}$$

is sound for any label $\sigma \in \text{free}(\mathbf{L}, \Gamma \cup \Delta \cup \Gamma_1 \cup \Delta_1 \cup \dots \cup \Gamma_n \cup \Delta_n)$.

Proposition 5.10 is proved in Appendix A based on the notion of free labels defined above.

6 Case Studies

In this section, we illustrate the potential usage of DLp by several instances.

We firstly show how labeled dynamic formulas in DLp can be derived according to the cyclic proof system proposed in Table 2. We give an example of deduction for a while program (Section 6.1). In Appendix B, we briefly introduce another instantiation of DLp for the synchronous language Esterel [12] and show a cyclic derivation of an Esterel program. The second example better highlights the advantages of DLp since the loop structures of some Esterel programs are implicit.

Secondly, we take the rules in FODL as examples to illustrate how to carry out rule lifting in DLp (Section 6.2). It demonstrates the compatibility of DLp to the existing dynamic-logic theories, allowing them to be reused in DLp. As we can see, this also helps increasing the efficiency of the derivations in DLp by adopting the compositional rules in special domains.

In Section 4 and 5, we have seen the instantiation theory DLp-WP of DLp. Section 6.3 and 6.4 further introduces more complex instantiations of DLp.

In Section 6.3, We embed FODL theory into DLp. This example shows the potential usefulness of DLp for different program models in practice, because FODL is the basic theory underlying many dynamic-logic variations (such as [7, 43, 44, 62]). By an example of reasoning about both while programs and regular programs at the same time, we also show the heterogeneity of DLp, that different program models can be easily compared with different operational semantics.

In Section 6.4, more complex encoding of labels and formulas are further displayed. We propose to encode a first-ordered version of process logic [28] into DLp. Process logic provides a logical framework for not just reasoning about program properties after the terminations, but also properties during the executions. From this example, we see that in DLp the labels and formulas can be more expressive than in traditional Hoare-style logics where only before-after properties can be reasoned about. In Appendix C, we give another example to show this by encoding separation logic [51] in DLp. It provides a novel way of reasoning about separation-logic formulas directly through symbolic executions.

6.1 A Cyclic Deduction of A While Program

We prove the property in Example 4.7 according to the rules in Table 2. This property can be captured by the following equivalent labeled sequent

$$v_1 =_{df} \sigma_1 : n \geq 0 \Rightarrow \sigma_1 : [WP](s = ((N + 1)N)/2),$$

$ \begin{array}{c} \frac{16}{15} \text{ (WkL)} \\ \frac{15}{14} \text{ (Sub)} \\ \frac{14}{11} \text{ (WkL)} \quad \frac{13}{12} \text{ (WkR)} \\ \frac{11}{10} \text{ (Cut)} \\ \frac{10}{9} \text{ ([}\alpha\text{]}R) \\ \frac{9}{5} \text{ ([}\alpha\text{]}R) \\ \frac{17}{3} \text{ (Ter)} \quad \frac{8}{7} \text{ ([}\downarrow\text{])} \\ \frac{7}{6} \text{ ([}\alpha\text{]}R) \\ \frac{2}{v_1:1} \text{ (Sub)} \quad \frac{4}{6} \text{ (}\vee\text{L)} \end{array} $		<p>Definitions of other symbols:</p> $ \begin{aligned} WP &=_{df} \{ \text{while } (n > 0) \text{ do } s := s + n ; n := n - 1 \text{ 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 \} \end{aligned} $	
1:	$\sigma_1 : n \geq 0$	\Rightarrow	$\sigma_1 : [\text{while } (n > 0) \text{ do } \alpha_1 \text{ end}] \phi_1$
2:	$\sigma_2 : n \geq 0$	\Rightarrow	$\sigma_2 : [\text{while } (n > 0) \text{ do } \alpha_1 \text{ end}] \phi_1$
3:	$\sigma_2 : n \geq 0$	\Rightarrow	$\sigma_2 : [\text{while } (n > 0) \text{ do } \alpha_1 \text{ end}] \phi_1, \sigma_2 : (n > 0 \vee n \leq 0)$
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	$\sigma_2 : [\text{while } (n > 0) \text{ do } \alpha_1 \text{ end}] \phi_1$
5:	$\sigma_2 : n \geq 0, \sigma_2 : n > 0$	\Rightarrow	$\sigma_2 : [\text{while } (n > 0) \text{ do } \alpha_1 \text{ end}] \phi_1$
9:	$\sigma_2 : n \geq 0, \sigma_2 : n > 0$	\Rightarrow	$\sigma_3 : [n := n - 1; \text{while } (n > 0) \text{ do } \alpha_1 \text{ end}] \phi_1$
10:	$\sigma_2 : n \geq 0, \sigma_2 : n > 0$	\Rightarrow	$\sigma_4 : [\text{while } (n > 0) \text{ do } \alpha_1 \text{ end}] \phi_1$
11:	$\sigma_2 : n \geq 0, \sigma_2 : n > 0, \sigma_4 : n \geq -1, \sigma_4 : n \geq 0$	\Rightarrow	$\sigma_4 : [\text{while } (n > 0) \text{ do } \alpha_1 \text{ end}] \phi_1$
14:	$\sigma_4 : n \geq -1, \sigma_4 : n \geq 0$	\Rightarrow	$\sigma_4 : [\text{while } (n > 0) \text{ do } \alpha_1 \text{ end}] \phi_1$
15:	$\sigma_2 : n \geq -1, \sigma_2 : n \geq 0$	\Rightarrow	$\sigma_2 : [\text{while } (n > 0) \text{ do } \alpha_1 \text{ end}] \phi_1$
16:	$\sigma_2 : n \geq 0$	\Rightarrow	$\sigma_2 : [\text{while } (n > 0) \text{ do } \alpha_1 \text{ end}] \phi_1$
12:	$\sigma_2 : n \geq 0, \sigma_2 : n > 0$	\Rightarrow	$\sigma_4 : [\text{while } (n > 0) \text{ do } \alpha_1 \text{ end}] \phi_1, \sigma_4 : n \geq -1, \sigma_4 : n \geq 0$
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	$\sigma_2 : [\text{while } (n > 0) \text{ do } \alpha_1 \text{ end}] \phi_1$
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. A Derivation of Property v_1

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 as follows:

$$\frac{\frac{\Gamma, \phi \Rightarrow \Delta}{\Gamma \Rightarrow \neg \phi, \Delta} (\neg R) \quad \frac{\Gamma, \psi \Rightarrow \Delta}{\Gamma \Rightarrow \neg \psi, \Delta} (\neg R)}{\frac{\Gamma \Rightarrow (\neg \phi) \wedge (\neg \psi), \Delta}{\Gamma, \phi \vee \psi \Rightarrow \Delta} (\wedge R)} (\neg L)$$

The derivation from sequent 1 to 2 (also the derivation from 14 to 15) is according to the rule (Sub) :

$$\frac{\Gamma \Rightarrow \Delta}{\Gamma[e/x] \Rightarrow \Delta[e/x]} (Sub)$$

where the function $(\cdot)[e/x]$ is an instantiation of the abstract substitution defined in Definition 4.12. 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 m th 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.

One feature of the above deduction process is that the loop structure of the while program WP (i.e. *while...do...end*) is reflected in the cyclic derivation tree itself. To reason about WP one does not need the inference rule for decomposing the loop structure. This is useful especially in program models in which loop structures are usually implicit, such as CCS-like process algebras [38, 39] and imperative synchronous languages [12, 55]. DLp provides an incremental reasoning in which we can avoid prior program transformations as done in work like [9, 55].

6.2 Lifting Rules From FODL

Two examples in DLp -WP are given to illustrate how the existing inference rules from the theory of FODL (cf. [29]) can be applied for deriving the compositional while programs through the lifting processes as defined in Section 5.5.

In FODL, consider the rule

$$\frac{\Gamma \Rightarrow [\alpha][\beta]\phi, \Delta}{\Gamma \Rightarrow [\alpha; \beta]\phi, \Delta} \quad ([;])$$

which means that to prove formula $[\alpha; \beta]\phi$, we only need to prove formula $[\alpha][\beta]\phi$ in which program α is firstly proved separated from program β . It comes from the valid formula $[\alpha][\beta]\phi \rightarrow [\alpha; \beta]\phi$, acting as a compositional rule appearing in many dynamic logic calculi that are based on FODL (e.g. [7]). By Proposition 5.10, in DLp -WP, we can lift ($[;]$) as a rule

$$\frac{\sigma : \Gamma \Rightarrow \sigma : [\alpha][\beta]\phi, \sigma : \Delta}{\sigma : \Gamma \Rightarrow \sigma : [\alpha; \beta]\phi, \sigma : \Delta} \quad (\sigma[;])$$

where σ is a free configuration in $free(L, \Gamma \cup \Delta \cup \{[\alpha][\beta]\phi, [\alpha; \beta]\phi\})$. As an additional rule, in system Pr_{dlp} , $(\sigma[;])$ provides a compositional reasoning for sequential programs. It is useful when verifying a property like $\Gamma, \sigma' : [\beta]\phi \Rightarrow \sigma : [\alpha; \beta]\phi, \Delta$, in which we might finish the proof by only symbolic executing program α as:

$$\frac{\frac{\frac{\Gamma, \sigma' : [\beta]\phi \Rightarrow \sigma' : [\beta]\phi, \Delta}{\Gamma, \sigma' : [\beta]\phi \Rightarrow \sigma' : [\downarrow][\beta]\phi, \Delta} \quad (ax)}{\vdots} \quad ([\downarrow])}{\vdots} \quad ([\alpha]R) \quad \frac{\vdots}{\Gamma, \sigma' : [\beta]\phi \Rightarrow \sigma : [\alpha][\beta]\phi, \Delta} \quad ([\alpha]R) \quad \frac{\Gamma, \sigma' : [\beta]\phi \Rightarrow \sigma : [\alpha][\beta]\phi, \Delta}{\Gamma, \sigma' : [\beta]\phi \Rightarrow \sigma : [\alpha; \beta]\phi, \Delta} \quad (\sigma[;])$$

especially when verifying the program β can be very costly.

Another example is the rule

$$\frac{\phi \Rightarrow \psi}{[\alpha]\phi \Rightarrow [\alpha]\psi} \quad ([Gen])$$

$\frac{}{\Gamma \Rightarrow (x := e, \sigma) \longrightarrow (\downarrow, \sigma_e^x), \Delta} \quad (x := e)$		$\frac{\Gamma \Rightarrow \sigma : \phi, \Delta}{\Gamma \Rightarrow (\phi?, \sigma) \longrightarrow (\downarrow, \sigma), \Delta} \quad (\phi?)$	
$\frac{\Gamma \Rightarrow (\alpha, \sigma) \longrightarrow (\alpha', \sigma'), \Delta}{\Gamma \Rightarrow (\alpha; \beta, \sigma) \longrightarrow (\alpha'; \beta, \sigma'), \Delta} \quad 1 \text{ (;)}$		$\frac{\Gamma \Rightarrow (\alpha, \sigma) \longrightarrow (\downarrow, \sigma'), \Delta}{\Gamma \Rightarrow (\alpha; \beta, \sigma) \longrightarrow (\beta, \sigma'), \Delta} \quad (;\downarrow)$	
$\frac{}{\Gamma \Rightarrow (\alpha \cup \beta, \sigma) \longrightarrow (\alpha, \sigma), \Delta} \quad (\cup 1)$		$\frac{}{\Gamma \Rightarrow (\alpha \cup \beta, \sigma) \longrightarrow (\beta, \sigma), \Delta} \quad (\cup 2)$	
$\frac{}{\Gamma \Rightarrow (\alpha^*, \sigma) \longrightarrow (\alpha; \alpha^* \cup \text{true?}, \sigma), \Delta} \quad 2 \text{ (*)}$			

Table 4. Partial Rules of $(\mathbf{Pr}_{op})_{fodl}$ for Program Transitions of Regular Programs

for generating modality $[\cdot]$, which were used for deriving the structural rule of star regular programs in FODL (cf. [29]). By Proposition 5.10, we lift $([Gen])$ as the following rule:

$$\frac{\sigma : \phi \Rightarrow \sigma : \psi}{\sigma : [\alpha]\phi \Rightarrow \sigma : [\alpha]\psi} \quad (\sigma[Gen]),$$

where $\sigma \in \text{free}(\mathbf{L}, \{[\alpha]\phi, [\alpha]\psi, \phi, \psi\})$. It is useful, for example, when deriving a property $\sigma : [\alpha]\langle\beta\rangle\phi \Rightarrow \sigma : [\alpha]\langle\beta'\rangle\psi$, where we can skip the derivation of program α as follows:

$$\frac{\sigma : \langle\beta\rangle\phi \Rightarrow \sigma : \langle\beta'\rangle\psi}{\sigma : [\alpha]\langle\beta\rangle\phi \Rightarrow \sigma : [\alpha]\langle\beta'\rangle\psi} \quad (\sigma[Gen]),$$

and directly focus on deriving the programs β and β' .

From these two examples it can be seen that in practical derivations, lifting process can be used to reduce the burden of certain verifications.

6.3 Instantiation of DLp in FODL Theory

We instantiate DLp with the theory of FODL. The resulted theory, namely DLp-FODL, provides an alternative way of reasoning about FODL formulas through symbolically executing regular programs.

The instantiation process mainly follows that for while programs as we have seen in Example 4.4, 4.9, 4.10 and 5.2, where the only differences are: (1) the parameter \mathbf{P} is instantiated as the set of regular programs (Section 3), denoted by \mathbf{P}_{fodl} ; (2) the program behaviours are captured by a set of rules for regular programs, denoted by $(\mathbf{Pr}_{op})_{fodl}$, whose rules for the part of the program transitions of regular programs are shown in Table 4. And we omit the part of the rules for program terminations of regular programs.

It is interesting to compare our proof system: $(Pr_{dlp})_{fodl} =_{df} Pr_{ldlp} \cup (\mathbf{Pr}_{op})_{fodl}$ for FODL with the traditional proof system of FODL (cf. [29]). By simple observations, we can see that for non-star regular programs, our proof system can do what the traditional proof system can. For example, an FODL formula

$$[(\alpha; \beta) \cup \alpha]\phi,$$

where let $\alpha =_{df} (x := x + 1)$ and $\beta =_{df} (y := 0)$, can be derived in the following process in the traditional proof system of FODL by using certain rules:

$$\frac{\frac{x \geq 0 \Rightarrow [\beta]x + 1 > 0}{x \geq 0 \Rightarrow [\alpha][\beta]x > 0} (x:=e) \quad \frac{x \geq 0 \Rightarrow [\alpha][\beta]x > 0}{x \geq 0 \Rightarrow [\alpha;\beta]x > 0} (;)}{\frac{x \geq 0 \Rightarrow [\alpha]x > 0}{x \geq 0 \Rightarrow [(\alpha;\beta) \cup \alpha]x > 0} (\cup)}.$$

In $(Pr_{dlp})_{fodl}$, correspondingly, we can find a logical equivalent labeled version:

$$\{x \mapsto t\} : x \geq 0 \Rightarrow \{x \mapsto t\} : [(\alpha;\beta) \cup \alpha]x > 0$$

and have the following derivations by applying the rule $([\alpha]R)$ and using the corresponding operational rules in $(Pr_{op})_{fodl}$ (which are not shown below):

$$\frac{\frac{\{x \mapsto t\} : x \geq 0 \Rightarrow \{x \mapsto t + 1\} : [\beta]x > 0}{\{x \mapsto t\} : x \geq 0 \Rightarrow \{x \mapsto t\} : [\alpha;\beta]x > 0} ([\alpha]R) \quad \{x \mapsto t\} : x \geq 0 \Rightarrow \{x \mapsto t\} : [\alpha]x > 0}{\{x \mapsto t\} : x \geq 0 \Rightarrow \{x \mapsto t\} : [(\alpha;\beta) \cup \alpha]x > 0} ([\alpha]R).$$

Especially, we are interested in whether $(Pr_{dlp})_{fodl}$ (w.r.t. a suitable $(Pr_{op})_{fodl}$ for which we only give a part of the rules here), like the traditional proof system of FODL, is complete related to the arithmetical theory of integers. One way to prove this, as we can see now, is by applying our conditional completeness result for DLp proposed in Section 7.2, in which the crucial step is showing that regular programs in the proof system $(Pr_{dlp})_{fodl}$ is well-behaved (Definition 7.14).

More of these aspects will be discussed in detail in our future work.

The heterogeneity of the verification framework of DLp can be reflected from this example. Although while programs are a subset of regular programs in the context of our discussion (see Section 3), the while programs have its own set of the inference rules for their program transitions (Table 1), which is different from those for regular programs (Table 4). Our DLp formulas provide a convenient way to compare the behaviours of different models. For example, given a regular program WP_r which is syntactically equivalent to WP (Example 4.4):

$$WP_r =_{df} ((n > 0)? ; s := s + n ; n := n - 1)^* ; \neg(n > 0)?,$$

we want to verify that whether their behaviours lead to the same result according to their own operational semantics. This property can be described as a DLp formula as follows:

$$\cdot \Rightarrow \sigma : [WP][WP_r](s = s' \wedge n = n'),$$

where $\sigma = \{s \mapsto t, n \mapsto N, s' \mapsto t, n' \mapsto N\}$; WP_r' is obtained from WP_r by replacing all appearances of the variables s, n with their fresh counterparts s', n' in order to avoid variable collisions; t, N are fresh variables other than s, n, s', n' . Intuitively, the formula says that if the inputs of WP and WP_r are the same, after running them separately (without interactions), their outputs are the same.

6.4 An Encoding of Process Logic in DLp

DLp allows even more complex labels and formulas: the labels can be more than simple program states, while the formulas can be more than simple static ones (which is either true or false at a world). Below, we give a sketch of a possible encoding of a first-ordered version of the theory of process logic [28] (PL) in DLp, namely DLp-PL. DLp-PL is able to specify and reason about progressive behaviours of programs using temporal formulas. Compared to Hoare

logic, it is more suitable for reactive systems [26], in which a program may never terminate and we care more about if a property holds on an intermediate state.

Process logic can be seen as a type of dynamic logics in which the semantics of formulas is defined in terms of paths rather than worlds. The form $[\alpha]\phi$ of a PL dynamic formula inherits from that of PDL, where α is a regular program just as in PDL. Formula ϕ is a temporal formula defined such that (1) any atomic proposition p is a temporal formula; (2) $\mathbf{f}\phi$ and $\phi \mathbf{suf} \psi$ are temporal formulas, provided that ϕ and ψ are temporal formulas; (3) $\neg\phi$, $\phi \wedge \psi$ are temporal formulas, if ϕ and ψ are temporal formulas. The semantics of a regular program and a temporal formula is thus given by paths of worlds. For a regular program, its semantics corresponds to the set of its execution paths. For a temporal formula, given a path tr , its semantics is defined as follows:

1. $tr \models p$, if $tr_b \models p$;
2. $tr \models \mathbf{f}\phi$, if $tr_b \models \phi$;
3. $tr \models \phi \mathbf{suf} \psi$, if there is a path tr' such that (i) $tr' \prec_s tr$ and $tr' \models \psi$, and (ii) for all paths tr'' with $tr' \prec_s tr'' \prec_s tr$, $tr'' \models \phi$;
4. $tr \models \neg\phi$, if $tr \not\models \phi$, and $tr \models \phi \wedge \psi$, if $tr \models \phi$ and $tr \models \psi$.

Note that the operators \mathbf{f} and \mathbf{suf} are sufficient to express the meaning of the usual temporal operators (cf. [28]), for example the “next” operator: $\mathbf{n}\phi =_{df} \text{false} \mathbf{suf} \phi$, the “future” operator: $\Diamond\phi =_{df} \phi \vee (\text{true} \mathbf{suf} \phi)$, etc. Based on the semantics of temporal formulas, the semantics of a dynamic PL formula $[\alpha]\phi$ is defined w.r.t. a path tr as:

$$tr \models [\alpha]\phi, \text{ if for all execution paths } tr' \text{ of } \alpha, tr' \cdot tr \models \phi.$$

Our instantiation process is almost the same as DLp-FODL, except that we choose \mathbf{F} to be the set of temporal formulas introduced above, denoted by \mathbf{F}_{pl} . And we choose a different set of labels named \mathbf{L}_{pl} in which each label is defined as a form that captures the meaning of a sequence of the configurations in \mathbf{L}_W :

$$\sigma_1\sigma_2\ldots\sigma_n \ (n \geq 1), \sigma_i \in \mathbf{L}_W \ (1 \leq i \leq n).$$

Besides, similar to the choosing of $(\mathbf{Pr}_{op})_{fodl}$, we propose a set $(\mathbf{Pr}_{op})_{pl}$ of rules for the regular programs. In $(\mathbf{Pr}_{op})_{pl}$, the forms of the rules for the program transitions are the same as those in $(\mathbf{Pr}_{op})_{fodl}$ (see Table 4), except that (1) we replace each configuration with the configuration in \mathbf{L}_{pl} , and (2) we have the following rule for assignments:

$$\frac{\Gamma \Rightarrow (x := e, l) \longrightarrow (\downarrow, l(\sigma_n)_e^x), \Delta}{\Gamma \Rightarrow (x := e, l) \longrightarrow (\downarrow, l(\sigma_n)_e^x), \Delta} \quad (x:=e), \text{ where } l = \sigma_1\sigma_2\ldots\sigma_n \ (n \geq 1)$$

where instead we append the current result $(\sigma_n)_e^x$ to the tail of the current sequence l of the configurations in \mathbf{L}_W .

Consider an example of programs’ temporal properties:

$$\sigma : [\alpha; \alpha] \Diamond x > 0,$$

with $\sigma = \{x \mapsto -1\}$, $\alpha = (x := x + 1)$. It says that along the execution path of $\alpha; \alpha$, $x > 0$ eventually holds. By the rules in $(\mathbf{Pr}_{op})_{pl}$, we have the following derivation:

$$\frac{\frac{\frac{\cdot \Rightarrow \sigma\sigma'\sigma'' : \Diamond x > 0}{\cdot \Rightarrow \sigma\sigma'\sigma'' : [\downarrow] \Diamond x > 0} \quad ([\downarrow])}{\cdot \Rightarrow \sigma\sigma' : [\alpha] \Diamond x > 0} \quad ([\alpha]R),}{\cdot \Rightarrow \sigma : [\alpha; \alpha] \Diamond x > 0} \quad ([\alpha]R)$$

where $\sigma' = \{x \mapsto 0\}$, $\sigma'' = \{x \mapsto 1\}$.

$\frac{l_b : \phi}{l : \mathbf{f} \phi} \text{ (f)} \quad \frac{\Gamma \Rightarrow l : \phi, \Delta \quad \Gamma \Rightarrow l : \phi \mathbf{suf} \psi, \Delta}{\Gamma \Rightarrow \sigma l : \phi \mathbf{suf} \psi, \Delta} \text{ (suf R1)} \quad \frac{\Gamma \Rightarrow l : \psi, \Delta}{\Gamma \Rightarrow \sigma l : \phi \mathbf{suf} \psi, \Delta} \text{ (suf R2)}$	
$\frac{\Gamma, l : \phi, l : \phi \mathbf{suf} \psi \Rightarrow \Delta \quad \Gamma, l : \psi \Rightarrow \Delta}{\Gamma, \sigma l : \phi \mathbf{suf} \psi \Rightarrow \Delta} \text{ (suf L)}$	
<hr/> <p>in the above rules, $l \in \mathbf{L}_W \mathbf{L}_W^*$, $\sigma \in \mathbf{L}_W$.</p> <hr/>	

Table 5. Rules for Labeled Temporal Formulas in DLp-PL

Unlike the formulas of \mathbf{F}_{af0} in DLp-WP and DLp-FODL, in DLp-PL we can further derive labeled temporal formulas (e.g. $\sigma\sigma'\sigma'' : \Diamond x > 0$) using the following additional rules shown in Table 5. These rules are directly according to the semantics of the operators \mathbf{f} and \mathbf{suf} .

A cyclic derivation for iterative programs (like α^*) in DLp-PL, however, requires higher-ordered label structures together with a suitable instantiation of the abstract substitutions as defined in Definition 4.12. Our future work will discuss more about it, as well as the analysis of the (relative) completeness of DLp-PL and its comparison to the traditional theory of PL.

7 Analysis of Soundness and Completeness of DLp

In this section, we analyze the soundness and completeness of the proof system Pr_{dlp} . Currently, we consider the soundness under a restriction on the program behaviours of \mathbf{P} (Definition 7.2). However, as analyzed below in detail, the set of programs under the restriction is still a rich one. For the completeness, since DLp is not a specific logic, generally, it is impossible to discuss about its completeness without any restrictions on the parameters of DLp. Instead, we study under which conditions (Definition 7.13 and 7.14) can we obtain a completeness result relative to the labeled non-dynamic formulas.

Section 7.1 discusses about the soundness of Pr_{dlp} , while Section 7.2 discusses about its completeness.

7.1 Conditional Soundness of DLp

We first introduce the concept of *minimum execution paths*.

Definition 7.1. An execution path (Definition 4.5) $w_1 \dots w_n$ ($n \geq 1$) 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.

The restriction condition is stated in the following definition.

Definition 7.2 (Termination Finiteness). Starting from a world $w \in \mathcal{S}$ and a program $\alpha \in \mathbf{P}$, there is only a finite number of minimum execution paths (of the form: $w \xrightarrow{\alpha/\cdot} \dots$).

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 obviously fall into this category. For example, automata that have non-deterministic transitions but have a finite number of states. More on this restriction will be discussed in our future work.

THEOREM 7.3 (CONDITIONAL SOUNDNESS OF DLp). *If the programs in \mathbf{P} satisfy the termination finiteness property, then for any labeled formula $\sigma : \phi \in \mathfrak{F}_{dlp}, Pr_{dlp} \vdash (\cdot \Rightarrow \sigma : \phi)$ implies $\models \sigma : \phi$.*

Main Idea for Proving the Soundness. We follow the main idea behind [15] to prove Theorem 7.3 by contradiction. The key point is that, if the conclusion of a cyclic proof is invalid, then by the soundness of all the rules in Pr_{dlp} (Theorem 5.3), there must exist an *invalid derivation path* in which each node is invalid, and one of its progressive traces leads to an infinite descent sequence of some well-founded set (introduced below), which violates the definition of the well-foundedness (cf. [18]) itself.

Below we firstly introduce the well-founded relation \preceq_m we rely on, then we focus on the main skeleton of proving Theorem 7.3. Other proof details are given in Appendix A.

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: $a \succ a_1 \succ a_2 \succ \dots$ in S . Set S is called a *well-founded set* w.r.t. \preceq .

Definition 7.4 (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 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, \dots, tr_n ($n \geq 1$) in C_1 are proper suffixes of tr .

In Definition 7.4, note that we can replace an element of C_2 with an *empty execution path* whose length is 0. And if we do so, it is equivalent to that we remove an element from C_2 .

PROPOSITION 7.5. \preceq_m is a partial-order relation.

The proof of Proposition 7.5 is given in Appendix A.

Example 7.6. 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 7.7. Relation \preceq_m is a well-founded relation.

We omit the proof of Proposition 7.7. Relation \preceq_m is just a special case of the “multi-set ordering” introduced in [18], where it has been proved to be well-founded. Intuitively, we observe that for two sets C_1 and C_2 such that $C_1 \prec_m C_2$, for each set D_{tr} of the paths in C_1 that replaces an element tr in C_2 , the maximum length of the elements of D_{tr} is strictly smaller than that of tr . By that C_2 is finite, we can see that such a replacement decreases the number of the paths that have the maximum length of the elements in C_2 .

Proof Skeleton of Theorem 7.3. Below we give the main skeleton of the proof by skipping the details of the proof of Lemma 7.9, which can be found in Appendix A.

Following the main idea above, we first introduce the concept of the “execution paths of a dynamic DLp formula”. They are the elements of a well-founded relation \preceq_m . Next, we propose Lemma 7.9, which plays a key role in the proof of Theorem 7.3 that follows.

Definition 7.8 (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 $mex(w, \alpha) =_{df} \{w \dots w' \mid w \xrightarrow{\alpha/\cdot} \dots \xrightarrow{\cdot/\downarrow} w' \text{ is a min. exec. path for some } w' \in \mathcal{S}\}$ is the set of all minimum paths of α starting from world w .

In Definition 7.8, an execution path of a dynamic formula may be concatenated by several execution paths that belong to different programs in a sequence of modalities. As seen in the proof of Lemma 7.9 (Appendix A), this consideration is necessary because a dynamic formula may contain more than one modality (e.g. $[\alpha][\beta]\phi$). It is also one of the main differences between our proof and the proof given in [61].

In the following, we call $m \in \mathbf{M}$ a *counter-example mapping* of a node v , if it makes v invalid.

LEMMA 7.9. *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 (v, v') of an invalid derivation path, where $\phi, \phi' \in \mathfrak{F}_{dlp}$. For any set $EX(m(\sigma), \phi)$ of $\sigma : \phi$ w.r.t. a counter-example mapping m of v , there exists a counter-example mapping m' of v' and a set $EX(m'(\sigma'), \phi')$ of $\sigma' : \phi'$ such that $EX(m'(\sigma'), \phi') \preceq_m EX(m(\sigma), \phi)$. Moreover, if $(\sigma : \phi, \sigma' : \phi')$ is a progressive step, then $EX(m'(\sigma'), \phi') \prec_m EX(m(\sigma), \phi)$.*

Intuitively, Lemma 7.9 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 7.7 and Lemma 7.9, we give the proof of Theorem 7.3 as follows.

PROOF OF THEOREM 7.3. Let $v = (\cdot \Rightarrow \sigma : \phi)$. By contradiction, suppose $\not\models \sigma : \phi$, that is, v is invalid. Then by the soundness of each rule in Pr_{dlp} (Theorem 5.3), there exists an invalid derivation path p from v (where every sequent is invalid). Since $Pr_{dlp} \vdash v$ (i.e., a cyclic proof tree is formed to prove the conclusion v), let $\tau_1 \tau_2 \dots \tau_k \dots$ be a progressive trace over p of the form: $v \dots v_1 v_2 \dots v_k \dots$ ($k \geq 1$), where each formula τ_i is in v_i ($i \geq 1$). Let $\tau_i =_{df} \sigma_i : \phi_i$.

Since v_1 is invalid, let m_1 be one of its counter-example mappings. By Lemma 7.9, from $EX(m_1(\sigma_1), \phi_1)$, there exists an infinite sequence of sets EX_1, \dots, EX_k, \dots ($k \geq 1$), where each $EX_i =_{df} EX(m_i(\sigma_i), \phi_i)$ ($i \geq 1$) with m_i a counter-example mapping of node v_i , and which satisfies that $EX_1 \succeq_m \dots \succeq_m EX_k \succeq_m \dots$. Moreover, since trace $\tau_1 \tau_2 \dots \tau_k \dots$ is progressive (Definition 5.5), 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 7.7).

□

7.2 Conditional Completeness of DLp

We propose two sufficient conditions for the relative completeness of DLp: 1) that the program models of DLp always have finite expressions (Definition 7.13); and 2) that their *loop programs* are always *well-behaved* (Definition 7.14) during the reasoning process. A loop program is a program that eventually reaches itself during a sequence of symbolic-execution reasoning under a label.

Below we first introduce these conditions, under them we then prove the relative completeness of DLp. We only give an outline and put the technical details of the proof in Appendix A.

Definition 7.10 (Program Sequences). Given a context Γ , a program $\alpha \in \mathbf{P}$ and a label $\sigma \in \mathbf{L}$, a (potentially infinite) sequence: $\Gamma : (\alpha_1, \sigma_1, \Gamma_1)(\alpha_2, \sigma_2, \Gamma_2) \dots (\alpha_n, \sigma_n, \Gamma_n) \dots$ ($1 \leq n < \infty, \alpha_1 = \alpha, \sigma_1 = \sigma$), called an “ α sequence”, is defined if there

is a sequence of derivations in system Pr_{dlp} as follows: $Pr_{dlp} \vdash (\Gamma_1 \Rightarrow (\alpha_1, \sigma_1) \longrightarrow (\sigma_2, \sigma_2)), Pr_{dlp} \vdash (\Gamma_2 \Rightarrow (\alpha_2, \sigma_2) \longrightarrow (\sigma_3, \sigma_3)), \dots, Pr_{dlp} \vdash (\Gamma_{n-1} \Rightarrow (\alpha_{n-1}, \sigma_{n-1}) \longrightarrow (\alpha_n, \sigma_n)), \dots$, which satisfies that $\models (\Gamma_1 \Rightarrow \Gamma)$ and $\models (\Gamma_n \Rightarrow \Gamma_{n-1})$ for all $n \geq 2$.

We call a sequence $\Gamma : (\alpha_1, \sigma_1)(\alpha_2, \sigma_2) \dots (\alpha_n, \sigma_n) \dots$ a “core α sequence” if there exist $\Gamma_1, \Gamma_2, \dots, \Gamma_n, \dots$ such that $\Gamma : (\alpha_1, \sigma_1, \Gamma_1)(\alpha_2, \sigma_2, \Gamma_2) \dots (\alpha_n, \sigma_n, \Gamma_n) \dots$ is an α sequence.

Intuitively, starting from (α, σ) under context Γ , an α sequence is a sequence of derivations where in each step, the context can only be strengthened from Γ . By the soundness of Pr_{dlp} w.r.t. \mathfrak{F}_{pt} (see Definition 5.1), each derivation of an α sequence is actually a symbolic execution step of the program.

Definition 7.11 (Program Loop Sequences). An “ α -loop sequence” of a program $\alpha \in \mathbf{P}$ is an α sequence: $\Gamma : (\alpha_1, \sigma_1, \Gamma_1) \dots (\alpha_n, \sigma_n, \Gamma_n)$ ($n \geq 1$) such that $\alpha_1 = \alpha_n = \alpha$, and $\alpha_i \neq \alpha_j$ for any other α_i and α_j , with $1 \leq i < j \leq n$.

Example 7.12. In the instantiation theory DLp-FODL as defined in Section 6.3, let $\alpha =_{df} (x := x + 1)$, $\beta =_{df} (y := 0)$, $\sigma = \{x \mapsto t\}$, then for the program $\alpha^* ; \beta$, we have an α sequence: $\emptyset : (\alpha^* ; \beta, \sigma, \emptyset)((\alpha ; \alpha^* \cup true?) ; \beta, \sigma, \emptyset)(\alpha ; \alpha^* ; \beta, \sigma, \emptyset)(\alpha^* ; \beta, \sigma', \emptyset)$, where $\sigma' = \{x \mapsto t + 1\}$. It corresponds to the following derivations:

$$\begin{aligned} (Pr_{dlp})_{fodl} \vdash (\cdot \Rightarrow (\alpha^* ; \beta, \sigma) \longrightarrow ((\alpha ; \alpha^* \cup true?) ; \sigma), \\ (Pr_{dlp})_{fodl} \vdash (\cdot \Rightarrow ((\alpha ; \alpha^* \cup true?) ; \beta, \sigma) \longrightarrow (\alpha ; \alpha^* ; \beta, \sigma)), \\ (Pr_{dlp})_{fodl} \vdash (\cdot \Rightarrow (\alpha ; \alpha^* ; \beta, \sigma) \longrightarrow (\alpha^* ; \beta, \sigma')) \end{aligned}$$

according to the corresponding rules in Table 4. This α sequence is also a loop one.

Definition 7.13 (Expression Finiteness Property). For a program $\alpha \in \mathbf{P}$, there is a natural number N_α such that in each α sequence under a label $\sigma \in \mathbf{L}$ and a context $\Gamma : \Gamma : (\alpha_1, \sigma_1, \Gamma_1) \dots (\alpha_n, \sigma_n, \Gamma_n) \dots$ ($1 \leq n < \infty$, $\alpha_1 = \alpha$, $\sigma_1 = \sigma$), the number of the different programs among $\alpha_1, \dots, \alpha_n, \dots$ is no greater than N_α .

Definition 7.13 means that as a program proceeds, it eventually reaches to the form of itself in a limit number of steps. Most program models in practice satisfy this property. But there are exceptions, for example, the programs in π -calculus with the replication operator (cf. [39]).

Definition 7.14 (Well-behaved Loop Programs). A program $\alpha \in \mathbf{P}$ is called a “loop program”, if there exists an “ α -loop sequence” for some label and context.

A loop program α is “well-behaved”, if for any label $\sigma \in \mathbf{L}$ and context Γ , there exist a label $\sigma' \in \mathbf{L}$, a context Γ' and a substitution $\eta : \mathbf{L} \rightarrow \mathbf{L}$ satisfying the following conditions:

1. $\sigma = \eta(\sigma')$ and $\Gamma = \eta(\Gamma')$;
2. $\models (\Gamma \Rightarrow \sigma \Downarrow \alpha)$ implies $\models (\Gamma' \Rightarrow \sigma' \Downarrow \alpha)$;
3. For each α -loop sequence: $\Gamma' : (\alpha, \sigma', \Gamma_1) \dots (\alpha, \sigma'', \Gamma_n)$ ($n \geq 1$), there exist a context Γ'' and a substitution $\xi : \mathbf{L} \rightarrow \mathbf{L}$ such that $\Gamma'' = \xi(\Gamma')$, $\sigma'' = \xi(\sigma')$ and $\models (\Gamma_n \Rightarrow \Gamma'')$.

The well-behaved property describes the “ability” of a loop program α to form back-links along its symbolic executions. To be more specific, for a label $\sigma \in \mathbf{L}$, it is possible to find a suitable label σ' from which σ can be obtained through substitutions, and for every symbolic execution starting from (α, σ') , we can go back to (α, σ') through substitutions. The σ' here plays the same role of the loop invariants in the normal deduction approaches of program logics.

THEOREM 7.15 (CONDITIONAL COMPLETENESS OF DLp). *If the programs in \mathbf{P} satisfy the expression finiteness property and among them all loop programs are well-behaved, then for any labeled formula $\sigma : \phi \in \mathcal{F}_{dlp}, \models \sigma : \phi$ implies $Pr_{dlp} \vdash (\cdot \Rightarrow \sigma : \phi)$.*

Note that the relativeness of the completeness result to labeled non-dynamic formulas is reflected by the rule (Ter) (Table 2).

Main Idea for Proving the Completeness. To prove Theorem 7.15, we firstly reduce it to the special case of deriving a sequent of the form $\Gamma \Rightarrow [\langle \alpha \rangle] \phi$ as shown in Lemma 7.16. For this step we take a similar approach from [27]. The main technical part is deriving the sequent $\Gamma \Rightarrow \sigma : [\langle \alpha \rangle] \phi$. We proceed by a simultaneous induction on the number of the modalities and maximum number of the forms of the programs that can appear during the derivation of the sequent. The critical observation is that by the finiteness when executing α (Lemma 7.17) and the well-behaved property (Definition 7.14) it satisfies, each non-terminal derivation branch from $\Gamma \Rightarrow \sigma : [\langle \alpha \rangle] \phi$ is able to form a back-link, which in-turn shows that the whole derivation of $\Gamma \Rightarrow \sigma : [\langle \alpha \rangle] \phi$ can form a cyclic proof.

We put the proof of Lemma 7.16 in Appendix A in details.

LEMMA 7.16. *Under the same conditions as in Theorem 7.15, for any valid sequent of the form: $\Gamma \Rightarrow \sigma : [\langle \alpha \rangle] \phi$, $Pr_{dlp} \vdash (\Gamma \Rightarrow \sigma : [\langle \alpha \rangle] \phi)$.*

In Lemma 7.16, “ $[\langle \alpha \rangle]$ ” just means either $[\alpha]$ or $\langle \alpha \rangle$.

LEMMA 7.17. *Starting from a program $\alpha \in \mathbf{P}$ and a label $\sigma \in \mathbf{L}$ under a context Γ , there is only a finite number of core α -loop sequences.*

PROOF. By the finiteness of set \mathbf{Pr}_{op} , fixing a program β , there is a maximum number of transitions starting from (β, l) in the form of: $\Gamma \Rightarrow (\beta, l) \longrightarrow \dots$ for all labels $l \in \mathbf{L}$ and contexts Γ . By the expression finiteness property and the characteristic of program loop sequences, it is not hard to see the result. \square

To close this section, we give the proof of Theorem 7.15.

PROOF OF THEOREM 7.15. For a labelled formula $\sigma : \phi$, ϕ is semantically equivalent to a conjunctive normal form: $C_1 \wedge \dots \wedge C_n$ ($n \geq 1$). Each clause C_i ($1 \leq i \leq n$) is a disjunction of literals: $C_i = l_{i,1} \vee \dots \vee l_{i,m_i}$, where $l_{i,j}$ ($1 \leq i \leq n, 1 \leq j \leq m_i$) is an atomic DLp formula or its negation. By the rules for labeled proposition logical formulas in Table 2, to prove formula $\sigma : \phi$, it is enough to show that for each clause $C_i, \models \sigma : C_i$ implies $Pr_{dlp} \vdash \sigma : C_i$. Without loss of generality, let $C_i = \psi \vee [\langle \alpha \rangle] \phi$. Then it is sufficient to prove $Pr_{dlp} \vdash (\sigma : \neg \psi \Rightarrow \sigma : [\langle \alpha \rangle] \phi)$. But it is just a special case of Lemma 7.16. \square

8 Related Work

Matching Logic and Its Variations. The idea of reasoning about programs based on their operational semantics is not new. Previous work such as [17, 52, 53, 56] in the last decade has addressed this issue using theories based on rewriting logic [37]. Matching logic [52] 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 [53, 56] were developed by enhancing the expressive power of the reachability rule. A more powerful matching μ -logic [17] 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 $[\alpha]\phi$ of DLp requires additional work and program transformations. On the other hand, dynamic logics like DLp provide a more direct way 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 $[\cdot]$ in dynamic logic when the programs are non-deterministic (which means that there are more than one execution path starting from a world and a program). We conjecture that matching μ -logic can encode DLp, as it has been claimed that it can encode traditional dynamic logics (cf. [17]).

General Frameworks based on Set Theories. [40] 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 [40], 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 [40], [36] also proposed a general framework for program reasoning, but via big-step operational semantics. Unlike the frameworks in [40] and [36] which are directly built up on mathematical set theory, DLp 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 DLp cannot be expressed in the framework of [40].

Updates. The structure “updates” adopted in work [6, 7, 44] are “delay substitutions” of variables and terms. They in fact can be defined as a special case of the more general structure labels in DLp by choosing suitable label mappings accordingly.

Logics based on Cyclic Proof Approach. The proof system of DLp relies on the cyclic proof theory which firstly arose in [57] and was later developed in different logics such as [15, 16], and more recent work like [3, 33, 59]. [34] proposed a complete cyclic proof system for μ -calculus, which subsumes PDL [23] in its expressiveness. In [20], the authors proposed a complete labeled cyclic proof system for PDL. Both logics in [20, 34] 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 labeled form of DLp formula $\sigma : [\alpha]\phi$ is inspired from [20], where a label is just a variable of worlds in a traditional Kripke structure. On the other hand, the labels in DLp allow arbitrary terms from actual program configurations.

Generalizations of Dynamic Logic. There are some recent work for generalizing the theories of dynamic logics [2, 30, 60]. [30] proposes a general dynamic logic by allowing the program models of PDL to be any forms than regular programs. The semantics of a program is given by a set of so-called “interaction-based” behaviours, very similar to the program transitions here. However, there, it only focuses on the building of the logic theory. No associated proof systems were proposed. In [2], an operational version of PDL (namely OPDL) was studied. There, the proof of a dynamic formula $[\alpha]\phi$ can be reduced to the proof of formula $[a][\beta]\phi$ if $\alpha \xrightarrow{a} \beta$ is a transition by doing an action a . Similar to [20], a complete non-well-founded proof system was built for OPDL. Although in [2] it was claimed that OPDL can be adapted to arbitrary program models, its theory was analyzed only on the propositional level and only for regular programs. [60] develops heterogeneous dynamic logic (HDL), a theoretical framework in which different dynamic-logic theories can be compared and jointly used. Unlike [60] which makes a systematic analysis of the integration of different theories, the start point of our work (as well as [61]) is to facilitate the operationally-based reasoning of different programs. This leads to the introduction of labels and the development of the cyclic reasoning in DLp as critical contributions, while the lifting process acts as a “side technique” to compensate for the core proof system. The result of [60] offers a thorough guide for the analysis of the completeness and other properties of the lifted theories in DLp in our future work.

9 Conclusion & Future Work

In this paper, we propose a novel verification framework based on dynamic logic for reasoning about programs based on their operational semantics. We mainly build the theory of DLp and analyze its soundness and completeness under certain conditions. Through the examples and case studies, we have shown the potential usage of this formalism in different aspects of program reasoning.

For future work we focus on two aspects. On the theoretical aspect, we are interested in whether we can further relax our conditions for proving the soundness and completeness of DLp. This is important to know how our framework can be also adapted to more complex models, such as hybrid or probabilistic systems. We will further study the instantiated theory DLp-PL, as a promising first-ordered version of process logic ever built, and also DLp-SP. On the practical aspect, we are carrying out a full mechanization of DLp in Rocq [13]. Currently, we have managed to deeply embed the whole theory of DLp (cf. [1]). To explore the potential applications of DLp in practice, we are working on applying DLp for specifying and reasoning about different program models, like Esterel, Rust, etc.

Acknowledgment. This work is partially supported by the New Faculty Start-up Foundation of NUAU (No. 90YAT24003) and the General Program of the National Natural Science Foundation of China (No. 62272397).

References

- [1] [n. d.]. <https://github.com/yrz5a/Coq-DLp.git>.
- [2] Matteo Acclavio, Fabrizio Montesi, and Marco Peressotti. 2024. On propositional dynamic logic and concurrency. *arXiv [cs.LO]* (March 2024).
- [3] Bahareh Afshari, Sebastian Enqvist, and Graham E Leigh. 2022. Cyclic proofs for the first-order μ -calculus. *Logic Journal of the IGPL* 32, 1 (08 2022), 1–34. arXiv:<https://academic.oup.com/jigpal/article-pdf/32/1/1/56586573/jzac053.pdf>
- [4] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, and Mattias Ulbrich. 2025. The many uses of dynamic logic. In *Lecture Notes in Computer Science*. Springer Nature Switzerland, Cham, 56–82.
- [5] Andrew W. Appel, Robert Dockins, and et al. 2014. *Program Logics for Certified Compilers*. Cambridge University Press.
- [6] Bernhard Beckert and Daniel Bruns. 2013. Dynamic Logic with Trace Semantics. In *CADE 2013*. Springer Berlin Heidelberg, 315–329.
- [7] Bernhard Beckert, Vladimir Klebanov, and Benjamin Weiß. 2016. *Dynamic Logic for Java*. Springer International Publishing, 49–106.
- [8] Mario R.F. Benevides and L. Menasché Schechter. 2010. A Propositional Dynamic Logic for Concurrent Programs Based on the π -Calculus. In *M4M 2009*. Elsevier, 49–64.
- [9] Mario R. F. Benevides and L. Menasché Schechter. 2008. A Propositional Dynamic Logic for CCS Programs. In *Logic, Language, Information and Computation*. Springer Berlin Heidelberg, Berlin, Heidelberg, 83–97.
- [10] Gérard Berry. 1989. *Programming a digital watch in Esterel v3*. Technical Report RR-1032.
- [11] Gérard Berry and Laurent Cosserat. 1985. The ESTEREL synchronous programming language and its mathematical semantics. In *Seminar on Concurrency*. Springer Berlin Heidelberg, Berlin, Heidelberg, 389–448.
- [12] Gérard Berry and Georges Gonthier. 1992. The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming* 19, 2 (1992), 87 – 152.
- [13] Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Springer, 1–472 pages.
- [14] Denis Bogdanas and Grigore Roşu. 2015. K-Java: A Complete Semantics of Java. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) (POPL '15). Association for Computing Machinery, New York, NY, USA, 445–456. <https://doi.org/10.1145/2676726.2676982>
- [15] James Brotherston, Richard Bornat, and Cristiano Calcagno. 2008. Cyclic proofs of program termination in separation logic. *SIGPLAN Not.* 43, 1 (2008), 101–112.
- [16] James Brotherston and Alex Simpson. 2007. Complete Sequent Calculi for Induction and Infinite Descent. In *LICS 2007*. 51–62.
- [17] X. Chen and G. Rosu. 2019. Matching μ -Logic. In *LICS 2019*. IEEE Computer Society, 1–13.
- [18] Nachum Dershowitz and Zohar Manna. 1979. Proving termination with multiset orderings. In *Automata, Languages and Programming*. Springer Berlin Heidelberg, Berlin, Heidelberg, 188–202.
- [19] Canh Minh Do, Tsubasa Takagi, and Kazuhiro Ogata. 2024. Automated Quantum Protocol Verification Based on Concurrent Dynamic Quantum Logic. *ACM Trans. Softw. Eng. Methodol.* (Dec. 2024). Just Accepted.
- [20] Simon Docherty and Reuben N. S. Rowe. 2019. A Non-wellfounded, Labelled Proof System for Propositional Dynamic Logic. In *TABLEAUX 2019*. Springer International Publishing, 335–352.

- [21] Chucky Ellison and Grigore Rosu. 2012. An executable formal semantics of C with applications. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Philadelphia, PA, USA) (POPL '12)*. Association for Computing Machinery, New York, NY, USA, 533–544. <https://doi.org/10.1145/2103656.2103719>
- [22] Yishai A. Feldman and David Harel. 1984. A probabilistic dynamic logic. *J. Comput. System Sci.* 28, 2 (1984), 193–215.
- [23] Michael J. Fischer and Richard E. Ladner. 1979. Propositional dynamic logic of regular programs. *J. Comput. System Sci.* 18, 2 (1979), 194–211.
- [24] Manuel Gesell and Klaus Schneider. 2012. A Hoare Calculus for the Verification of Synchronous Languages. In *PLPV 2012 (Philadelphia, Pennsylvania, USA)*. Association for Computing Machinery, 37–48.
- [25] Jens Oliver Gutsfeld, Markus Müller-Olm, and Christoph Ohrem. 2020. Propositional Dynamic Logic for Hyperproperties. In *31st International Conference on Concurrency Theory (CONCUR)*. 50:1–50:22.
- [26] Nicolas Halbwachs, Fabienne Lagnier, and Pascal Raymond. 1993. Synchronous Observers and the Verification of Reactive Systems. In *Proceedings of the Third International Conference on Methodology and Software Technology: Algebraic Methodology and Software Technology (AMAST '93)*. Springer-Verlag, Berlin, Heidelberg, 83–96.
- [27] David Harel. 1979. *First-Order Dynamic Logic*. Lecture Notes in Computer Science (LNCS), Vol. 68. Springer.
- [28] David Harel, Dexter Kozen, and Rohit Parikh. 1982. Process Logic: Expressiveness, Decidability, Completeness. *J. Comput. System Sci.* 25, 2 (1982), 144–170.
- [29] David Harel, Dexter Kozen, and Jerzy Tiuryn. 2000. *Dynamic Logic*. MIT Press.
- [30] Rolf Hennicker and Martin Wirsing. 2019. *A Generic Dynamic Logic with Applications to Interaction-Based Systems*. Springer International Publishing, 172–187.
- [31] C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580.
- [32] Benjamin Icard. 2024. A Dynamic Logic for Information Evaluation in Intelligence. arXiv:2405.19968 [cs.LO] <https://arxiv.org/abs/2405.19968>
- [33] Eddie Jones, C.-H. Luke Ong, and Steven Ramsay. 2022. CycleQ: an efficient basis for cyclic equational reasoning. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (San Diego, CA, USA) (PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 395–409.
- [34] Natthapong Jungteerapanich. 2009. A Tableau System for the Modal μ -Calculus. In *TABLEAUX 2009*. Springer Berlin Heidelberg, 220–234.
- [35] Dexter Kozen. 1985. A probabilistic PDL. *J. Comput. System Sci.* 30, 2 (1985), 162–178.
- [36] Ximeng Li, Qianying Zhang, Guohui Wang, Zhiping Shi, and Yong Guan. 2021. Reasoning About Iteration and Recursion Uniformly Based on Big-Step Semantics. In *SETTA 2021*. Springer International Publishing, 61–80.
- [37] José Meseguer. 2012. Twenty years of rewriting logic. *The Journal of Logic and Algebraic Programming* 81, 7 (2012), 721–781.
- [38] R. Milner. 1982. *A Calculus of Communicating Systems*. Springer-Verlag, Berlin, Heidelberg.
- [39] Robin Milner, Joachim Parrow, and David Walker. 1992. A Calculus of Mobile Processes, I. *Information and Computation* 100, 1 (1992), 1–40.
- [40] Brandon Moore, Lucas Peña, and Grigore Rosu. 2018. Program Verification by Coinduction. In *ESOP 2018*. Springer International Publishing, 589–618.
- [41] Oleg Mürk, Daniel Larsson, and Reiner Hähnle. 2007. KeY-C: A tool for verification of C programs. In *Automated Deduction – CADE-21*. Springer Berlin Heidelberg, Berlin, Heidelberg, 385–390.
- [42] Peter W. O’Hearn. 2019. Incorrectness logic. *Proc. ACM Program. Lang.* 4, POPL, Article 10 (2019), 32 pages.
- [43] Raúl Pardo, Einar Broch Johnsen, and et al. 2022. A Specification Logic for Programs in the Probabilistic Guarded Command Language. In *ICTAC 2022 (Tbilisi, Georgia)*. Springer-Verlag, 369–387.
- [44] André Platzer. 2007. Differential Dynamic Logic for Verifying Parametric Hybrid Systems.. In *TABLEAUX 2007 (2007-09-20)*. Springer Berlin Heidelberg, 216–232.
- [45] André Platzer. 2018. *Logical Foundations of Cyber-Physical Systems*. Springer.
- [46] André Platzer and Jan-David Quesel. 2008. KeYmaera: A Hybrid Theorem Prover for Hybrid Systems (System Description). In *Automated Reasoning*. Springer Berlin Heidelberg, Berlin, Heidelberg, 171–178.
- [47] G. D. Plotkin. 1981. *A Structural Approach to Operational Semantics*. Technical Report DAIMI FN-19. University of Aarhus.
- [48] Dumitru Potop-Butucaru, Stephen A. Edwards, and Gérard Berry. 2010. *Compiling Esterel*. Springer New York, NY.
- [49] Vaughan R. Pratt. 1976. Semantical Considerations on Floyd-Hoare Logic. In *Annual IEEE Symposium on Foundations of Computer Science (FOCS)*. IEEE Computer Society, 109–121.
- [50] Wolfgang Reif. 1995. *The Kiv-approach to software verification*. Springer Berlin Heidelberg, Berlin, Heidelberg, 339–368.
- [51] J.C. Reynolds. 2002. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. 55–74.
- [52] Grigore Roşu and Andrei Ştefănescu. 2012. Towards a Unified Theory of Operational and Axiomatic Semantics. In *ICALP 2012*. Springer Berlin Heidelberg, 351–363.
- [53] Grigore Rosu, Andrei Stefanescu, Stefan Ciobăcă, and Brandon M. Moore. 2013. One-Path Reachability Logic. In *LICS 2013*. 358–367.
- [54] K. Rustan and M. Leino. 2007. *Verification of Object-Oriented Software. The KeY Approach*. Lecture Notes in Computer Science (LNCS), Vol. 4334. Springer.
- [55] Klaus Schneider and Jens Brandt. 2017. *Quartz: A Synchronous Language for Model-Based Design of Reactive Embedded Systems*. Springer Netherlands, Dordrecht, 1–30.

- [56] Andrei Ștefănescu, Ștefan Ciobăcă, and et al. 2014. All-Path Reachability Logic. In *RTA-TLCA 2014*. Springer International Publishing, 425–440.
- [57] Colin Stirling and David Walker. 1991. Local model checking in the modal mu-calculus. *Theor. Comput. Sci.* 89, 1 (1991), 161–177.
- [58] Tsubasa Takagi, Canh Minh Do, and Kazuhiro Ogata. 2024. Automated Quantum Program Verification in Dynamic Quantum Logic. In *Dynamic Logic. New Trends and Applications*. Springer Nature Switzerland, Cham, 68–84.
- [59] Gadi Tellez and James Brotherston. 2020. Automatically Verifying Temporal Properties of Pointer Programs with Cyclic Proof. *Journal of Automated Reasoning* 64, 3 (2020), 555–578.
- [60] Samuel Teuber, Mattias Ulbrich, André Platzer, and Bernhard Beckert. 2025. Heterogeneous Dynamic Logic: Provability modulo program theories. *arXiv [cs.LO]* (July 2025).
- [61] Yuanrui Zhang. 2025. Parameterized Dynamic Logic – Towards A Cyclic Logical Framework for General Program Specification and Verification. *arXiv:2404.18098* [cs.LO] <https://arxiv.org/abs/2404.18098>
- [62] Yuanrui Zhang, Frédéric Mallet, and Zhiming Liu. 2022. A dynamic logic for verification of synchronous models based on theorem proving. *Front. Comput. Sci.* 16, 4 (2022), 3 pages.
- [63] Yuanrui Zhang, Hengyang Wu, Yixiang Chen, and Frédéric Mallet. 2021. A clock-based dynamic logic for the verification of CCSL specifications in synchronous systems. *Science of Computer Programming* 203 (2021), 102591.
- [64] Noam Zilberstein, Derek Dreyer, and Alexandra Silva. 2023. Outcome Logic: A Unifying Foundation for Correctness and Incorrectness Reasoning. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 93 (apr 2023), 29 pages.

A Other Propositions and Proofs

LEMMA A.1. *Given labeled formulas $\tau_1, \dots, \tau_n, \tau$, for any label mapping $\mathfrak{m} \in \mathbf{M}$ with $\mathfrak{m} \models \Gamma$, if $\mathfrak{m} \models \tau_1$ and ... and $\mathfrak{m} \models \tau_n$ implies $\mathfrak{m} \models \tau$, then rule*

$$\frac{\Gamma \Rightarrow \tau_1, \Delta \quad \dots \quad \Gamma \Rightarrow \tau_n, \Delta}{\Gamma \Rightarrow \tau, \Delta}$$

is sound for any contexts Γ, Δ .

PROOF OF LEMMA A.1. Assume $\Gamma \Rightarrow \tau_1, \Delta, \dots, \Gamma \Rightarrow \tau_n, \Delta$ are valid, for any $\mathfrak{m} \in \mathbf{M}$ with $\mathfrak{m} \models \Gamma$, let $\mathcal{K}, \mathfrak{m} \not\models \tau'$ for all $\tau' \in \Delta$, we need to prove $\mathfrak{m} \models \tau$. From the assumption we have $\mathfrak{m} \models \tau_1, \dots, \mathfrak{m} \models \tau_n$. Then $\mathfrak{m} \models \tau$ is an immediate result. \square

LEMMA A.2. *Given labeled formulas $\tau_1, \dots, \tau_n, \tau$, for any label mapping $\mathfrak{m} \in \mathbf{M}$ with $\mathcal{K}, \mathfrak{m} \models \Gamma$, if $\mathfrak{m} \models \tau$ implies either $\mathfrak{m} \models \tau_1$ or ... or $\mathfrak{m} \models \tau_n$, then rule*

$$\frac{\Gamma, \tau_1 \Rightarrow \Delta \quad \dots \quad \Gamma, \tau_n \Rightarrow \Delta}{\Gamma, \tau \Rightarrow \Delta}$$

is sound for any contexts Γ, Δ .

PROOF OF LEMMA A.2. Assume $\Gamma, \tau_1 \Rightarrow \Delta, \dots, \Gamma, \tau_n \Rightarrow \Delta$ are valid. For any $\mathfrak{m} \in \mathbf{M}$ with $\mathfrak{m} \models \Gamma$, if $\mathfrak{m} \models \tau$, by the assumption, $\mathfrak{m} \models \tau_i$ for some $1 \leq i \leq n$. By the validity of $\Gamma, \tau_i \Rightarrow \Delta$, $\mathfrak{m} \models \Delta$. By the arbitrariness of \mathfrak{m} we know that $\Gamma, \tau \Rightarrow \Delta$ is valid. \square

Content of Theorem 5.3: Each rule from Pr_{dlp} in Table 2 is sound.

Below we only prove the soundness of the rules ($[\alpha]R$), ($[\alpha]L$) and (Sub). Other rules can be proved similarly based on the semantics of DLp formulas (Definition 4.6).

PROOF OF THEOREM 5.3. For rule ($[\alpha]R$), by Lemma A.1, it is sufficient to prove that for any $\mathfrak{m} \in \mathbf{M}$ with $\mathfrak{m} \models \Gamma$, if $\mathfrak{m} \models \sigma' : [\alpha']\phi$ for all $(\alpha', \sigma') \in \Phi$, then $\mathfrak{m} \models \sigma : [\alpha]\phi$. For any relation $\mathfrak{m}(\sigma) \xrightarrow{\alpha/\alpha_0} w_0$ with some $\alpha_0 \in \mathbf{P}$ and $w_0 \in \mathbf{S}$, by Item 1 of Definition 5.1, there is a label $\sigma_0 \in \mathbf{L}$ such that $\mathfrak{m}(\sigma_0) = w_0$ and $\models (\Gamma \Rightarrow (\alpha, \sigma) \longrightarrow (\alpha_0, \sigma_0))$. So $Pr_{dlp} \vdash (\Gamma \Rightarrow (\alpha, \sigma) \longrightarrow (\alpha_0, \sigma_0))$ (by Item 3 of Definition 5.1). Hence $(\alpha_0, \sigma_0) \in \Phi$, and by assumption, $\mathcal{K}, \mathfrak{m} \models \sigma_0 : [\alpha_0]\phi$. By the arbitrariness of α_0 and σ_0 , we have $\mathfrak{m} \models \sigma : [\alpha]\phi$ according to the semantics of formula $[\alpha]\phi$ (Definition 4.6).

For rule $([\alpha]L)$, by Lemma A.2, it is sufficient to prove that for any $\mathbf{m} \in \mathbf{M}$ with $\mathbf{m} \models \Gamma$, if $\mathbf{m} \models \sigma : [\alpha]\phi$, then $\mathbf{m} \models \sigma' : [\alpha']\phi$. For any execution path $\mathbf{m}(\sigma') \xrightarrow{\alpha'/\cdot} \dots \xrightarrow{\cdot/\downarrow} w$ of α' for some $w \in \mathcal{S}$, by the soundness of the side deduction $Pr_{dlp} \vdash (\Gamma \Rightarrow (\alpha, \sigma) \longrightarrow (\alpha', \sigma'), \Delta)$, $\mathbf{m}(\sigma) \xrightarrow{\alpha/\alpha'} \mathbf{m}(\sigma')$ is a relation on \mathcal{K} , so $\mathbf{m}(\sigma) \xrightarrow{\alpha/\alpha'} \mathbf{m}(\sigma') \xrightarrow{\alpha'/\cdot} \dots \xrightarrow{\cdot/\downarrow} w$ is an execution path of α . By the semantics of the dynamic formulas (Definition 4.6), we obtain the result.

For rule (Sub) , we need to prove that the validity of $\Gamma \Rightarrow \Delta$ implies the validity of $Sub(\Gamma) \Rightarrow Sub(\Delta)$. For any $\mathbf{m} \in \mathbf{M}$ satisfying $\mathbf{m} \models Sub(\Gamma)$, by Definition 4.12, there exists a \mathbf{m}' such that for any formula $\sigma : \phi \in \Gamma \cup \Delta$, $\mathbf{m}'(\sigma) = \mathbf{m}(Sub(\sigma))$. So $\mathcal{K}, \mathbf{m}' \models \Gamma$. Since $\Gamma \Rightarrow \Delta$ is valid, $\mathcal{K}, \mathbf{m}' \models \sigma_0 : \phi_0$ for some $\sigma_0 : \phi_0 \in \Delta$. By that $\mathbf{m}'(\sigma_0) = \mathbf{m}(Sub(\sigma_0))$, $\mathbf{m} \models Sub(\sigma_0) : \phi_0$ with $Sub(\sigma_0) : \phi_0 \in Sub(\Delta)$. By the arbitrariness of \mathbf{m} , $Sub(\Gamma) \Rightarrow Sub(\Delta)$ is valid. \square

Content of Proposition 7.5: \preceq_m is a partial-order relation.

Before proving Proposition 7.5, we firstly review the notion of *partial-order relation*.

A relation \preceq on a set S is *partially ordered*, if it satisfies the following properties: (1) Reflexivity. $t \preceq t$ for each $t \in S$. (2) Anti-symmetry. For any $t_1, t_2 \in S$, if $t_1 \preceq t_2$ and $t_2 \preceq t_1$, then t_1 and t_2 are the same element in S , we denote by $t_1 = t_2$. (3) Transitivity. For any $t_1, t_2, t_3 \in S$, if $t_1 \preceq t_2$ and $t_2 \preceq t_3$, then $t_1 \preceq t_3$. $t_1 \prec t_2$ is defined as $t_1 \preceq t_2$ and $t_1 \neq t_2$.

Recall that we use \preceq_s to represent the suffix relation between execution paths. \preceq_s is obviously a partial-order relation.

PROOF OF PROPOSITION 7.5. The reflexivity is trivial. The transitivity can be proved by the definition of ‘replacements’ as described in Definition 7.4 and the transitivity of relation \prec_s . Below we only prove the anti-symmetry.

For any finite sets D_1, D_2 of finite paths, if $D_1 \preceq_m D_2$ but $D_1 \neq D_2$, let $f_{D_1, D_2} : D_1 \rightarrow D_2$ be the function defined such that for any $tr \in D_1$, either (1) $f_{D_1, D_2}(tr) \approx_s tr$; or (2) tr is one of the replacements of a replaced element $f_{D_1, D_2}(tr)$ in D_2 with $tr \prec_s f_{D_1, D_2}(tr)$.

For the anti-symmetry, suppose $C_1 \preceq_m C_2$ and $C_2 \preceq_m C_1$ but $C_1 \neq C_2$. Let $tr \in C_1$ but $tr \notin C_2$. Then from $C_1 \preceq_m C_2$, we have $tr \prec_s f_{C_1, C_2}(tr)$. If $f_{C_1, C_2}(tr) \in C_1$, then we must have $f_{C_1, C_2}(tr) \prec_m f_{C_1, C_2}(f_{C_1, C_2}(tr))$ because $f_{C_1, C_2}(tr)$ is already a replaced element in C_2 . If $f_{C_1, C_2}(tr) \notin C_1$, then by $C_2 \preceq_m C_1$, we have $f_{C_1, C_2}(tr) \prec_s f_{C_2, C_1}(f_{C_1, C_2}(tr))$. Continuing this process by considering $f_{C_1, C_2}(f_{C_1, C_2}(tr))$ or $f_{C_2, C_1}(f_{C_1, C_2}(tr))$ and further elements, we in fact can construct an infinite descent sequence like $tr \prec_s f_{C_1, C_2}(tr) \prec_s f_{C_1, C_2}(f_{C_1, C_2}(tr)) \prec_s \dots$ w.r.t. relation \preceq_s , which violates its well-foundedness. So the only possibility is $C_1 = C_2$. \square

Content of Lemma 7.9: 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 (v, v') of an invalid derivation path, where $\phi, \phi' \in \mathcal{F}_{dlp}$. For any set $EX(\mathbf{m}(\sigma), \phi)$ of $\sigma : \phi$ w.r.t. a counter-example mapping \mathbf{m} of v , there exists a counter-example mapping \mathbf{m}' of v' and a set $EX(\mathbf{m}'(\sigma'), \phi')$ of $\sigma' : \phi'$ such that $EX(\mathbf{m}'(\sigma'), \phi') \preceq_m EX(\mathbf{m}(\sigma), \phi)$. Moreover, if $(\sigma : \phi, \sigma' : \phi')$ is a progressive step, then $EX(\mathbf{m}'(\sigma'), \phi') \prec_m EX(\mathbf{m}(\sigma), \phi)$.

PROOF OF LEMMA 7.9. Consider the rule application from node v , we only consider the cases when it is an instance of rule $([\alpha]R)$, rule $([\alpha]L)$, rule (Sub) , and rule $(\wedge R)$, and when the first element of the CP pair we consider is their target formula.

Case for rule $([\alpha]R)$: If from node v rule $([\alpha]R)$ is applied with $\tau \stackrel{def}{=} \sigma : [\alpha]\phi$ the target formula, let $\tau' = (\sigma' : [\alpha']\phi)$ for some α' and σ' , so $v = (\Gamma \Rightarrow \tau, \Delta)$ and $v' = (\Gamma \Rightarrow \tau', \Delta)$. (In this case, (v, v') is already a progressive step.) Since \mathbf{m} is a counter-example of v , $\mathbf{m} \not\models \tau$, so $mex(\mathbf{m}(\sigma), \alpha) \neq \emptyset$. Thus $EX(\mathbf{m}(\sigma), [\alpha]\phi) \neq \emptyset$. By the soundness of rule $([\alpha]R)$ and the assumption that $Pr_{dlp} \vdash \Gamma \Rightarrow (\alpha, \sigma) \longrightarrow (\alpha', \sigma'), \Delta$, for each path $tr = \mathbf{m}(\sigma')s_1 \dots s_n \in EX(\mathbf{m}(\sigma'), [\alpha']\phi)$ ($n \geq 0$),

path $m(\sigma)tr \in EX(m(\sigma), [\alpha]\phi)$ has tr as its proper suffix. By Definition 7.2, $EX(m(\sigma), [\alpha]\phi)$ and $EX(m(\sigma'), [\alpha']\phi)$ are also finite. Therefore $EX(m(\sigma'), [\alpha']\phi) \prec_m EX(m(\sigma), [\alpha]\phi)$.

Case for rule $([\alpha]L)$: If from node v rule $([\alpha]L)$ is applied with $\tau =_{df} \sigma : [\alpha]\phi$ the target formula, let $\tau' = (\sigma' : [\alpha']\phi)$ for some α' and σ' , so $v = (\Gamma \Rightarrow \tau, \Delta)$ and $v' = (\Gamma \Rightarrow \tau', \Delta)$. By the soundness of rule $([\alpha]L)$ and the assumption that $Pr_{dlp} \vdash \Gamma \Rightarrow (\alpha, \sigma) \rightarrow (\alpha', \sigma'), \Delta$, for each path $tr = m(\sigma')s_1 \dots s_n \in EX(m(\sigma'), [\alpha']\phi)$ ($n \geq 0$), path $m(\sigma)tr \in EX(m(\sigma), [\alpha]\phi)$ has tr as its proper suffix. By Definition 7.2, $EX(m(\sigma), [\alpha]\phi)$ and $EX(m(\sigma'), [\alpha']\phi)$ are also finite. Therefore $EX(m(\sigma'), [\alpha']\phi) \preceq_m EX(m(\sigma), [\alpha]\phi)$, where the equivalence relation $=$ holds only when $EX(m(\sigma), [\alpha]\phi) = \emptyset$. When (τ, τ') is progressive, which means that we also have the derivation $Pr_{dlp} \vdash (\Gamma \Rightarrow \alpha \Downarrow \sigma, \Delta)$, then $mex(m(\sigma), \alpha) \neq \emptyset$. This means that $EX(m(\sigma), [\alpha]\phi) \neq \emptyset$. Therefore $EX(m(\sigma'), [\alpha']\phi) \prec_m EX(m(\sigma), [\alpha]\phi)$.

Case for rule (Sub) : If from node v a substitution rule (Sub) is applied, let $\tau = Sub(\sigma) : \phi$ be the target formula of v , then $\tau' = \sigma : \phi$. By Definition 4.12, for the label mapping m , there exists a m' such that $m'(\sigma) = m(Sub(\sigma))$. So $EX(m(Sub(\sigma)), \phi) = EX(m'(\sigma), \phi)$.

Case for rule $(\wedge R)$: If from node v rule $(\wedge R)$ is applied, let $\tau = \sigma : \phi \wedge \psi$ be the target formula of v , and $\tau' = \sigma : \phi$. By the definition of EX (Definition 7.8), we have $EX(m(\sigma), \phi) \subseteq EX(m(\sigma), \sigma : \phi \wedge \psi)$, so $EX(m(\sigma), \phi) \preceq_m EX(m(\sigma), \sigma : \phi \wedge \psi)$. \square

From the case of rule $([\alpha]L)$ in the above proof, we see that derivation $Pr_{dlp} \vdash (\Gamma \Rightarrow \alpha \Downarrow \sigma, \Delta)$ imposes $EX(m(\sigma), [\alpha]\phi) \neq \emptyset$, which is the key to prove the strict relation \prec_m between $EX(m(\sigma'), [\alpha']\phi)$ and $EX(m(\sigma), [\alpha]\phi)$.

Content of Proposition 5.10: Given a sound rule of the form

$$\frac{\Gamma_1 \Rightarrow \Delta_1 \quad \dots \quad \Gamma_n \Rightarrow \Delta_n}{\Gamma \Rightarrow \Delta}, \quad n \geq 1,$$

in which all formulas are unlabeled, then the rule

$$\frac{\sigma : \Gamma_1 \Rightarrow \sigma : \Delta_1 \quad \dots \quad \sigma : \Gamma_n \Rightarrow \sigma : \Delta_n}{\sigma : \Gamma \Rightarrow \sigma : \Delta}$$

is sound for any label $\sigma \in free(L, \Gamma \cup \Delta \cup \Gamma_1 \cup \Delta_1 \cup \dots \cup \Gamma_n \cup \Delta_n)$.

PROOF OF PROPOSITION 5.10. Let $A = \Gamma \cup \Delta \cup \Gamma_1 \cup \Delta_1 \cup \dots \cup \Gamma_n \cup \Delta_n$. Assume sequents $\sigma : \Gamma_i \Rightarrow \sigma : \Delta_i$ ($1 \leq i \leq n$) are valid, we need to prove that sequent $\sigma : \Gamma \Rightarrow \sigma : \Delta$ is valid. First, notice that each sequent $\Gamma_i \Rightarrow \Delta_i$ ($1 \leq i \leq n$) is valid. Because since $\sigma \in free(L, A)$, for any $s \in \mathcal{S}$ such that $s \models \Gamma_i$, there is a label mapping denoted by $m_s \in \mathbf{M}$ with $s =_A m_s(\sigma)$ such that $m_s(\sigma) \models \Gamma_i$ (Definition 5.6), so $m_s \models \sigma : \Gamma_i$ (By Definition 4.13). By the validity of the sequent $\sigma : \Gamma_i \Rightarrow \sigma : \Delta_i$, $m_s \models \sigma : \phi$ for some $\phi \in \Delta_i$, which means $m_s(\sigma) \models \phi$. Hence $s \models \phi$. From the validity of $\Gamma_i \Rightarrow \Delta_i$, we get that $\Gamma \Rightarrow \Delta$ is valid. For any $m \in \mathbf{M}$, if $m \models \sigma : \Gamma$, then $m(\sigma) \models \Gamma$ (Definition 4.13). By the validity of $\Gamma \Rightarrow \Delta$, we have $m(\sigma) \models \phi$ for some $\phi \in \Delta$. But this just means $m \models \sigma : \phi$. Therefore, we have concluded that $\sigma : \Gamma \Rightarrow \sigma : \Delta$ is valid. \square

Content of Lemma 7.16: Under the same conditions as in Theorem 7.15, for any valid sequent of the form: $\Gamma \Rightarrow \sigma : [\langle \alpha \rangle]\phi$, $Pr_{dlp} \vdash (\Gamma \Rightarrow \sigma : [\langle \alpha \rangle]\phi)$.

PROOF OF LEMMA 7.16. Let $v =_{df} (\Gamma \Rightarrow \sigma : [\langle \alpha \rangle]\phi)$. We proceed by simultaneous induction on the number M_v of the modalities $[\cdot]$ or $\langle \cdot \rangle$ in v and the maximum number N_α of the different programs along the program sequences starting from α for all labels and contexts (see Definition 7.13).

Base case. $M_v = N_\alpha = 1$, so in the sequent v there is only one modality, which is $[\langle\alpha\rangle]$, and program α is either \downarrow or a loop program $\neq\downarrow$ that can only perform transitions of the form: $(\alpha, \sigma) \longrightarrow (\alpha, \sigma')$ for some α' under some context. The case for $\alpha = \downarrow$ is trivial, as by applying rule $([\downarrow])$ or rule $(\langle\downarrow\rangle)$ and rule (Ter) , we can directly obtain the result. For the case when $\alpha \neq\downarrow$ is a loop program, the proof is just a special case to the proof for the step case as follows.

Step case. This case is divided into two parts. Part I describes the process of constructing the derivation of sequent v (named “Proc” below); Part II further proves that this derivation is cyclic.

Part I: Without loss of generality, suppose α is a loop program. For the label $\sigma \in L$ and the context Γ , by Definition 7.14, there exist a label σ' , a context Γ' and a substitution η satisfying that

- (a) $\sigma = \eta(\sigma')$ and $\Gamma = \eta(\Gamma')$;
- (b) $\models (\Gamma \Rightarrow \sigma \Downarrow \alpha)$ implies $\models (\Gamma' \Rightarrow \sigma' \Downarrow \alpha)$;
- (c) for each α -loop sequence: $\Gamma' : (\alpha, \sigma', \Gamma_1) \dots (\alpha, \sigma'', \Gamma_n)$ ($n \geq 1$), there exist a context Γ'' and a substitution ξ such that $\Gamma'' = \xi(\Gamma')$, $\sigma'' = \xi(\sigma')$ and $\models (\Gamma_n \Rightarrow \Gamma'')$.

From $\Gamma \Rightarrow \sigma : [\langle\alpha\rangle]\phi$, we can have the following derivation named “Proc”:

$$\begin{array}{c}
 \begin{array}{c}
 \frac{7 : \Gamma' \Rightarrow \sigma' : [\langle\alpha\rangle]\phi}{6 : \Gamma'' \Rightarrow \sigma'' : [\langle\alpha\rangle]\phi} \text{ (Sub)} \\
 \frac{\text{(Ind. Hypo.)}}{8 : \Gamma_n \Rightarrow \Gamma''} \quad \frac{6 : \Gamma'' \Rightarrow \sigma'' : [\langle\alpha\rangle]\phi}{\Gamma_n, \Gamma'' \Rightarrow \sigma'' : [\langle\alpha\rangle]\phi} \text{ (WkR)} \\
 \frac{8 : \Gamma_n \Rightarrow \Gamma''}{4 : \Gamma_n \Rightarrow \sigma'' : [\langle\alpha\rangle]\phi} \text{ (Cut)}
 \end{array}
 \quad
 \begin{array}{c}
 \text{(Ind. Hypo.)} \\
 \frac{10 : \Lambda \Rightarrow \zeta' : [\langle\beta'\rangle]\phi}{9 : \Lambda \Rightarrow \zeta : [\langle\beta\rangle]\phi} \text{ ([}\langle\alpha\rangle\text{)]R}
 \end{array}
 \\
 \frac{4 : \Gamma_n \Rightarrow \sigma'' : [\langle\alpha\rangle]\phi}{\vdots} \text{ ([}\langle\alpha\rangle\text{)]R}
 \quad
 \frac{10 : \Lambda \Rightarrow \zeta' : [\langle\beta'\rangle]\phi}{\vdots} \text{ ([}\langle\alpha\rangle\text{)]R}
 \\
 \frac{\vdots}{3 : \Gamma_1 \Rightarrow \sigma' : [\langle\alpha\rangle]\phi} \text{ ([}\langle\alpha\rangle\text{)]R}
 \quad
 \frac{\vdots}{\vdots}
 \quad
 \frac{\vdots}{\vdots}
 \\
 \frac{3 : \Gamma_1 \Rightarrow \sigma' : [\langle\alpha\rangle]\phi}{2 : \Gamma' \Rightarrow \sigma' : [\langle\alpha\rangle]\phi} \text{ ([}\langle\alpha\rangle\text{)]R}
 \quad
 \frac{10 : \Lambda \Rightarrow \zeta' : [\langle\beta'\rangle]\phi}{9 : \Lambda \Rightarrow \zeta : [\langle\beta\rangle]\phi} \text{ ([}\langle\alpha\rangle\text{)]R}
 \\
 \frac{2 : \Gamma' \Rightarrow \sigma' : [\langle\alpha\rangle]\phi}{1 : \Gamma \Rightarrow \sigma : [\langle\alpha\rangle]\phi} \text{ (Sub)}
 \end{array}$$

The derivation from node 1 to 2 is according to (a). From node 2, the context Γ' is strengthened for the derivation of each α sequence starting from (α, σ') . This process can be realized by applying rule (Cut) and the other rules for labeled propositional logical formulas as shown in Table 2. Each α -loop sequence: $\Gamma' : (\alpha, \sigma', \Gamma_1) \dots (\alpha, \sigma'', \Gamma_n)$ ($n \geq 1$) corresponds to a derivation, named “Sub-proc 1”, like the one from node 3 as shown in Proc. From node 3 to 4 includes a series of derivation steps that symbolically executes the α -loop sequence by applying the rule $([\alpha]R)$ or $(\langle\alpha\rangle R)$ (denoted by $([\langle\alpha\rangle]R)$) and also the other rules for strengthening the contexts $\Gamma_2, \dots, \Gamma_n$. The (c) above provides the evidence for both the derivation from node 6 to 7, and the validation of node 8. The derivation from node 2 to 10 is a general case of an α sequence: $\Gamma' : (\alpha, \sigma', \cdot) \dots (\beta, \zeta, \Lambda)(\beta', \zeta', \Lambda)$, where from (β', ζ') under Λ program α can never be reached. We name the derivation like the one from node 10 here as “Sub-proc 2”.

Part II: Now we show that the derivation Proc is actually a cyclic proof. We firstly show that the whole proof Proc is a preproof (i.e. a finite tree structure with buds), which is based on the following 3 proof statements.

- (1) The derivation part as shown above in Proc is finite. On one hand, by the finiteness of set Pr_{dlp} , each derivation step must have finite premises; On the other hand, by Lemma 7.17, from node 1 there is a finite number of α sequences (by selecting a set of contexts for each core α sequence). This means that the number of the branches Sub-proc 1 is finite.

- (2) Each Sub-proc 1 is a preproof branch. On one hand, By that $\xi(\Gamma') = \Gamma''$ (see (c) above), the number of modalities in Γ'' is the same as that in Γ' . On the other hand, by Item 4 of Definition 5.1, when we strengthen the context Γ' during the derivation from node 3 to 4, we can make sure that compared to Γ' there is no more dynamic formulas added in $\Gamma_1, \dots, \Gamma_n$. Therefore, the number of modalities in the sequent $\Gamma_n \Rightarrow \Gamma''$ equals to that in sequent ν , and is thus strictly less than M_ν . So, by induction hypothesis, $Pr_{dlp} \vdash (\Gamma_n \Rightarrow \Gamma'')$.
- (3) Each Sub-proc 2 is a cyclic proof branch. For any derivation step like the one from node 9 to 10, since from (β', ζ') under Λ the program α can never be reached, clearly we have $N_{\beta'} < N_\alpha$, because except α itself, any program that β' can reach can be reached by α (through the α sequence: $\Gamma' : (\alpha, \sigma', \cdot) \dots (\beta, \zeta, \Lambda)(\beta', \zeta', \Lambda)$). So by induction hypothesis, $Pr_{dlp} \vdash (\Lambda \Rightarrow \zeta' : [\langle \beta' \rangle] \phi)$.

It remains to show that in Proc along every derivation path, there exists a progressive derivation trace. Observing that in Proc, every derivation path must at least pass through a preproof branch Sub-proc 1 for infinite times. So, it is enough to show that each Sub-proc 1 has a progressive derivation trace. In a Sub-proc 1, consider two cases:

- (i) If the modality is $\langle \alpha \rangle$, by Definition 5.5, in every inference of rule $([\alpha]R)$, the CP pair on the right of both sequents is a progressive step.
- (ii) If the modality is $\langle \alpha \rangle$, since $\models (\Gamma \Rightarrow \sigma : \langle \alpha \rangle \phi), \models (\Gamma \Rightarrow \sigma \Downarrow \alpha)$. By (b) above, $\models (\Gamma' \Rightarrow \sigma' \Downarrow \alpha)$. By the completeness w.r.t. \mathfrak{F}_{ter} (Item 3 of Definition 5.1), $Pr_{dlp} \vdash (\Gamma' \Rightarrow \sigma' : \langle \alpha \rangle \phi)$. So according to Definition 7.2, in the first inference of rule $(\langle \alpha \rangle R)$, the CP pair on the right of both sequents is a progressive step.

In both cases above, the progressive derivation trace is: $\sigma' : [\langle \alpha \rangle] \phi$ in node 3, ..., $\sigma'' : [\langle \alpha \rangle] \phi$ in node 4, $\sigma'' : [\langle \alpha \rangle] \phi$, $\sigma'' : [\langle \alpha \rangle] \phi$ in node 6, $\sigma' : [\langle \alpha \rangle] \phi$ in node 7, ... as shown in Proc.

□

B A Cyclic Deduction of An Esterel Program

Esterel [12] is a synchronous programming language for reactive systems. Below we first introduce the semantics of an Esterel program, then we explain why it needs extra program transformations in traditional verification frameworks. Lastly, we explain how to express Esterel programs in DLp and give a cyclic deduction of this program.

Note that the introduction we provide below is informal and does not cover all aspects of the semantics of Esterel. But it is enough to clear our point.

B.1 An Esterel Program in DLp

We consider a synchronous program E of an instantiation P_E of programs written in Esterel language [12]:

$$E =_{df} \{ trap A \parallel B end \},$$

where

$$\begin{aligned} A &=_{df} \{ loop (emit S(0) ; x := x - S ; if x = 0 then exit end ; pause) end \} \\ B &=_{df} \{ loop (emit S(1) ; pause) end \}. \end{aligned}$$

The behaviour of a synchronous program is characterized by a sequence of *instances*. At each instance, several (atomic) executions of a program may occur. The value of each variable is unique in an instance. When several programs run in parallel, their executions at one instance are thought to occur simultaneously. In this manner, the behaviour of a parallel synchronous program is deterministic.

In this example, the behaviour of the program E is illustrated as follows:

World w	$w(x)$	$w(S)$	n th Instance
w_1	3	1	1
w_2	2	1	2
w_3	1	1	3
w_4	0	1	4

Table 6. Transitional Behaviours of Program E Starting From w_1

- x, S are two variables. x is a local variable with \mathbb{Z} as its domain, S is a “signal” whose domain is $\mathbb{Z} \cup \{\perp\}$, with \perp indicating the absence of a signal.
- The key word *pause* marks the end of an instance, when all signals are set to \perp , representing the state “absence”.
- Signal emission *emit* $S(e)$ means assigning the value of an expression e to a signal S and broadcasts the value. $x := x - S$ is the usual assignment as in FODL.
- At each instance, program A firstly emits signal S with value 0 and subtracts x with the current value of S ; then checks if $x = 0$. While program B emits signal S with value 1. The value of signal S in one instance should be the sum of all of its emitted values by different current programs. So the value of S should be $1 + 0 = 1$.
- The whole program E continues executing until condition $x = 0$ is satisfied, when *exit* terminates the whole program by jumping out of the *trap* statement.

Starting from an initial world w_1 with $w_1(x) = 3$ and $w_1(S) = 1$, we have an execution path $w_1 w_2 w_3 w_4$ of program E explained in Table 6, where we omit the intermediate forms of programs during the execution.

B.2 Prior Program Transformations in Esterel Programs

In Esterel, the behaviour of a parallel program is usually not true interleaving. There exist data dependencies between its processes. For instance, in the program E above, the assignment $x := x - S$ can only be executed after all values of S in both programs A and B are collected. In other words, $x := x - S$ can only be executed after *emit* $S(0)$ and *emit* $S(1)$. For a synchronous program like this, additional program transformations are mandatory (cf. [24]). We need to first transform the program E , for example, into a sequential one as:

$$E' = \{\text{trap } C \text{ end}\},$$

where

$$C =_{df} \{\text{loop } \text{emit } S(0) ; \text{emit } S(1) ; x := x - S ; \text{if } x = 0 \text{ then } \text{exit} \text{ end} ; \text{pause} \text{ end}\}.$$

In C , we collect all micro steps happen in an instance from both A and B , in a correct order. In [24], E' is called an STA program. Note that such a prior transformation can be very heavy, since one can imagine that the behaviour of a parallel Esterel program can be very complex (e.g. [10]).

B.3 Instantiation of DLp in Esterel Programs

The instantiation process is similar to while programs. Let \mathbf{P}_E be the set of Esterel programs and Var_E be the set of local variables and signals in Esterel. We assume a PLK structure $\mathcal{K}_E = (\mathcal{S}_E, \longrightarrow, \mathcal{I}_E)$ for \mathbf{P}_E , where each world $w \in \mathcal{S}_E$ is a mapping $w : \text{Var}_E \rightarrow (\mathbb{Z} \cup \{\perp\})$ that maps each local variable to an integer and maps each signal to a value of $\mathbb{Z} \cup \{\perp\}$. A program configuration $\sigma \in \mathbf{L}_E$ in \mathbf{P}_E , as a label, is defined to capture the meaning of the structure

$$\{x_1 \mapsto e_1 \mid \dots \mid x_n \mapsto e_n\} \ (n \geq 1),$$

where the only difference from that of a while program is that it is a stack (with $x_n \mapsto e_n$ the top element), allowing several local variables with the same name. For example, configuration $\{x \mapsto 5 \mid y \mapsto 1 \mid y \mapsto 2\}$ has two different local variables y , storing the values 1 and 2 respectively.

Given a world $w \in \mathcal{S}_E$, a label mapping $\mathbf{m}_w \in \mathbf{M}_E$ (w.r.t. w) is defined such that for any configuration $\sigma \in \mathbf{L}_E$ of the form: $\{x_1 \mapsto e_1 \mid \dots \mid x_n \mapsto e_n\}$, $\mathbf{m}_w(\sigma)$ is a world satisfying that

- (1) $\mathbf{m}_w(\sigma)(x_i) = w(e_j)$ with $n \geq j \geq i \geq 1$ and j the largest index for $x_i \mapsto e_j$ in σ (i.e. the right-most value of variable x_i);
- (2) $\mathbf{m}_w(\sigma)(y) = w(y)$ for other variable $y \in \text{Var}_E$,

where $w(e)$ for an expression e is defined similarly as in while programs (see Example 4.10). For example, we have $\mathbf{m}_w(\{x \mapsto 5 \mid y \mapsto 1 \mid y \mapsto 2\})(y) = w(2) = 2$ for any \mathbf{m}_w .

We omit the details of the set $(\mathbf{Pr}_{op})_E$ of rules for the operational semantics of Esterel programs, as they are too complex (cf. [48]).

B.4 A Cyclic Deduction of Program E

In DLp, program E can be directly reasoned about without additional program transformations. This is achieved because DLp supports a cyclic reasoning directly based on the operational semantics. During the following derivation, we see that the outside loop structure of E (which is C above after the transformations) is actually reflected by the cyclic derivation tree itself.

We prove the property

$$v_2 \stackrel{df}{=} \sigma_1 : x > 0 \Rightarrow \sigma_1 : \langle E \rangle \text{true},$$

which says that under configuration $\sigma_1 = \{x \mapsto v, S \mapsto \perp\}$, with v a fresh variable representing an initial value of x , if $x > 0$, then E can finally terminate.

The derivations of v_2 is depicted in Table 7. The symbolic executions of program E rely on rule

$$\frac{\Gamma \Rightarrow \sigma' : \langle \alpha' \rangle \phi, \Delta}{\Gamma \Rightarrow \sigma : \langle \alpha \rangle \phi, \Delta} \quad (\langle \alpha \rangle), \text{ if } \text{Pr}_{dlp} \vdash (\Gamma \Rightarrow (\alpha, \sigma) \longrightarrow (\alpha', \sigma'), \Delta),$$

which can be derived by rules $([\alpha]L)$, $(\neg R)$ and $(\neg L)$ from Table 2. We omit all the side deductions of the program transitions and terminations in the instances of rule $(\langle \alpha \rangle)$.

From node 2 to 3 is a progressive step, where to see that program $\sigma_2 \Downarrow \text{trap}((x := x - S; A'); A) \parallel B \text{ end}$ terminates, informally, we observe that the value of variable x decreases by 1 (by executing $x := x - S$) in each loop so that statement *exit* is finally reached. From node 13 to 14 and node 15 to 16, rule

$$\frac{\Gamma, \phi' \Rightarrow \Delta}{\Gamma, \phi \Rightarrow \Delta} \quad (LE), \text{ if } \phi \rightarrow \phi' \in \mathbf{F} \text{ is valid}$$

$ \begin{array}{c} \frac{16}{15} \text{ (LE)} \\ \frac{14}{13} \text{ (Sub)} \\ \frac{12}{11} \text{ (LE)} \\ \frac{10}{9} \text{ (Ter)} \\ \frac{8}{7} \text{ ((\langle \rangle)} \\ \frac{6}{5} \text{ ((\langle \rangle)} \\ \frac{4}{3} \text{ ((\langle \rangle)} \\ \frac{2}{1} \text{ ((\langle \rangle)} \\ v_2 : 1 \end{array} $	$ \begin{array}{c} \frac{16}{15} \text{ (LE)} \\ \frac{14}{13} \text{ (Sub)} \\ \frac{12}{11} \text{ (LE)} \\ \frac{10}{9} \text{ (Ter)} \\ \frac{8}{7} \text{ ((\langle \rangle)} \\ \frac{6}{5} \text{ ((\langle \rangle)} \\ \frac{4}{3} \text{ ((\langle \rangle)} \\ \frac{2}{1} \text{ ((\langle \rangle)} \\ v_2 : 1 \end{array} $
$ \begin{array}{l} 1: \sigma_1 : x > 0 \\ 2: \sigma_1 : x > 0 \\ 3: \sigma_1 : x > 0 \\ 4: \sigma_1 : x > 0 \\ 5: \sigma_1 : x > 0 \\ 17: \sigma_1 : x > 0 \end{array} $	$ \begin{array}{l} \Rightarrow \sigma_1 : \langle \text{trap } A \parallel B \text{ end} \rangle \text{true} \\ \Rightarrow \sigma_2 : \langle \text{trap } ((x := x - S; A') ; A) \parallel B \text{ end} \rangle \text{true} \\ \Rightarrow \sigma_3 : \langle \text{trap } ((x := x - S; A') ; A) \parallel (\text{pause} ; B) \text{ end} \rangle \text{true} \\ \Rightarrow \sigma_4 : \langle \text{trap } (A' ; A) \parallel (\text{pause} ; B) \text{ end} \rangle \text{true} \\ \Rightarrow \sigma_4 : \langle \text{trap } (A' ; A) \parallel (\text{pause} ; B) \text{ end} \rangle \text{true}, \sigma_1 : (x - 1 \neq 0 \vee x - 1 = 0) \\ \Rightarrow \sigma_1 : (x - 1 \neq 0 \vee x - 1 = 0) \end{array} $
$ \begin{array}{l} 6: \sigma_1 : x > 0, \sigma_1 : (x - 1 \neq 0 \vee x - 1 = 0) \\ 7: \sigma_1 : x > 0, \sigma_1 : x - 1 \neq 0 \\ 12: \sigma_1 : x > 0, \sigma_1 : x - 1 \neq 0 \\ 13: \sigma_1 : x > 0, \sigma_1 : x - 1 \neq 0 \\ 14: \sigma_5 : x + 1 > 0, \sigma_5 : x \neq 0 \\ 15: \sigma_1 : x + 1 > 0, \sigma_1 : x \neq 0 \\ 16: \sigma_1 : x > 0 \end{array} $	$ \begin{array}{l} \Rightarrow \sigma_4 : \langle \text{trap } (A' ; A) \parallel (\text{pause} ; B) \text{ end} \rangle \text{true} \\ \Rightarrow \sigma_4 : \langle \text{trap } (A' ; A) \parallel (\text{pause} ; B) \text{ end} \rangle \text{true} \\ \Rightarrow \sigma_4 : \langle \text{trap } (\text{pause} ; A) \parallel (\text{pause} ; B) \text{ end} \rangle \text{true} \\ \Rightarrow \sigma_5 : \langle \text{trap } A \parallel B \text{ end} \rangle \text{true} \\ \Rightarrow \sigma_5 : \langle \text{trap } A \parallel B \text{ end} \rangle \text{true} \\ \Rightarrow \sigma_1 : \langle \text{trap } A \parallel B \text{ end} \rangle \text{true} \\ \Rightarrow \sigma_1 : \langle \text{trap } A \parallel B \text{ end} \rangle \text{true} \end{array} $
$ \begin{array}{l} 8: \sigma_1 : x > 0, \sigma_1 : x - 1 = 0 \\ 9: \sigma_1 : x > 0, \sigma_1 : x - 1 = 0 \\ 10: \sigma_1 : x > 0, \sigma_1 : x - 1 = 0 \end{array} $	$ \begin{array}{l} \Rightarrow \sigma_4 : \langle \text{trap } (A' ; A) \parallel (\text{pause} ; B) \text{ end} \rangle \text{true} \\ \Rightarrow \sigma_5 : \langle \downarrow \rangle \text{true} \\ \Rightarrow \sigma_5 : \text{true} \end{array} $

is applied, which can be derived by the following derivations:

$$\frac{\frac{\Gamma, \phi' \Rightarrow \Delta}{\Gamma, \phi, \phi' \Rightarrow \Delta} \quad (WkL) \quad \frac{\Gamma, \phi \Rightarrow \phi', \Delta}{\Gamma, \phi \Rightarrow \Delta} \quad \begin{matrix} (Ter) \\ (Cut) \end{matrix}}{\Gamma, \phi \Rightarrow \Delta}$$

is applied, with $(\cdot)[e/x]$ an instantiation of the substitution *Sub* of labels (Definition 4.12). It is defined just as that in the example of Section 6.1. Observe that sequent 14 can be written as:

$$\sigma_1[v - 1/v] : x + 1 > 0, \sigma_1[v - 1/v] : x \neq 0 \Rightarrow \sigma_1[v - 1/v] : \langle trap A \parallel B end \rangle true.$$

C An Encoding of Separation Logic in DLp

We instantiate DLp to express separation logic [51] — an extension of Hoare logic for reasoning about shared mutable program data structures. The instantiated theory is called DLp-SP. Below we only deal with a part of separation logic, but it is enough to clear our point.

Separation Logic. In the following, we assume the readers are familiar with separation logic and we only give an informal explanations of its semantics. For simplicity, we only introduce partial separation logic primitives: the atomic formula $e \mapsto e'$ and critical operator $*$ for formulas, and the atomic statements $x := \text{cons}(e)$, $x := [e]$, $[e] := e'$ and $\text{disp}(e)$ for programs. We omit another critical operator \multimap in formulas and the compositional programs that vary from case to case.

Below we follow some conventions of notations: Given a partial function $f : A \mapsto B$, we use $\text{dom}(f)$ to denote the domain of f . For a set C , partial function $f|_C : A \mapsto B$ is the function f restricted on domain $\text{dom}(f) \cap C$. $f[x \mapsto e]$ represents the partial function that maps x to e , and maps the other variables in its domain to the same value as f does. Let $\mathbb{V} = \mathbb{Z} \cup \text{Addr}$ be the set of values, where Addr is a set of addresses. We assume Addr to be expressed with an infinite set of integer numbers. In separation logic, a store $s : V \rightarrow \mathbb{V}$ is a function that maps each variable to a value of \mathbb{V} , a heap $h : \text{Addr} \mapsto \mathbb{V}$ is a partial function that maps an address to a value of \mathbb{V} , expressing that the value is stored in a memory indicated by the address. $\text{dom}(h)$ is a finite subset of Addr . A state is a store-heap pair (s, h) . The *disjoint relation* $h_1 \perp h_2$ is defined if $\text{dom}(h_1) \cap \text{dom}(h_2) = \emptyset$.

Here we informally explain the semantics of each primitive. Given a state (s, h) , statement $x := \text{cons}(e)$ allocates a memory addressed by a new integer n in h to store the value of expression e (thus obtaining a new heap $h \cup \{(n, s(e))\}$ where $n \notin \text{dom}(h)$), and assigns n to x . Statement $x := [e]$ assigns the value of the address e in h (i.e. $h(s(e))$) to variable x . $[e] := e'$ means to assign the value e' to the memory of the address e in h (i.e. obtaining a new heap $h[s(e) \mapsto s(e')]$). $\text{disp}(e)$ means to de-allocate the memory of address e in the heap (i.e. obtaining a new heap $h|_{\text{dom}(h) \setminus \{s(e)\}}$). Formula $e \mapsto e'$ means that value e' is stored in the memory of address e . Given a state (s, h) , $s, h \models e \mapsto e'$ is defined if $h(s(e)) = s(e')$. For any separation logical formulas ϕ and ψ , $s, h \models \phi * \psi$ if there exist heaps h_1, h_2 such that $h = h_1 \cup h_2$, $h_1 \perp h_2$, and $s, h_1 \models \phi$ and $s, h_2 \models \psi$.

Example C.1. Let (s, h) be a state such that $s(x) = 3, s(y) = 4$ and $h = \emptyset$, then the following table shows the information of each state about focused variables and addresses during the process of the following executions:

$$(s, h) \xrightarrow{x := \text{cons}(1)} (s_1, h_1) \xrightarrow{y := \text{cons}(1)} (s_2, h_2) \xrightarrow{[y] := 37} (s_3, h_3) \xrightarrow{y := [x+1]} (s_4, h_4) \xrightarrow{\text{disp}(x+1)} (s_5, h_5).$$

	Store		Heap
s	$x : 3, y : 4$	h	empty
s_1	$x : 37, y : 4$	h_1	$37 : 1$
s_2	$x : 37, y : 38$	h_2	$37 : 1, 38 : 1$
s_3	$x : 37, y : 38$	h_3	$37 : 1, 38 : 37$
s_4	$x : 37, y : 37$	h_4	$37 : 1, 38 : 37$
s_5	$x : 37, y : 37$	h_5	$37 : 1$

Let $\phi =_{df} (x \mapsto 1 * y \mapsto 1)$, $\psi =_{df} (x \mapsto 1 \wedge y \mapsto 1)$, we have $s_2, h_2 \models \phi$ and $s_2, h_2 \models \psi$, $s_5, h_5 \models \psi$, but $s_5, h_5 \not\models \phi$ since x and y point to the single memory storing value 1.

$\frac{\Gamma \Rightarrow (x := \mathbf{cons}(e), (s, h)) \longrightarrow (\downarrow, (s[x \mapsto n], h \cup \{(n, s(e))\})), \Delta}{\Gamma \Rightarrow (x := [e], (s, h)) \longrightarrow (\downarrow, (s[x \mapsto h(s(e))], h)), \Delta} \quad \text{where } n \text{ is new w.r.t. } h$	$\text{}^1 \text{ (cons)}$ $(x := [e])$
$\frac{\Gamma \Rightarrow ([e] := e', (s, h)) \longrightarrow (\downarrow, (s, h[s(e) \mapsto s(e')])), \Delta}{\Gamma \Rightarrow (\mathbf{disp}(e), (s, h)) \longrightarrow (\downarrow, (s, h _{\text{dom}(h) \setminus \{s(e)\}})), \Delta} \quad ([e] := e')$	(\mathbf{disp})

Table 8. Partial Rules of $(\mathbf{Pr}_{op})_{SP}$ for Program Transitions of Atomic Statements in Separation Logic

Encoding of Separation Logic in DLp. In DLp, let \mathbf{P}_{SP} and \mathbf{F}_{SP} be the set of programs and formulas of separation logic. In the PLK structure $\mathcal{K}_{SP} = (\mathcal{S}_{SP}, \longrightarrow_{SP}, \mathcal{I}_{SP})$ of separation logic, $\mathcal{S}_{SP} = \{(s, h) \mid s : V \rightarrow \mathbb{V}, h : \text{Addr} \rightarrow \mathbb{V}\}$, \mathcal{I}_{SP} interprets each formula of \mathbf{F}_{SP} as explained above. We directly choose the store-heap pairs as the configurations of separation logic named \mathbf{L}_{SP} . In this case, we simply let $\mathbf{M} =_{df} \{\tau\}$, where $\tau : \mathbf{L}_{SP} \rightarrow \mathcal{S}_{SP}$ is a constant mapping satisfying that $\tau((s, h)) =_{df} (s, h)$ for any $(s, h) \in \mathcal{S}_{SP}$. Table 8 lists the rules for the program transitions of the atomic statements in $(\mathbf{Pr}_{op})_{SP}$.

To further derive the formulas like $\phi * \psi$ in \mathbf{F}_{SP} into simpler forms, additional rules apart from Pr_{ldlp} need to be proposed. For example, we can propose a rule

$$\frac{\Gamma \Rightarrow h_1 \perp h_2, \Delta \quad \Gamma \Rightarrow (s, h_1) : \phi, \Delta \quad \Gamma \Rightarrow (s, h_2) : \psi, \Delta}{\Gamma \Rightarrow (s, h_1 \cup h_2) : \phi * \psi, \Delta} \quad (\sigma^*)$$

to decompose the heap $h_1 \cup h_2$ and the formula $\phi * \psi$, or a frame rule

$$\frac{\Gamma \Rightarrow (s, h) : \phi, \Delta}{\Gamma \Rightarrow (s, h) : \phi * \psi, \Delta} \quad (\sigma_{Frm}), \text{ if no variables of } \text{dom}(h) \text{ appear in } \psi,$$

to just decompose the formula $\phi * \psi$. These rules are inspired from their counterparts for programs in separation logic. In practice, the labels can be more explicit structures than the store-heap pairs shown here. Similar encoding can be obtained accordingly. From this example, we envision that the entire theory of separation logic can be embedded into DLp, where additional rules like the above ones support a “configuration-level reasoning” of separation-logic formulas.