A Clock-based Dynamic Logic for the Verification of CCSL Specifications in Synchronous Systems

Yuanrui Zhang[a,b,c], Hengyang Wu[c], Yixiang Chen[c], Frédéric Mallet[d,*]

[a] *School of Mathematics and Statistics, Southwest University*
[b] *RISE, College of Computer & Information Science, Southwest University*
[c] *MoE Engineering Research Center for Software/Hardware Co-design Technology and Application, East China Normal University, Shanghai 200062, China*
[d] *University Cote d'Azur, CNRS, Inria, I3S, 06900 Sophia Antipolis, France*

Abstract

The Clock Constraint Specification Language (CCSL) is a clock-based specification language for real-time embedded systems. With logical clocks defined as a first-class citizen, CCSL provides a natural way for describing clock constraints in synchronous systems — a special program model for real-time embedded systems. In this paper, we propose a clock-based dynamic logic called *CCSL Dynamic Logic* (CDL) for the verification of CCSL specifications in synchronous systems. It extends first-order dynamic logic with a synchronous execution mechanism in its program model and with CCSL primitives as terms in its logical formulae. We build a sound and relatively complete proof system for CDL to support the verification. Compared with previous approaches for verifying CCSL specifications, which are based on model checking and SMT checking techniques, our approach, which is based on theorem-proving, offers a unified verification framework in which both bounded and unbounded CCSL specifications can be verified. Technically, with the proof system of CDL, a complex CDL formula can be semi-automatically transformed into a set of quantifier-free, arithmetical first-order logic (QF-AFOL) formulae which can be checked by an SMT solver in an efficient way. As a case study, we analyze a simple synchronous system throughout the paper to illustrate how CDL works. We analyze and prove the soundness and completeness of the proof system for CDL. Currently, CDL is partially mechanized in Coq.

Keywords: Dynamic Logic, Clock Constraint Specification Language, Synchronous Systems, Verification, Theorem Proving
2010 MSC: 00-01, 99-00

1. Introduction

The Clock Constraint Specification Language [1, 2] (CCSL) is a formal declarative language for specification of real-time embedded systems. It was first defined as an annex of UML/MARTE (as indicated in [3], page 169) to supersede the UML profile for SPT [4], but later developed as an independent language equipped with a formal semantics [1]. In CCSL, *clock* is a primitive concept capturing a sequence of the occurrences of system events, while clock expressions capture the logical constraints between events. For example, "event c_1 always occurs before event c_2" can be captured as a clock expression $c_1 \prec c_2$ in CCSL (see Section 2.1). Clock constraints in CCSL are purely logical (i.e., irrelevant to explicit time values) and so are independent from the time model of a system. Therefore, CCSL can be widely applied to specify the properties of clocks of different types of system models, such as MARTE models [3, 5], timed automata [6, 7, 8] and synchronous system models [9, 10] (such as Signal [11] and Esterel [12]). Especially in

*Corresponding author
Email addresses: zhangyrmath@126.com (Yuanrui Zhang), Frederic.Mallet@inria.fr (Frédéric Mallet)

synchronous models, where the time model is discrete and each event is triggered at a lock step without time consumption, CCSL can serve as a clock calculus that can naturally capture the fundamental pure logical constraints between events. Some typical examples of using CCSL in different system models can be found in [3, 5, 6, 7, 8, 9, 10], including the one used in this paper (see Example 3.1).

Given a system model and a CCSL specification, the verification problem of the CCSL specification is to answer whether all behaviours of the system model satisfy the given CCSL specification [13]. Previous approaches to the verification of CCSL specifications are mainly based on model checking [6, 7, 9, 14, 15], where the main idea is to encode a system and a CCSL specification as finite transition systems, and make reachability analysis on their product. Despite being fully automatic, one major problem is that it can only verify CCSL specifications whose corresponding transition systems have a finite number of states (which we also call *bounded CCSL specifications* [16]). In CCSL, however, even very simple constraints can be unbounded, such as the expression $c_1 \prec c_2$ mentioned above. Recently, a promising solution for verifying unbounded CCSL specifications was proposed [17, 18], where a specification is encoded as a logical formula that can be checked by an SMT solver. But the problem with this method is that directly encoding CCSL as logical formulae would generate very complex formulae that contain quantifiers and functions (e.g., formula $\forall i \in \mathbb{N}^+.\mathcal{X}_\sigma(c_1, i) \geq \mathcal{X}_\sigma(c_2, i)$), which are undecidable or can be costly for SMT solvers, especially when using existential quantifiers and uninterpreted functions.

Compared with model checking and SMT solving, theorem proving is a different method. It is not based on state space exploration and it supports well the verification of specifications with infinite-state transition systems. In this method, logics like Hoare logic [19] and dynamic logic [20] provide the ability to describe program models, so that a system and its specification can be described in a single logical form, such as a Hoare triple in Hoare logic and a formula of the form $[p]\phi$ (where p is a program model, ϕ is a formula that describes a specification) in dynamic logic. The verification of the specification can then be realized by transforming the logical form into a verification condition (which consists of a set of pure first-order logic (FOL) formulae) according to the syntactical structure of the program model. In the verification of synchronous program models (such as Esterel programs [9]), another advantage of taking theorem-proving-based approaches is that in the derivation the structure of the program model can be kept. This is different from model-checking-based approaches where such information is lost since system models are expressed as finite transition systems. With program structures, modular verification is possible for large-scale systems.

In this paper, we focus on the verification of CCSL specifications in a special type of system models — synchronous system models — based on theorem proving. Specifically, we propose a variation of dynamic logic, called *CCSL Dynamic Logic* (CDL), for specifying and verifying CCSL specifications in a synchronous system model. We propose a proof system for CDL and prove that it is sound and relatively complete to arithmetical first-order logic (AFOL) in the sense of [21] (which is also called *relative completeness*). Currently, we have mechanized a part of CDL in Coq [22]. CDL consists of a program model and a set of logical formulae to capture both the dynamic synchronous system behaviour and the static CCSL specification in the same language. It allows verifying both bounded and unbounded CCSL specifications in a single verification framework.

Fig. 1 shows this verification framework. In contrast to the previous approaches [17, 18] which directly encode a CCSL specification as a relatively complex logical formula, CDL, as an intermediate language, captures the verification problem of the specification as a formula ϕ of the form: $I \wedge I_c \rightarrow [\|(p, q_1, ..., q_n)]\xi$. By verifying this formula, the verification problem that whether the synchronous system model satisfies the specification can be solved. In formula ϕ, the synchronous system is modeled as a program p in CDL, while the CCSL specification, which consists of a set of *clock definitions* and a set of *clock relations* (see Section 2.1), is modeled as programs $q_1, ..., q_n$ and a term ξ in CDL (with $q_1, ..., q_n$ modeling the clock definitions and ξ modeling the set of clock relations respectively); the operator $\|$ combines the programs $p, q_1, ..., q_n$ so that they can run concurrently; $I \wedge I_c$ is an initial condition of all variables in the programs $p, q_1, ..., q_n$. Formula of the form $[q]\xi$ means that all execution traces of program q satisfies term ξ. To verify the formula ϕ, we transform it into a set of pure FOL formulae (which in our case are quantifier-free arithmetical first-order logic (QF-AFOL) formulae) through the proof system of CDL in a modular way. These formulae contain no quantifiers and functions so that they can be checked by an SMT solver [23] in an efficient way compared to the previous approaches. The whole transformation procedure is semi-automatic,

and can be implemented by popular theorem provers such as Coq [22] and Isabelle [24].

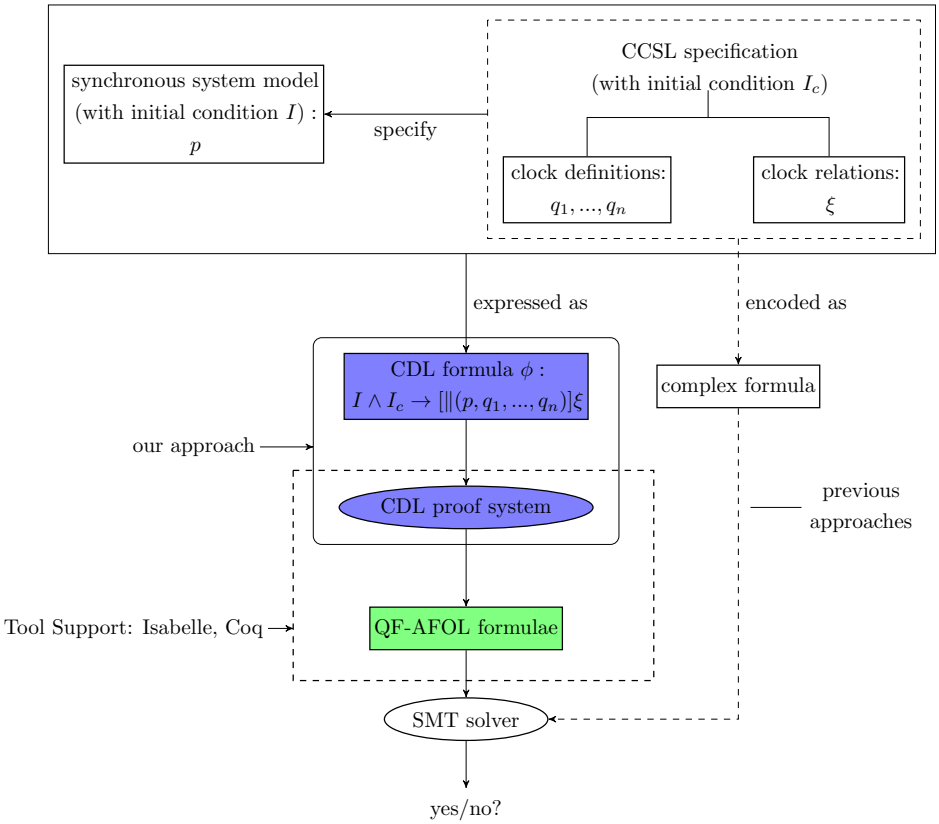

Figure 1: Verification framework of CDL

Fig. 2 shows an overview of CDL. It gives a picture of each part of the logic, its functionality and its relationship with CCSL and traditional *first-order dynamic logic* (FODL) [25]. The syntax of CDL is based on FODL. On one hand, the program model of CDL inherits from FODL the sequence operator ; and the choice operator $\cup$, and is extended with a synchronous execution mechanism that is supported by several new features: *signal*, *combinational event*, *infinite loop* and *concurrent executions* (see Section 3). On the other hand, the logical formulae of CDL inherit from FODL the formulae of the form $[p]\phi$ that capture the *state properties* of programs (see Section 3.1), and are extended with the formulae of the form $[p]\xi$ in order to express CCSL specifications — a type of *temporal properties* of programs. Clock relations *Rel* are embedded in the term ξ, while clock definitions can be encoded as CDL programs.

The proof system of CDL is mainly based on that of FODL, but enriched with rules for the new primitives introduced in CDL (see Table 3 in Section 4): the combinational events and the formulae of the form $[p]\xi$. To reason about parallel programs, we propose rewrite rules (see Table 6 in Section 4) together with an automatic procedure (Algorithms 1 - 5 in Section 4). These rules reduce a parallel program into a sequential one so that the rules for sequential programs can be applied. Such a way for verifying parallel programs is specially suitable for synchronous models because the behaviour of a synchronous model is always deterministic [12], which means that any synchronous parallel program can always be reduced to only one sequential program. The procedure for proving a CDL formula in the proof system of CDL is semi-automatic, since it is generally undecidable to find a loop invariant in a program that is complex enough to express the Presburger arithmetic theory [26]. The proof system is proved to be sound and relatively complete (see Section 6). Essentially, as other dynamic logics, the expressiveness of CDL stays the same as AFOL in the domain of natural numbers [20].

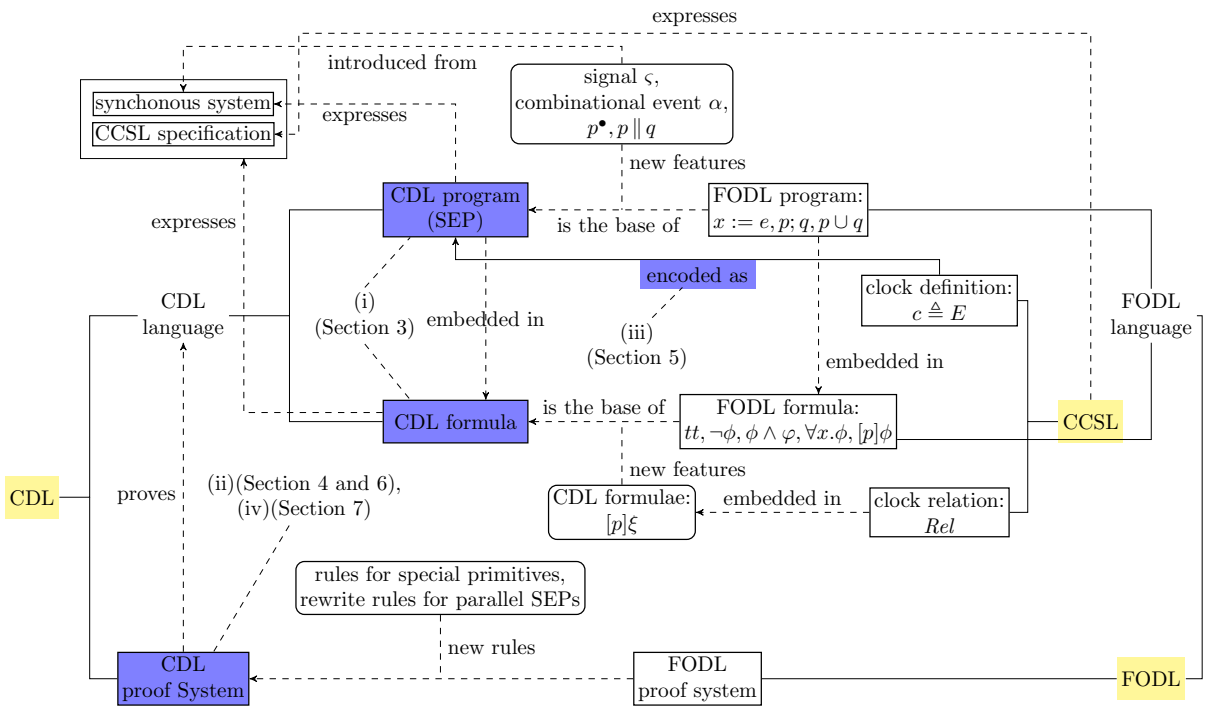

Figure 2: An overview of CDL

The work in this paper is mainly based on and extends [27], where a primary version of CDL was proposed. That primary CDL only supports verifying simple CCSL clock relations. When a specification contains clock definitions, it cannot be directly expressed in the primary CDL because it does not support parallel programs. We extend the primary CDL with a parallel operator and its related rewrite rules in the proof system. This extension is essential, considering the application domain, as well as non-trivial, as we show here, but could not be achieved in limited length without the foundations defined in [27]. In this paper we also give a full analysis of the soundness and completeness of CDL.

In summary, the main contributions of this paper are listed as follows (shown in Fig. 2):

(i) We define the syntax and semantics of CDL.

(ii) We propose a proof system for CDL, and analyze its soundness and completeness.

(iii) We show how to express CCSL specifications in CDL by proposing an encoding of the clock definitions of CCSL as the programs of CDL.

(iv) We mechanize a part of CDL, which consists of all formulae that contain the rules for *sequential SEPs* (i.e., all rules in Table 3, 4 and 5 in Section 4) but no parallel SEPs, in Coq.

The rest of this paper is organized as follows: Section 2 gives a general background to CCSL. Section 3 introduces the syntax and semantics of CDL. In Section 4, we propose a proof system for CDL. In Section 5, we show how to express a CCSL specification in CDL. Section 6 analyzes the soundness and completeness of CDL. In Section 7, we mechanize a part of CDL using Coq and illustrate how a CDL formula can be proved in Coq through a small example. Section 8 discusses related work. Section 9 concludes this paper and discusses future work.

2. The Clock Constraint Specification Language (CCSL)

2.1. Syntax and Semantics of CCSL

We present the syntax and semantics of CCSL based on [13, 17].

In synchronous systems, a CCSL clock captures a sequence of the occurrences of a signal. A *tick* of a clock indicates an occurrence of its corresponding signal. We often use c to denote a clock, and use $\mathcal{C}$ to denote a finite set of clocks. A *clock schedule* $\sigma : \mathbb{N} \to 2^{\mathcal{C}}$ defines one possible arrangement of all clocks over a discrete timeline (where $\mathbb{N} = \{0, 1, 2, ...\}$ is the set of natural numbers), where each clock either ticks or does not tick at each time step $i \in \mathbb{N}$ (also called *instant*). We define $c \in \sigma(i)$ iff clock c ticks at instant $i \in \mathbb{N}$. We assume $\sigma(0) = \emptyset$, indicating that no clock ticks at the beginning of the timeline. A configuration $\mathcal{X}_\sigma : \mathcal{C} \times \mathbb{N} \to \mathbb{N}$ keeps track of the number of ticks of all clocks up to the current instant $i \in \mathbb{N}$ of a schedule σ. It is defined as

$$\mathcal{X}_\sigma(c, i) =_{df} \begin{cases} 0, & \text{if } i = 0 \\ \mathcal{X}_\sigma(c, i-1) + 1, & \text{if } i > 0 \text{ and } c \in \sigma(i) \\ \mathcal{X}_\sigma(c, i-1), & \text{if } i > 0 \text{ and } c \notin \sigma(i) \end{cases}.$$

There are two types of clock expressions in CCSL: clock relations and clock definitions.

Clock relations describe binary relationships between clocks. The syntax of clock relations is defined as follows:

$$Rel ::= c_1 \subseteq c_2 \mid c_1 \# c_2 \mid c_1 \prec c_2 \mid c_1 \preceq c_2,$$

where c_1 and c_2 are clocks. The semantics of a clock relation is defined as the set of clock schedules satisfying it, shown as the items 1-4 in Table 1, where $\mathbb{N}^+ = \mathbb{N} - \{0\}$. *Subclock* says that c_1 can only tick if c_2 ticks; *Exclusion* says that c_1 and c_2 cannot tick at the same instant; *Precedence* means that c_1 always ticks faster than c_2; *Causality* expresses that c_1 ticks not slower than c_2.

For example, the leftmost figure of Fig. 3 shows a possible schedule σ (starting from the instant 1) that satisfies the clock relation $c_1 \prec c_2$, where

$$\sigma = \{\}\{c_1\}\{c_2\}\{c_1\}\{c_1, c_2\}\{\}\{\}\{c_1, c_2\}\{c_2\}\{\}\{c_1\}\{c_2\}\{\}....$$

We use '{}' to mean an empty set $\emptyset$. The configuration $\mathcal{X}_\sigma$ satisfies $\mathcal{X}_\sigma(c_1, 1) = 1$, $\mathcal{X}_\sigma(c_1, 2) = 1$, $\mathcal{X}_\sigma(c_1, 3) = 2$, $\mathcal{X}_\sigma(c_2, 1) = 0$ and $\mathcal{X}_\sigma(c_2, 2) = 1$.

1.	$\sigma \vDash_{ccsl} c_1 \subseteq c_2$	iff	$\forall i \in \mathbb{N}^+.c_1 \in \sigma(i) \to c_2 \in \sigma(i)$	(Subclock)
2.	$\sigma \vDash_{ccsl} c_1 \# c_2$	iff	$\forall i \in \mathbb{N}^+.c_1 \notin \sigma(i) \vee c_2 \notin \sigma(i)$	(Exclusion)
3.	$\sigma \vDash_{ccsl} c_1 \prec c_2$	iff	$\forall i \in \mathbb{N}^+.(\mathcal{X}_\sigma(c_1, i) > \mathcal{X}_\sigma(c_2, i) \vee (\mathcal{X}_\sigma(c_1, i) = \mathcal{X}_\sigma(c_2, i) \to c_1 \notin \sigma(i))$	(Precedence)
4.	$\sigma \vDash_{ccsl} c_1 \preceq c_2$	iff	$\forall i \in \mathbb{N}^+.\mathcal{X}_\sigma(c_1, i) \geq \mathcal{X}_\sigma(c_2, i)$	(Causality)
5.	$\sigma \vDash_{ccsl} c \triangleq c_1 + c_2$	iff	$\forall i \in \mathbb{N}^+.c \in \sigma(i) \leftrightarrow (c_1 \in \sigma(i) \vee c_2 \in \sigma(i))$	(Union)
6.	$\sigma \vDash_{ccsl} c \triangleq c_1 * c_2$	iff	$\forall i \in \mathbb{N}^+.c \in \sigma(i) \leftrightarrow (c_1 \in \sigma(i) \wedge c_2 \in \sigma(i))$	(Intersection)
7.	$\sigma \vDash_{ccsl} c \triangleq c_1 \blacktriangleright c_2$	iff	$\forall i \in \mathbb{N}^+.c \in \sigma(i) \leftrightarrow ($ $c_2 \in \sigma(i) \wedge$ $\exists j.(0 < j < i) \wedge (\forall k.(j \leq k < i) \to (c_1 \in \sigma(j) \wedge c_2 \notin \sigma(k)))$ $)$	(Strict Sample)
8.	$\sigma \vDash_{ccsl} c \triangleq c_1 \rhd c_2$	iff	$\forall i \in \mathbb{N}^+.c \in \sigma(i) \leftrightarrow ($ $c_2 \in \sigma(i) \wedge$ $\exists j.(0 < j \leq i) \wedge (\forall k.(j \leq k < i) \to (c_1 \in \sigma(j) \wedge c_2 \notin \sigma(k)))$ $)$	(Sample)
9.	$\sigma \vDash_{ccsl} c \triangleq c_1 \curvearrowright c_2$	iff	$\forall i \in \mathbb{N}^+.c \in \sigma(i) \leftrightarrow (c_1 \in \sigma(i) \wedge \forall j.(0 < j \leq i) \to c_2 \notin \sigma(j))$	(Interruption)
10.	$\sigma \vDash_{ccsl} c \triangleq c' \propto n$	iff	$\forall i \in \mathbb{N}^+.c \in \sigma(i) \leftrightarrow (c' \in \sigma(i) \wedge \exists m \in \mathbb{N}^+.\mathcal{X}_\sigma(c', i) = m \cdot (n+1))$	(Periodicity)
11.	$\sigma \vDash_{ccsl} c \triangleq c' \,\$\, n$	iff	$\forall i \in \mathbb{N}^+.\mathcal{X}_\sigma(c, i) = max(\mathcal{X}_\sigma(c', i) - n, 0)$	(Delay)
12.	$\sigma \vDash_{ccsl} c \triangleq c_1 \wedge c_2$	iff	$\forall i \in \mathbb{N}^+.\mathcal{X}_\sigma(c, i) = max(\mathcal{X}_\sigma(c_1, i), \mathcal{X}_\sigma(c_2, i))$	(Infimum)
13.	$\sigma \vDash_{ccsl} c \triangleq c_1 \vee c_2$	iff	$\forall i \in \mathbb{N}^+.\mathcal{X}_\sigma(c, i) = min(\mathcal{X}_\sigma(c_1, i), \mathcal{X}_\sigma(c_2, i))$	(Supremum)

Table 1: Semantics of CCSL

Clock definitions define new clocks from existing clocks by combining them using different clock expressions. They greatly enhance the expressiveness of CCSL. A clock definition has the form:

$$Cdf ::= c \triangleq E,$$

where E is defined by the following grammar:

$$E ::= c_1 + c_2 \mid c_1 * c_2 \mid c_1 \blacktriangleright c_2 \mid c_1 \triangleright c_2 \mid c_1 \curvearrowright c_2 \mid c \propto n \mid c \,\$\, n \mid c_1 \vee c_2 \mid c_1 \wedge c_2.$$

In the above definition, c_1 and c_2 are arbitary clocks, and $n \geq 1$. The semantics of a clock definition is defined as a set of schedules satisfying it, shown as the items 5-13 in Table 1. *Union* defines the clock that ticks iff either c_1 or c_2 ticks; *Intersection* defines the clock that ticks whenever both c_1 and c_2 tick; (resp. *Strict*) *Sample* defines the clock that (resp. strictly) samples c_1 based on c_2; *Interruption* defines the clock that ticks as c_1 until c_2 ticks; *Periodicity* defines the clock that ticks every n ticks of clock c'; *Delay* defines the clock that ticks when c' ticks but is delayed for n ticks of c'; *Infimum* (resp. *Supremum*) defines the slowest (resp. fastest) clock that is faster (resp. slower) than both c_1 and c_2.

With clock definitions one can define complex CCSL expressions like $c \prec c_2$ and $c \triangleq c_1 \,\$\, 2$. Fig. 3 shows a possible schedule that satisfies the clock definitions $c \triangleq c' \propto n$ and $c \triangleq c' \,\$\, n$ (when $n = 2$) respectively. They are used in the case study in Example 3.1.

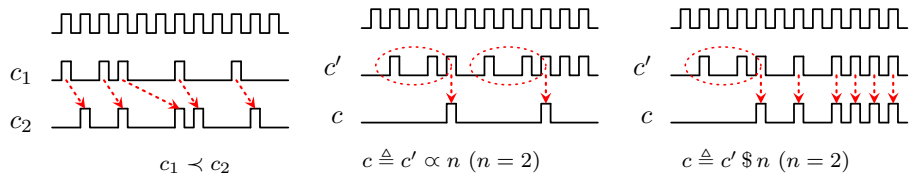

Figure 3: A possible schedule of selected clock expressions

2.2. CCSL Specifications

A CCSL specification is a set of clock relations and clock definitions, defined as a triple

$$Spec =_{df} \langle \mathcal{C}, \mathbf{Cdf}, \mathbf{Rel} \rangle,$$

where $\mathcal{C}$ is the set of all clocks appearing in the sets **Cdf** and **Rel**. **Cdf** is a set of clock definitions. **Rel** is a set of clock relations. A schedule σ satisfies a CCSL specification $\langle \mathcal{C}, \mathbf{Cdf}, \mathbf{Rel} \rangle$, denoted by $\sigma \models_{ccsl} \langle \mathcal{C}, \mathbf{Cdf}, \mathbf{Rel} \rangle$, if $\sigma \models_{ccsl} Rel$ and $\sigma \models_{ccsl} Cdf$ for each $Rel \in \mathbf{Rel}$ and $Cdf \in \mathbf{Cdf}$.

In CCSL, each specification can be captured as a special transition system (see [13]), whose traces are the schedules of clocks in the specification. For example, the clock relation $c_1 \subseteq c_2$ corresponds to a transition system shown in Fig. 4(i). A CCSL specification is called *bounded* if its corresponding transition system has a finite number of states [16], otherwise it is called *unbounded*. For unbounded specifications, the reason that causes the unbounded size of the state space is that in some clock expressions, the difference in the numbers of ticks between some clocks is not bounded. For example, in the transition system of clock relation $c_1 \prec c_2$ shown in Fig. 4(ii), the difference (δ) in the number of ticks between the clock c_1 and c_2 is tagged at each state and it is increasing infinitely.

3. The Syntax and Semantics of CDL

In this section we define the CCSL Dynamic Logic (CDL). CDL enriches the program model of FODL with a synchronous execution mechanism, and enriches the logical formulae of FODL with CCSL primitives. As a background, we first give a brief introduction to FODL in Section 3.1. In Section 3.2 we define the syntax of the program model and the logical formulae of CDL. Then in Section 3.3, we define the semantics for the program model and the logical formulae of CDL.

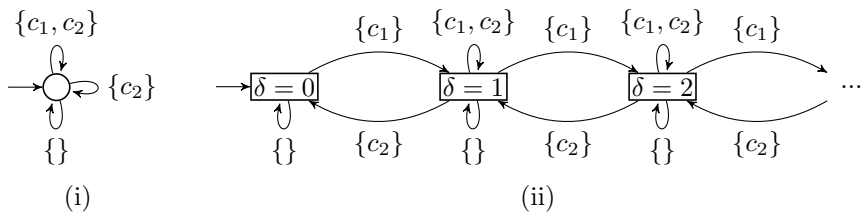

Figure 4: Transition systems of $c_1 \subseteq c_2$ and $c_1 \preceq c_2$

3.1. First-Order Dynamic Logic

First-order dynamic logic (FODL) is a variation of modal logic for modeling and verifying program specifications. It is an extension of propositional dynamic logic with an assignment $x := e$ and a test $P?$ in its program model. The FODL we present here is based on [20].

The language of FODL consists of two parts: a program model and a set of logical formulae.

The program model of FODL is a *regular language*, whose syntax is defined as follows:

$$p ::= x := e \mid P? \mid p;p \mid p \cup p \mid p^*,$$

where e is a term of a type of expressions, P is a proposition of a logic. The syntax of e and P depends on the discussed domain(e.g., e could be an arithmetic expression and P could be a quantifier-free first-order logical formula defined in Def. 3.1). The test $P?$ means that at the current state, P is true. $p;q$ means that the program first executes p and after p terminates, it executes q. $p \cup q$ means that the program either executes p, or executes q, it is a non-deterministic choice. p^* means that the program executes p a finite number of times.

The logical formulae of FODL is defined as follows:

$$\phi ::= tt \mid a \mid [p]\phi \mid \neg\phi \mid \phi \wedge \phi \mid \forall x.\phi,$$

where tt is Boolean true, and a is an atomic formula whose definition depends on the discussed domain (e.g., it could be θ defined in Def. 3.4). $[p]\phi$ is a formula that captures a state property (i.e., a property held on a state) of the program. It means that after all executions of the program p, the formula ϕ holds.

The semantics of FODL is based on Kripke frames [20]. A Kripke frame is a pair (S, val) where S is a set of states, val is an interpretation function whose definition depends on the type of logic being discussed. In FODL, val interprets a program as a set of pairs (s, s') of states and interprets a formula as a set of states. Intuitively, each pair $(s, s') \in val(p)$ means that starting from the state s, after the execution of p, the program terminates at the state s'. For each state s, $s \in val([p]\phi)$ means that for all pairs $(s, s') \in val(p)$, s' satisfies ϕ. For a formal definition of the semantics of FODL, one can refer to [20].

The proof system of FODL is sound and relatively complete. Except for the rules for an assignment $x := e$ and a test $P?$, the rules for other operators and logical connectives are also defined as a part of the proof system of CDL (which consists of all rules in Table 4 and 5 in Section 4)[1]. We refer to [20] for more details.

3.2. The Syntax of CDL

Like FODL, CDL consists of two parts: a program model and a set of logical formulae. We first define the program model in Section 3.2.1 and then define the logical formulae in Section 3.2.3.

[1]where the rules for operator $*$ can be obtained by replacing each '$\bullet$' with '$*$' in the rules for operator $\bullet$ in Table 4.

3.2.1. Syntax of Synchronous Event Programs

To begin defining the syntax of CDL we first define the program model of CDL, called a *synchronous event program* (SEP), based on the program model of FODL.

As indicated in Fig. 2, in order to describe the behaviour of synchronous systems and CCSL specifications in dynamic logic, we introduce the synchronous execution mechanism in the program model of FODL. To this end we introduce the notion of signal as a primitive event to model the synchronous communication between system modules, as what has been done in synchronous programming languages like Signal [11] and Esterel [12]. We introduce the notion of combinational event to express a simultaneous execution of several events. With this notion, SEP is able to support a synchronous semantics: the time model is discrete and at each instant, several events can occur simultaneously. We introduce the parallel operator '$\|$' to describe the concurrent execution in synchronous systems.

In synchronous systems, infinite loop is a main feature. In the infinite loop (e.g., the *loop ... end* statement of Esterel [12]), the program continuously proceeds until a deadlock happens or an interruption point is reached. In SEP, we propose a new loop operator '$\bullet$' to express the infinite loop in synchronous systems, based on the finite loop '$*$' in the program model of FODL. Informally, a loop program $p^\bullet$ expresses that the program p either proceeds for any finite number of times, or proceeds for infinitely many times. The deadlock and interruption in the infinite loop can be realized by using the operator $\bullet$ and the test operator (see Def. 3.1) in SEP.

The syntax of SEPs is given as the following definition.

Definition 3.1 (Syntax of SEPs)**.** *The syntax of SEPs is defined as follows:*

$$p ::= \mu \mid \alpha \mid \varrho \& P ? \alpha \mid p; p \mid p \cup p \mid p^\bullet \mid \|(p_1, ..., p_n),$$

where α is defined as:

$$\alpha ::= \epsilon \mid b,$$
$$b ::= \varsigma ! e \mid x := e' \mid (b|b).$$

e, P and ϱ are defined as:

$$e ::= x \mid n \mid e + e \mid e - e \mid n \cdot e \mid e/n,$$
$$P ::= tt \mid e \leq e \mid \neg P \mid P \wedge P,$$
$$\varrho ::= \hat{\varsigma}(x) \mid \bar{\varsigma} \mid \varrho \,\overline{\wedge}\, \varrho \mid \varrho \,\underline{\vee}\, \varrho.$$

Note: ϱ only appears in a parallel program of the form $\|(p_1, ..., p_n)$.

μ, α and $\varrho \& P ? \alpha$ are atomic programs. Their intuitive meanings are given as follows:

- A *skip* program μ represents a program that does nothing and consumes no time.
- A combinational event α can be an *idle event* ϵ, or consists of several signals of the form $\varsigma ! e$ or *local assignments* of the form $x := e$ linked by an operator '$|$' (e.g., $\alpha = (\varsigma ! 5 | x := 5)$), meaning that they occur simultaneously. Signal and local assignment are indivisible (atomic) events. There is no execution order between the atomic events of a combinational event so the order in which they are linked by $|$ is irrelevant. For example, the behaviour of $(\varsigma ! 5 | x := 5)$ is the same as that of $(x := 5 | \varsigma ! 5)$. All atomic events in a combinational event occur simultaneously and consume one unit of time. An idle event ϵ does nothing, but consumes a unit of time.
- A signal $\varsigma ! e$ is a pair made of a signal name ς and a value expressed as an expression e. It means that the signal ς emits with the value of an expression e. When the value e of the signal can be ignored, we denote the signal simply as 'ς'. We also call a signal ς without a value a *pure signal*. The local assignment $x := e$ means to assign the value of an expression e to a variable x. x is called a *general variable*, with the domain $\mathbb{Z}$. We use *Var* to denote a set of general variables. e is a Presburger arithmetic expression, which consists of a variable x, an integer number $n \in \mathbb{Z}$, and arithmetic expressions connected by addition $_ + _$, subtraction $_ - _$, scalar multiplication $n \cdot _$ and scalar division $_/n$ $(n \neq 0)$.

- $\varrho \& P ? \alpha$ is a *test event*. It means that if the conditions ϱ and P are true, the event α proceeds, otherwise a deadlock occurs (i.e., the program halts without terminating). If there is no ϱ or P, we simply write it as $\varrho ? \alpha$ or $P ? \alpha$. The procedure for checking ϱ and P is assumed to consume no time. The condition P is a proposition expressed as a QF-AFOL formula, where *tt* represents Boolean true (*ff* represents Boolean false), $\leq$ represents the less-than relation between two integers. The *signal test condition* ϱ is a logical expression, it is only allowed when program is running concurrently with other programs (see below the intuitive explanation of the parallel operator '$\|$'). $\hat{\varsigma}(x)$ and $\bar{\varsigma}$ express the signal states (*present* or *absent*) at the current instant: $\hat{\varsigma}(x)$ means that the signal ς is present at the current instant, where the variable x is for storing the value of the signal ς; $\bar{\varsigma}$ means that "the signal ς is absent at the current instant". $\veebar$ and $\barwedge$ represent the logical *or* and *and* respectively.

For convenience, sometimes we abuse the term *event* to also denote a combinational event or a test event. They should not be confused in the context.

In SEP, we use a special symbol $\ddagger$ to represent an idle event with a false condition:

$$\ddagger =_{df} ff ? \epsilon.$$

This program always reaches a deadlock because its test condition will never be satisfied. It is often used to express the infinite loop (introduced below) program in the form $p^{\bullet}; \ddagger$ in SEP (see Example 3.1). The semantics of $\ddagger$ corresponds to the empty set (see Def. 3.9 below). We call a program, e.g. $\epsilon; \ddagger$, which always reaches a deadlock, a *miracle program*.

Compositional programs are composed by atomic programs using operators ;, $\cup$, $\bullet$ and $\|$. Their intuitive meanings are given as follows:

- Operators ; and $\cup$ correspond to the sequential and nondeterministic choice operators in the traditional FODL (see Section 3.1) respectively.

- $\bullet$ is the loop operator. $p^{\bullet}$ means that the program p nondeterministically executes for some finite number of times or infinitely many times.

- $\|$ is the n-tuple parallel operator. $\|(p_1, ..., p_n)$ means that the programs $p_1, ..., p_n$ are running concurrently. At each instant, the events in $p_1, ..., p_n$ are triggered simultaneously and all conditions ϱ are checked according to the signal state at the current instant. Signals are broadcasting their values, while each program can receive them at the same instant.

We call an SEP that is not in the form $p \| q$ a *sequential SEP*.

In SEP, we stipulate that signals are the only way for communication between programs that are running concurrently. So all assignments in SEP must be local. Formally, given a parallel program $\|(p_1, ..., p_n)$, if an assignment $x := e$ appears in one of the programs $p_1, ..., p_n$, e.g., say p_1, then the variable x must not appear in the other programs $p_2, ..., p_n$.

The precedence of operators are listed as follows from the highest to the lowest: $\bullet$, ;, $\cup$, $\|$. We stipulate that ; is right-associative, and $\cup$ is left-associative. For example, program $\alpha_1 \cup p_1; p_2; p_3^{\bullet} \cup P_1 ? \alpha_2^{\bullet} \cup P_2 ? \alpha_3$ means $(((\alpha_1 \cup p_1; (p_2; p_3^{\bullet})) \cup P_1 ? \alpha_2^{\bullet}) \cup P_2 ? \alpha_3)$.

Example 3.1 (Digital Filter System)**.** *We consider a simple synchronous system — a digital filter (DF) system — throughout this paper. The DF system we analyze here is based on [9]. As Fig. 5 shows, the main function of the DF system is to read image pixels from a memory, filter them in some way, and send the result out to a video device. The explicit structure of the DF system is shown in the right figure of Fig. 5, where we consider a simplified behaviour between two modules: a feeder and a filter. They interact with each other and with their environment through ports e, p, r and o. Ports are the only way for different modules to communicate in synchronous models. They can be modeled as signals ς_e, ς_p, ς_r and ς_o respectively in SEP. The behaviour of the DF system can be described in a loop where the behaviours of the Filter and the Feeder are described separately as follows:*

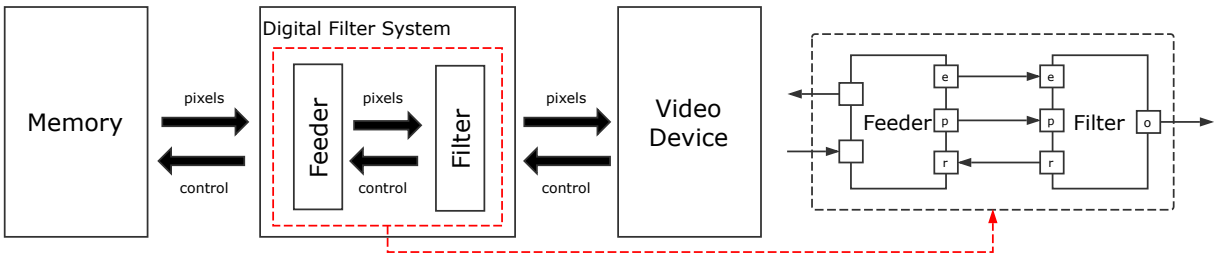

Figure 5: The Digital Filter system

(i) The filter sends a "Ready" message through the port r, telling the feeder that "I am ready for the pixels". From the next instant it continuously receives pixels from the port p, one pixel per unit of time. After receiving 2 pixels, the filter waits for an "Ending" message from the port e, then it processes the pixels (taking no time) and sends the result to the environment through the port o.

(ii) On the other hand, the feeder receives the "Ready" message from the port r at the same time when the filter sends it. From the next instant it starts an inner loop to send pixels through the port p, one pixel per unit of time. After sending 2 pixels, it sends an "Ending" message to inform the filter through the port e.

The behaviour of the filter and the feeder, and the behaviour of the DF system as a whole can be described as SEPs as follows:

$$\begin{aligned} Fil &=_{df} (\varsigma_r; \hat{\varsigma_p}?\epsilon; \hat{\varsigma_p}?\epsilon; \hat{\varsigma_e}?\varsigma_o)^\bullet; \ddagger, \\ Fee &=_{df} (\hat{\varsigma_r}?\alpha_1; (P_1?\alpha_2 \cup P_2?\alpha_3)^\bullet; f = 1?\varsigma_e)^\bullet; \ddagger, \\ DF &=_{df} \|(Fil, Fee), \end{aligned}$$

where $P_1 = (x = 2 \wedge f = 0)$, $P_2 = (x < 2 \wedge f = 0)$, $\alpha_1 = (x := 1 | f := 0)$, $\alpha_2 = (\varsigma_p | f := 1)$, $\alpha_3 = (\varsigma_p | x := x + 1)$. Generally, a program $p^\bullet; \ddagger$ describes that p is executed infinitely many times, just like the command "while true do p" in common imperative programs. The program p can not successfully terminate after executing for any finite number of times since the test event $\ddagger$, as defined above, causes a deadlock. The variable f is a flag indicating the end of the loop $(P_1?\alpha_2 \cup P_2?\alpha_3)^\bullet$.

The program Fil and Fee are running concurrently in DF. For example, at the first instant, ς_r is executed, since the condition $\hat{\varsigma_r}$ is satisfied, the event α_1 is also executed. So at the first instant, the combinational event $(\varsigma_r | x := 1 | f := 0)$ is executed.

3.2.2. SEPs vs. Imperative Synchronous Languages

As a synchronous version of the program model of FODL, SEP inherits some core features of synchronous languages from Esterel [12] (like signal and synchronous communication and execution) but mainly follows the style of regular languages for constructing programs. This makes SEP a convenient language both for synchronous modeling and compositional verification. Compared with imperative synchronous languages like Esterel, SEP is designed to be a lower-level language that is closer to the form for derivation in dynamic logic. SEP can encode the core behaviour of synchronous languages.

To show this, as an example, Table 2 gives an encoding of some statements from the Esterel language in [28] as SEPs. The full encoding would be complex and is outside of the scope of this paper. One of the main differences between Esterel and SEP is that in SEP every combinational event consumes a unit of time, while in Esterel only the statement *pause* consumes time. Thus the statement *pause* means more than the idle event ϵ in SEP. It also indicates the end of a combinational event and splits two events that run at different instants. In SEP, the sequential operator ; means the sequential execution of events at different instants, while in Esterel the operator ; is just a symbol for linking different statements.

Esterel Statement	Corresponding SEP Statement
nothing	μ
emit $\varsigma(e)$	$\varsigma!e$
$x := e$	$x := e$
pause	ϵ
emit ς_1; *emit* ς_2	$(\varsigma_1 \vert \varsigma_2)$
emit ς_1; *emit* ς_2; *pause*; $x := e$	$(\varsigma_1 \vert \varsigma_2); x := e$
present ς *then emit* ς_1 *else* $x := e$ *end*	$\hat{\varsigma}?\varsigma_1 \cup \bar{\varsigma}?x := e$
if P *then emit* ς_1 *else* $x := e$ *end*	$P?\varsigma_1 \cup \neg P?x := e$
loop (*emit* ς_1; *pause*) *end*	$(\varsigma_1)^{\bullet}; \ddagger$
suspend (*emit* ς_1; *pause*; *emit* ς_2) *when* ς	$\varsigma_1; (\hat{\varsigma}?\epsilon)^{\bullet}; \bar{\varsigma}?\varsigma_2$
(*emit* ς_1; *emit* ς_2) $\Vert$(*emit* ς_3; *pause*; *emit* ς_4)	$(\varsigma_1 \vert \varsigma_2) \Vert (\varsigma_3; \varsigma_4)$
trap T *in* {*present* ς *then exit* T *else emit* ς_1 *end*}	$\hat{\varsigma}?\epsilon \cup \bar{\varsigma}?\varsigma_1$

Table 2: An example of the encoding of Esterel programs as SEPs

3.2.3. Syntax of CDL Formulae

As indicated in Fig. 1 and 2, CDL supports describing CCSL specifications of synchronous systems by embedding clock relations as special terms into the logical formulae of FODL. Clock definitions are actually not a syntactical part of CDL. They can be encoded as an observer SEP and combined with the SEP of a system in parallel. This encoding is introduced in Section 5.

In synchronous system models, a CCSL clock indicates the occurrences of a signal. So in CDL we need to build a connection between signals and clocks, as stated in the following definition.

Definition 3.2 (Connection between Signals and Clocks)**.** *Given a set of pure signals Sig and a clock set* $\mathcal{C}$*,*

$$SigM : Sig \to \mathcal{C}$$

is a bijection that associates a clock to each signal. For each signal $\varsigma \in Sig$*, clock* $SigM(\varsigma)$ *models its occurrences.*

For each signal ς in SEPs, there is a clock $c = SigM(\varsigma)$. The clock c models a sequence of the occurrences of the signal ς: In a schedule, each tick of c corresponds to a present state of ς, and each absence of c corresponds to an absence of ς.

Given a clock set $\mathcal{C}$, we often denote the corresponding signal (if it exists) of a CCSL clock $c \in \mathcal{C}$ as ς^c. So there are $c = SigM(\varsigma^c)$ and $\varsigma^c = SigM^{-1}(c)$.

Example 3.2. *In Example 3.1, we define clocks* c_e, c_p, c_r *and* c_o *to model the sequences of the occurrences of the signals* ς_e, ς_p*,* ς_r *and* ς_o *respectively. Fig. 6 shows the schedule of the system DF for each clock.*

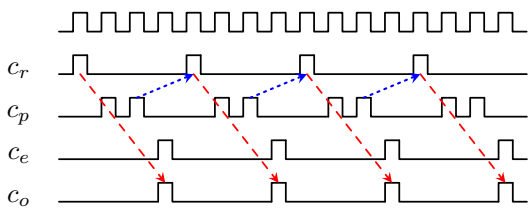

Figure 6: The schedule of system DF

In order to keep the schedule σ and the configuration $\mathcal{X}_\sigma$ for each clock at the current instant, in CDL we introduce a type of variables called *clock-related variables*. They play a crucial role for reasoning about clock relations in CDL.

Definition 3.3 (Clock-related Variables)**.** *For each clock c, we associate two* clock-related variables

$$c^n \text{ and } c^s$$

to it. The variable c^n has type $\mathbb{N}$, it records the number of times the clock has ticked up to the current instant. The variable c^s has type $\{0,1\}$, it records the status of the clock (1 for "ticked" and 0 for "not ticked") at the current instant.

The connection between some clock-related variables c^n and c^s in the Kripke frame of CDL and the schedule and configuration of the clock c in the sematics of CCSL is stated in Prop. 3.3 below.

Given a clock set $\mathcal{C}$, we denote the set of variables related to $\mathcal{C}$ as $RVar(\mathcal{C})$.

CDL formulae extend FODL formulae with clock relations as special terms in order to express the CCSL specifications of SEPs. The syntax of CDL formulae is given in the following definition.

Definition 3.4 (Syntax of CDL Formulae)**.** *A CDL formula ϕ is defined as follows:*

$$\phi ::= tt \mid \theta \mid [p]\xi \mid [p]\phi \mid \neg\phi \mid \phi \wedge \phi \mid \forall x.\phi$$

where

$$\begin{aligned} \xi &::= Rel \mid \curlywedge(Rel_1, ..., Rel_n), \\ \theta &::= E \leq E, \\ E &::= x \mid c^n \mid c^s \mid n \mid E + E \mid E \cdot E \end{aligned}$$

tt, θ, $[p]\xi$ and $[p]\phi$ are atomic CDL formulae. E is an integer arithmetic expression. Different from e, it contains clock-related variables c^n and c^s (see Def. 3.3) and the multiplication between expressions $(E \cdot E)$. $x \in Var$ is a general variable. p is an SEP. Different from $[p]\phi$ in FODL, formula $[p]\xi$ captures a temporal property related to a *CCSL term* ξ. It means that all execution paths of program p satisfies ξ.

In term ξ, $\curlywedge(Rel_1, ..., Rel_n)$ represents the conjunction of the clock relations $Rel_1, ..., Rel_n$. The semantics of $\curlywedge(Rel_1, ..., Rel_n)$ is defined as:

$$\models_{ccsl} \curlywedge(Rel_1, ..., Rel_n) \quad \text{iff} \quad \sigma \models_{ccsl} Rel_1, ..., \sigma \models_{ccsl} Rel_n.$$

To make it more convenient to express the negation of formula $[p]Rel$ in CDL, we also import the negation $\sim$ and the disjunction $\curlyvee$ of clock relations. Their semantics is defined as:

(i) $\sigma \models_{ccsl} \sim cr \quad \text{iff} \quad \sigma \not\models_{ccsl} cr$,

(ii) $\sigma \models_{ccsl} \curlyvee(cr_1, ..., cr_n) \quad \text{iff} \quad \sigma \models_{ccsl} \sim\curlywedge(\sim cr_1, ..., \sim cr_n)$,

where $cr, cr_i \in \{Rel_i, \sim Rel_i\}$ $(1 \leq i \leq n)$.

In this paper, we often call ξ or $\sim\xi$ a *path formula*, denoted by π. Other arithmetic expressions, relations and logical expressions, such as $E - E$, E/E, $E = E$, $E < E$, ff, $\langle p\rangle \sim\xi$, $\langle p\rangle\phi$, $\phi \vee \phi$, $\phi \rightarrow \phi$, $\exists x.\phi$, etc, can be expressed using the formulae given above. For example, $\langle p\rangle \sim\xi$ can be expressed as $\neg[p]\xi$, which means that there exists an execution path of p that satisfies $\sim\xi$; $E_1 - E_2$ and E_1/E_2 can be expressed as $\exists x.(E_2 + x = E_1)$ and $\exists x.(x \cdot E_2 = E_1)$ respectively.

Example 3.3. *Consider two CDL formulae*

$$\begin{aligned} \phi_1 &= \varphi_1 \rightarrow [p_1]c_r \prec c_o, \\ \phi_2 &= \varphi_2 \rightarrow [p_2]x > z, \end{aligned}$$

where $\varphi_1 = (c_r^n = 0 \wedge c_r^s = 0 \wedge y = 0)$, $p_1 = DF$ (where DF is given in Example 3.1), $\varphi_2 = (x = 1 \wedge z = 2 \wedge \exists z.x = z)$, $p_2 = (x := z + 1|\varsigma|y := 1); x := y + 1$. ϕ_1 means that if φ_1 is true, then all execution paths of p_1 satisfies the clock relation $c_r \prec c_o$. ϕ_2 means that if φ_2 holds, then after all executions of p_1, the formula $x > z$ holds.

Given a program p (resp. formula ϕ), we use $\mathcal{C}(p)$ (resp. $\mathcal{C}(\phi)$) to represent the set of all clocks appearing in p (resp. ϕ) and all clocks whose corresponding signals appear in p (resp. ϕ). For example, in Example 3.3, $\mathcal{C}(\phi_1) = \{c_e, c_p, c_r, c_o\}$.

Like in FODL [20], we introduce the notion of *dynamic/static variable* in CDL. Intuitively, a variable whose value might be changed along the execution of a program is called a *dynamic variable*.

Definition 3.5 (Dynamic/Static Variables)**.** *In CDL, given a program p (resp. formula ϕ), a variable X is dynamic w.r.t. p (resp. ϕ) if*

(1) $X \in Var$, and it appears on the left side of an assignment of the form $X := e$, or

(2) X is a clock-related variable c^n and there exists a signal ς^c in p (resp. ϕ), or

(3) X is a clock-related variable c^s.

A variable is static w.r.t. p (resp. ϕ) if it is not dynamic w.r.t. p (resp. ϕ).

For any general variable X, its value is set to the value of the expression e after the execution of an assignment $X := e$. According to Def. 3.3, since a variable c^n records the number of times the clock c has ticked up to the current instant, its value is increased by 1 if the signal ς^c occurs, and is kept unchanged otherwise. Since a variable c^s records the status of the clock c, its value is set to 1 if the signal ς^c occurs, and is set to 0 otherwise.

We use $\mathcal{V}(p)$ (resp. $\mathcal{V}(\phi)$) to represent the set of the dynamic variables in p (resp. ϕ), $Var(p)$ (resp. $Var(\phi)$) to represent the set of the variables in p (resp. ϕ).

Example 3.4. *In the formula*

$$\phi_1 = (c_r^n = 0 \wedge c_r^s = 0 \wedge y = 0) \rightarrow [DF]c_r \prec c_o$$

of Example 3.3, $\mathcal{V}(\phi_1) = \{c_e^n, c_e^s, c_p^n, c_p^s, c_r^n, c_r^s, c_o^n, c_o^s, x, f\}$. The set of the static variables in ϕ_1 is: $Var(\phi_1) - \mathcal{V}(\phi_1) = \{y\}$.

Like in FODL [20], we introduce the notion of *bound/free variable* in CDL. This notion is important when we discuss substitutions in CDL.

Definition 3.6 (Bound/Free Variables)**.** *In CDL, given a variable X, we say that X is in the scope of the effect of A, where $A \in \{X := e, \hat{\varsigma}(X)\}$ or $A = \varsigma^c!e$ (when $X \in \{c^n, c^s\}$), if:*

(1) there exists a formula of the form $[q]\varphi$ such that X is in φ, and A appears in q, or

(2) there exists a program of the form $q; r$ such that X is in r, A appears in q.

Given a program p (resp. formula ϕ), a variable X is bound w.r.t. p (resp. ϕ) if it is in the scope of the effect of some quantifier $\forall X$ or $\exists X$, some assignment $X := e$, some signal $\varsigma^c!e$ (if $X \in \{c^n, c^s\}$) or some signal test condition $\hat{\varsigma}(X)$.

Example 3.5. *In the formula*

$$\phi_2 = (x = 1 \wedge z = 2 \wedge \exists z.x = z) \rightarrow [(x := z + 1 | \varsigma | y := 1); x := y + 1]x > z$$

of Example 3.3, the first and second occurrences of the variable x are free, while the third occurrence of x (in "$x > z$") is bound by the assignment $x := y + 1$. The first occurrence of the variable z is free, the second occurrence of z is bound by the quantifier $\exists z$, and the third and fourth occurrences of z are free. The only occurrence of the variable y (in "$x := y + 1$") is bound by the assignment $y := 1$. Consider the formula

$$\phi_3 = [\|(\epsilon; \varsigma!(x + 1), y := z + 1; \hat{\varsigma}(z); x := z + 1)]x > z,$$

the first occurrence of the variable z is free, while the second and third occurrences of z are bound by the condition $\hat{\varsigma}(z)$.

With the concept of bound and free variables we introduce the notion of substitution in CDL. When doing a substitution on a formula, we need to make sure that the meaning of the formula does not change. So we introduce the notion of *admissible substitution.*

Definition 3.7 (Substitution/Admissible Substitution)**.** *Given a CDL formula ϕ and a variable $X \in Var \cup RVar(\mathcal{C})$, a substitution $\phi[E/X]$ replaces every free occurrence of X in p with the expression E.*

A substitution $\phi[E/X]$ is admissible if there exists no variable Y such that

(1) Y is in E, and

(2) Y is bound in $\phi[E/X]$.

Given a multi-set Γ of formulae, we write $\Gamma[E/X]$ to represent the multi-set obtained by doing the substitution $\phi[E/X]$ for each formula ϕ in Γ. Given two vectors $\vec{u} = (E_1, ..., E_n), \vec{v} = (X_1, ..., X_n)$, we write $\phi[\vec{u}/\vec{v}]$ as a shorthand for $\phi[E_1/X_1][E_2/X_2]...[E_n/X_n]$.

Example 3.6. *In the formula*

$$\phi_2 = (x = 1 \wedge z = 2 \wedge \exists z.x = z) \rightarrow [(x := z + 1|\varsigma|y := 1); x := y + 1]x > z$$

of Example 3.3, let $E_1 = z + 1$, $E_2 = x + 1$, and v be a variable different from x and y, then $\phi_2[v/z]$ is admissible, and

$$\phi_2[v/z] = ((x = 1 \wedge v = 2 \wedge \exists z.x = z) \rightarrow [(x := v + 1|\varsigma|y := 1); x := y + 1]x > v).$$

$\phi_2[E_1/z]$ is admissible, and

$$\phi_2[E_1/z] = ((x = 1 \wedge z + 1 = 2 \wedge \exists z.x = z) \rightarrow [(x := (z + 1) + 1|\varsigma|y := 1); x := y + 1]x > z + 1).$$

$\phi_2[E_2/z]$ is not admissible, and

$$\phi_2[e_2/z] = ((x = 1 \wedge x + 1 = 2 \wedge \exists z.x = z) \rightarrow [(x := (x + 1) + 1|\varsigma|y := 1); x := y + 1]x > x + 1).$$

$\phi_2[E_2/z]$ is not admissible because the variable z in "$x > z$" is a free occurrence in ϕ_2, but after the substitution, x in "$x > x + 1$" is bound.

Unless specially mentioned, all substitutions discussed in this paper are admissible substitutions.

3.3. Semantics of CDL

In the Kripke frame (S, val) (see also Section 3.1) of CDL, S is a set of states, $val : (\mathbf{Prog} \uplus \mathbf{Fmla}) \rightarrow ((S^* \cup S^\omega) \uplus 2^S)$ interprets a program as a set of traces on S and a logical formula as a set of states (where **Prog** denotes the set of all SEPs and **Fmla** denotes the set of all CDL formulae, S^* represents the set of all finite traces and S^ω represents the set of all infinite traces on S). A trace $tr \in (S^* \cup S^\omega)$ is a finite or infinite sequence of states. Given a finite trace $tr_1 = s_1 s_2 ... s_n$ and a (possibly infinite) trace $tr_2 = u_1 u_2 ... u_n ...$, concatenation $tr_1 \rtimes tr_2$ is a partial function defined as:

$$tr_1 \rtimes tr_2 =_{df} s_1 s_2 ... s_n u_2 u_3 ... \text{ if } s_n = u_1.$$

Given any tr_1, tr_2, we define

$$tr_1 \circ tr_2 =_{df} \begin{cases} tr_1 \rtimes tr_2, & \text{if } tr_1 \text{ is finite} \\ tr_1, & \text{otherwise} \end{cases}.$$

Given two sets of traces S_1 and S_2, $S_1 \circ S_2$ is defined as

$$S_1 \circ S_2 =_{df} \{tr_1 \circ tr_2 \mid tr_1 \in S_1, tr_2 \in S_2\}.$$

We use $tr(i)$ to denote the ith $(i \geq 0)$ element of trace tr, use tr_b to denote the first element of trace tr, $tr_b = tr(0)$, and use tr_e to denote the last element of trace tr provided that tr is a finite trace.

In CDL, we interpret arithmetic operators $+, -, \cdot, /$, relation $\leq$, and CCSL clock relations $\subseteq, \prec, \preceq, \#$ as their usual meanings. Next we define the concepts of *state* and *evaluation* in CDL.

Definition 3.8 (State and Evaluation in CDL)**.** *A state* $s : (RVar(\mathcal{C}) \uplus Var) \to (\mathbb{N} \uplus \{0,1\} \uplus \mathbb{Z})$ *in CDL is a total function defined as follows:*

(i) s *maps each variable* c^n *in* $RVar(\mathcal{C})$ *to a value in domain* $\mathbb{N}$*.*

(ii) s *maps each variable* c^s *in* $RVar(\mathcal{C})$ *to a value in domain* $\{0,1\}$*.*

(iii) s *maps each variable* x *in* Var *to a value in domain* $\mathbb{Z}$*.*

Given an expression E *and a state* s*, an evaluation* $Eval_s(E)$ *is defined as:*

(i) If $E = a$*, where* $a \in \{x, c^n, c^s\}$*, then* $Eval_s(a) =_{df} s(a)$*.*

(ii) If $E = n$*, then* $Eval_s(n) =_{df} n$*.*

(iii) If $E = f(E_1, E_2)$*, where* $f \in \{+, \cdot\}$*, then* $Eval_s(E) =_{df} f(Eval_s(E_1), Eval_s(E_2))$*.*

The semantics of CDL is defined by simultaneous induction in Def. 3.9, 3.14, 3.17 and 3.18 below. In the following subsections we first build the semantics of SEPs (Def. 3.9 and 3.14), then give the semantics of CDL formulae (Def. 3.17 and 3.18).

3.3.1. Semantics of SEPs

The semantics of SEPs consists of two parts, the semantics for sequential SEPs and the semantics for parallel SEPs. Since CDL contains path formulae ξ and $\sim\xi$, the semantics of SEPs is based on traces, i.e., the meaning of each formula is given as a set of traces. This is different from the traditional FODL.

Definition 3.9 (Semantics of Sequential SEPs)**.** *Given a set of clocks* $\mathcal{C}$ *and a set of general variables* Var*, for any sequential SEP* p*, its semantics is given as a Kripke frame* (S, val) *defined as follows:*

(i) $val(\mu) =_{df} S$*, where* S *is the set of all traces of length 1.*

(ii)
$$\begin{aligned} val(\alpha) =_{df} \{ss' \mid s, s' \in S;\ & \\ & \text{for each } c \text{ where } \varsigma^c \text{ is in } \alpha, s'(c^s) = 1 \wedge s'(c^n) = s(c^n) + 1; \\ & \text{for other } d \in \mathcal{C} \text{ not in } \alpha, s'(d^s) = 0 \wedge s'(d^n) = s(d^n); \\ & \text{for each } x := e \text{ in } \alpha, s'(x) = Eval_s(e); \\ & \text{for other } y \in Var \text{ not in } \alpha, s'(y) = s(y)\}. \end{aligned}$$

(iii) $val(P?\alpha) =_{df} \{ss' \mid s \in val(P), ss' \in val(\alpha)\}$*.*

(iv) $val(p; q) =_{df} val(p) \circ val(q)$*.*

(v) $val(p \cup q) =_{df} val(p) \cup val(q)$*.*

(vi) $val(p^\bullet) =_{df} (\bigcup_{n \geq 0} val^n(p)) \cup val^\omega(p)$*, where*

$$\begin{aligned} val^n(p) &=_{df} \underbrace{val(p) \circ val(p) \circ \ldots \circ val(p)}_{n}, \\ val^\omega(p) &=_{df} \underbrace{val(p) \circ val(p) \circ \ldots}_{\infty}, \\ val^0(p) &=_{df} S. \end{aligned}$$

(i) - (iii) give the definitions of the semantics of atomic SEPs. In (i), the event μ defines a set of traces with length 1. In (ii), the event α defines a transition from a state s to a state s'. In the state s', for each clock c whose corresponding signal ς^c is in α, the variable c^n recording the number of times clock c has ticked up to the current instant is increased by 1, and the variable c^s is set to 1, indicating at the current instant, clock c ticks. For each clock d whose corresponding signal ς^d is not in α, the variable d^n in state s' is kept unchanged while the variable d^s is set to 0, indicating at the current instant, d does not tick. For any assignment $x := e$ in α, the value of x in s' is set to the value of the expression e in s, while other general variables in both s and s' are kept unchanged. In (iii), traces satisfying $P?\alpha$ are exactly those traces satisfying p, adding that their beginning states must satisfy P. The definition of $val(P)$ is given in Def. 3.18 below.

(iv) - (vi) give the definitions of the semantics of compositional sequential SEPs. The semantics of the sequential operator ; and the choice operator $\cup$ are directly inherited from FODL [20]. In (vi), the semantics of the program $p^\bullet$ consists of all traces which are generated by executing p for a finite number of and infinitely many times. Intuitively, the traces of $p^\bullet$ can be divided into 4 types: (1) all traces of length 1, which corresponds to the semantics of the program μ; (2) all finite traces of the form $tr_1 \circ tr_2 \circ ... \circ tr_n$ where each $tr_i \in val(p)$ $(1 \leq i \leq n)$ is a finite trace; (3) all infinite traces of the form $tr_1 \circ tr_2 ... \circ tr_n \circ ...$ where each $tr_i \in val(p)$ $(i \geq 1)$ is a finite trace; (4) all infinite traces of the form $tr_1 \circ tr_2 ... \circ tr_m$ $(m \geq 1)$ where $tr_1, ..., tr_{m-1} \in val(p)$ are finite traces, while $tr_m \in val(p)$ is an infinite trace.

Example 3.7. *Let* $\alpha = (\varsigma^c | x := x + 1)$, $P = (x > 1)$, $\mathcal{C} = \{c, d\}$, $Var = \{x, y\}$, *if* $s = \{c^n \mapsto$ [2]$0, c^s \mapsto 0, d^n \mapsto 0, d^s \mapsto 0, x \mapsto 2, y \mapsto 0\}$, $s' = \{c^n \mapsto 1, c^s \mapsto 1, d^n \mapsto 0, d^s \mapsto 0, x \mapsto 3, y \mapsto 0\}$, *then* $ss' \in val(P?\alpha)$. *In Fig. 7, subfigure* $(P?\alpha)$ *gives a graphical illustration of trace* ss', *where the red state* s *satisfies the condition* P.

Subfigures $(p; q)$, $(p \cup q)$ and $(p^\bullet)$ of Fig. 7 show examples of how traces are constructed through the operators ;, $\cup$ and $\bullet$. Subfigure $(p; q)$ says that a trace of program $p; q$ can be constructed by a trace tr_1 of p and a trace tr_2 of q with a concatenation operator $\circ$ linking them. The trace colored blue belongs to p while the trace colored green belongs to q. The state s'' must belong to both p and q (which are colored half blue and half green). In subfigure $(p \cup q)$, we see that a trace of $p \cup q$ is either a trace of p (tr_1), or a trace of q (tr_2). The three traces of subfigure $(p^\bullet)$ correspond to 3 types of traces of $p^\bullet$ explained above respectively (the types (2), (3) and (4) above).

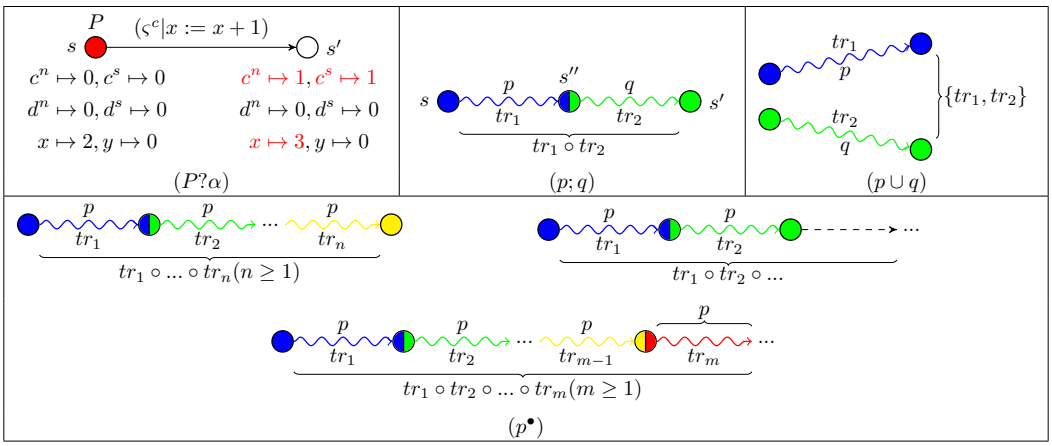

Figure 7: One of the traces of programs $P?\alpha$, $p; q$, $p \cup q$ and $p^\bullet$

The crucial point of defining the semantics for parallel SEPs is to define how two parallel SEPs interact with each other. For this purpose we introduce a language called *trec program*, following an idea from [29].

[2]we use $a \mapsto b$ to mean 'element a is mapped to element b' in a function.

A *trec* program can be considered as a *string* of an SEP, just like the string of the regular language. It characterizes one possible behaviour of an SEP. We first define the syntax of *trec* programs (Def. 3.10) and the semantics for sequential *trec* programs (Def. 3.11), based on which we then define the semantics for parallel *trec* programs (Def. 3.12), where we show how two *trec* programs interact with each other. Finally, based on the semantics of *trec* programs we define the semantics for parallel SEPs in Def. 3.14.

Definition 3.10 (Syntax of *trec* Programs)**.** *trec programs are a possibly infinite language, defined as follows:*

(a) μ *is a finite trec.*

(b) α *and* $\varrho\&P?\alpha$ *are finite trecs.*

(c) If $p_1, p_2, ..., p_n, q$ *are trecs (where* q *is finite), so are* $q; p_1$ *and* $\cap(p_1, p_2, ..., p_n)$*.*

(d) If p *is a finite trec, then* p^ω *is an infinite trec, where* $p^\omega = \underbrace{p; p; ...}_{\infty}$ *is an infinite word.*

To distinguish the parallel operator $\|$ in SEPs we adopted another operator '$\cap$' to express that n *trec* programs execute concurrently.

The semantics of sequential *trec* programs can be defined based on the semantics of sequential SEPs in Def. 3.9.

Definition 3.11 (Semantics of Sequential *trec* Programs)**.** *The semantics of a sequential trec program* p *is defined as follows based on the semantics of sequential SEPs in Def. 3.9:*

(i) $val(\mu)$*,* $val(\alpha)$*,* $val(\varrho\&P?\alpha)$ *and* $val(p; q)$ *are defined just as those in the semantics of sequential SEPs.*

(ii) $val(p^\omega) = val(p; p; ...) =_{df} \underbrace{val(p) \circ val(p) \circ val(p) \circ ....}_{\infty}$

Intuitively, a *trec* program characterizes exact one behaviour of an SEP, since it contains no choice operators $\cup$ and loop operators $\bullet$. The next example gives an intuitive view of how a *trec* program captures one behaviour of an SEP.

Example 3.8. *Consider a trec program*

$$p = \cap(\cap(\varsigma_1; \varsigma_4; \varsigma_1; \varsigma_4, \varsigma_2^\omega), \varsigma_3; \hat{\varsigma}_4?\varsigma_5; \varsigma_3; \varsigma_3; \varsigma_3),$$

where two programs $\varsigma_1; \varsigma_4; \varsigma_1; \varsigma_4$*,* ς_2^ω *and* $\varsigma_3; \hat{\varsigma}_4?\varsigma_5; \varsigma_3; \varsigma_3; \varsigma_3$ *are running concurrently. Fig. 8 shows the* computation tree *that exactly captures the behaviour of* p*. Each node of the tree is a state. Starting from the initial state, each intersecting surface (indicated by a dashed red rectangle) represents an instant. Transitions from a state of one intersecting surface to a state of another intersecting surface are triggered simultaneously (e.g., events* ς_4*,* ς_2 *and* $\hat{\varsigma}_4?\varsigma_5$*). trec* p *is an infinite string so its corresponding computation tree is infinite.*

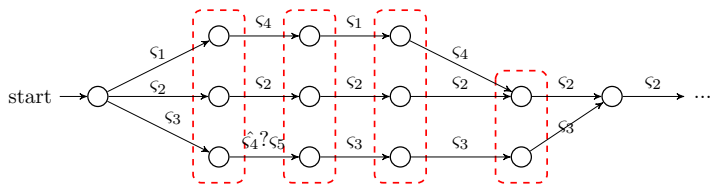

Figure 8: The computation tree of a *trec* program

Before defining the semantics for parallel *trec* programs, we introduce some auxiliary functions to ease this process. The function $Mer(a_1, ..., a_n)$ merges n events into one, where $a_i \in \{\alpha_i, \varrho_i\&P_i?\alpha_i\}$ $(1 \leq i \leq n)$. Given events $\beta_1, ..., \beta_m$ and test events $\varrho_1\&P_1?\alpha_1$,...,$\varrho_n\&P_n?\alpha_n$, we define

$$Mer(\beta_1, ..., \beta_m, \varrho_1\&P_1?\alpha_1, ..., \varrho_n\&P_n?\alpha_n) =_{df} P_1 \wedge ... \wedge P_n?(\beta_1|...|\beta_m|\alpha_1|...|\alpha_n),$$

where $(\beta_1|...|\beta_m|\alpha_1|...|\alpha_n)$ represents the combinational event whose events are exactly those in $\beta_1, ..., \beta_m$, $\alpha_1, ..., \alpha_n$.

The function $Mat(a, \varrho)$ matches the signal test condition ϱ with the event a, where $a \in \{\alpha, P?\alpha\}$. It returns *true* iff the match is successful as defined in the following rules:

(1) $Mat(a, \hat{\varsigma}(x)) =_{df} tt$ if ς is in α, $Mat(a, \hat{\varsigma}(x)) =_{df} ff$ if ς is not in α.

(2) $Mat(a, \bar{\varsigma}) =_{df} tt$ if ς is not in α, $Mat(a, \bar{\varsigma}) =_{df} ff$ if ς is in α.

(3) $Mat(a, \varrho_1 \veebar \varrho_2) =_{df} Mat(a, \varrho_1) \vee Mat(a, \varrho_2)$.

(4) $Mat(a, \varrho_1 \barwedge \varrho_2) =_{df} Mat(a, \varrho_1) \wedge Mat(a, \varrho_2)$.

For each condition of the forms $\hat{\varsigma_1}(x_1), ..., \hat{\varsigma_n}(x_n)$ in ϱ and each signal of the forms $\varsigma_1!e_1, ..., \varsigma_n!e_n$ in a (where $a \in \{\alpha, P?\alpha\}$), the function $Sub(p, a, \varrho)$ substitutes the variables $x_1, ..., x_n$ in p, which store the values of signals $\varsigma_1, ..., \varsigma_n$ in ϱ as the expressions $e_1, ..., e_n$ in a. It is defined as

$$Sub(p, a, \varrho) =_{df} p[e_1, ..., e_n/x_1, ..., x_n].$$

Definition 3.12 (Semantics of Parallel *trec* Programs)**.** *For any parallel trec program $\cap(p_1, ..., p_n)$, let*

$$a = Mer(a_{k_1}, ..., a_{k_m}),$$

where $a_{k_j} \in \{\alpha_{k_j}, P_{k_j}?\alpha_{k_j}\}$ $(1 \leq j \leq m)$. p_{k_1},...,p_{k_m} are all programs among the programs $p_1, ..., p_n$ such that $val(p_{k_1}) = val(a_{k_1}; p'_{k_1})$, ..., $val(p_{k_m}) = val(a_{k_m}; p'_{k_m})$ holds. The semantics of $\cap(p_1, ..., p_n)$ is defined as

$$val(\cap(p_1, ..., p_n)) =_{df} Com(\cap(p_1, ..., p_n), a),$$

where the function Com is defined as follows:

(i) If there exists a program r among the programs $p_1, ..., p_n$ such that $val(r) = val(\mu)$ holds, then

$$Com(\cap(p_1, ..., r, ..., p_n), a) =_{df} Com(\cap(p_1, ..., p_n), a).$$

(ii) If there exist programs $r_1, ..., r_m$ among the programs $p_1, ..., p_n$ such that $val(r_k) = val(\varrho_k \& P_k?\beta_k; q_k)$ $(1 \leq k \leq m)$ holds,

(1) if $Mat(a, \varrho_k)$ holds for all $1 \leq k \leq m$, then

$$Com(\cap(p_1, ..., r_1, ..., r_m, ..., p_n), a) =_{df} Com(\cap(p_1, ..., r'_1, ..., r'_m, ..., p_n), a'),$$

where

$$r'_k = P_k?Sub(\beta_k, a, \varrho_k); Sub(q_k, a, \varrho_k) \text{ for all } 1 \leq k \leq m,$$
$$a' = Mer(P_1?Sub(\beta_1, a, \varrho_1), ..., P_m?Sub(\beta_m, a, \varrho_m), a).$$

(2) if there exists $1 \leq k \leq m$ such that $Mat(a, \varrho_k)$ does not hold, then

$$Com(\cap(p_1, ..., r, ..., p_n), a) =_{df} \emptyset.$$

(iii) Otherwise (i.e., all programs $p_1, ..., p_n$ satisfy $val(p_1) = val(a_1; p'_1)$, ..., $val(p_n) = val(a_n; p'_n)$, where $a_j \in \{\alpha_j, P_j?\alpha_j\}$ for $1 \leq j \leq n$),

$$Com(\cap(p_1, ..., p_n), a) =_{df} val(a; \cap(p'_1, ..., p'_n)) = val(a) \circ val(\cap(p'_1, ..., p'_n)).$$

The semantics of $\cap(p_1, ..., p_n)$ is obtained by computing the parallel interaction between the *trec* programs at each instant. The computation is a process of continuously refining the event executed at the current instant. The symbol a represents the current observed event. At the beginning, we observe that all events of the form α or $P?\alpha$ in $p_1, ..., p_n$ are executable at the current instant. The function *Com* continuously updates the current observed event a according to the different situations ((i) - (iii)) until all events that can be executed at the current instant are observed. Situation (i) says that we simply neglect the event μ since it neither does anything nor consumes time. In situation (ii), we "unwrap" the condition of the events $\varrho_1\&P_1?\beta_1,...,\varrho_m\&P_m?\beta_m$ in the programs $r_1,...,r_m$ according to whether the event a matches the conditions $\varrho_1,...,\varrho_m$ ((ii) (1) and (ii) (2)). If the match is successful, we obtain a new observed event a' by adding the events that are unwrapped from $\varrho_1\&P_1?\beta_1,...,\varrho_m\&P_m?\beta_m$. If the match is unsuccessful, the program deadlock occurs so the semantics of $\cap(p_1, ..., p_n)$ is an empty set. In situation (iii), *Com* returns the event that is executed at the current instant if all test events have been already unwrapped.

Example 3.9. *Consider the trec program* $p = \cap(\cap(\varsigma_1; \varsigma_4; \varsigma_1; \varsigma_4, \varsigma_2^\omega), \varsigma_3; \hat{\varsigma_4}?\varsigma_5; \varsigma_3; \varsigma_3; \varsigma_3)$ *of Example 3.8, first there is*

$$\begin{aligned} val(\cap(\varsigma_1; \varsigma_4; \varsigma_1; \varsigma_4, \varsigma_2^\omega)) &= val(\varsigma_1|\varsigma_2) \circ val(\cap(\varsigma_4; \varsigma_1; \varsigma_4, \varsigma_2^\omega)) \\ &= val(\varsigma_1|\varsigma_2) \circ val(\varsigma_4|\varsigma_2) \circ val(\varsigma_1; \varsigma_4, \varsigma_2^\omega) \\ &= val(\varsigma_1|\varsigma_2) \circ val(\varsigma_4|\varsigma_2) \circ val(\varsigma_1|\varsigma_2) \circ val(\cap(\varsigma_4, \varsigma_2^\omega)) \\ &= val(\varsigma_1|\varsigma_2) \circ val(\varsigma_4|\varsigma_2) \circ val(\varsigma_1|\varsigma_2) \circ val(\varsigma_4|\varsigma_2) \circ ... \\ &= val((\varsigma_1|\varsigma_2); (\varsigma_4|\varsigma_2); (\varsigma_1|\varsigma_2); (\varsigma_4|\varsigma_2); ...), \end{aligned}$$

which means $\cap(\varsigma_1; \varsigma_4; \varsigma_1; \varsigma_4, \varsigma_2^\omega) \equiv (\varsigma_1|\varsigma_2); (\varsigma_4|\varsigma_2); (\varsigma_1|\varsigma_2); (\varsigma_4|\varsigma_2); ....$ *The semantics of* p *is given as:*

$$\begin{aligned} val(p) &= val(\cap((\varsigma_1|\varsigma_2); (\varsigma_4|\varsigma_2); (\varsigma_1|\varsigma_2); (\varsigma_4|\varsigma_2); ..., \varsigma_3; \hat{\varsigma_4}?\varsigma_5; \varsigma_3; \varsigma_3; \varsigma_3)) \\ &= val(\varsigma_1|\varsigma_2|\varsigma_3) \circ val(\cap((\varsigma_4|\varsigma_2); (\varsigma_1|\varsigma_2); (\varsigma_4|\varsigma_2); ..., \hat{\varsigma_4}?\varsigma_5; \varsigma_3; \varsigma_3; \varsigma_3)) \\ &= val(\varsigma_1|\varsigma_2|\varsigma_3) \circ val(\varsigma_4|\varsigma_2|\varsigma_5) \circ val(\cap((\varsigma_1|\varsigma_2); (\varsigma_4|\varsigma_2); ..., \varsigma_3; \varsigma_3; \varsigma_3)) \\ &= ... \\ &= val(\varsigma_1|\varsigma_2|\varsigma_3) \circ val(\varsigma_4|\varsigma_2|\varsigma_5) \circ val(\varsigma_1|\varsigma_2|\varsigma_3) \circ val(\varsigma_4|\varsigma_2|\varsigma_3) \circ .... \end{aligned}$$

The semantics of p *equals to the semantics of an infinite trec program*

$$(\varsigma_1|\varsigma_2|\varsigma_3); (\varsigma_4|\varsigma_2|\varsigma_5); (\varsigma_1|\varsigma_2|\varsigma_3); (\varsigma_4|\varsigma_2|\varsigma_3); ....$$

We define the set of *trec* programs of an SEP as the following definition.

Definition 3.13 (*trec* Set of SEPs)**.** *The trec set of an SEP* p*, denoted as* $\tau(p)$*, is inductively defined as follows:*

(i) $\tau(\mu) =_{df} \{\mu\}$.

(ii) $\tau(a) =_{df} \{a\}$*, where* $a \in \{\alpha, \varrho\&P?\alpha\}$.

(iii) $\tau(p; q) =_{df} \{r_1; r_2 \mid r_1 \in \tau(p), r_2 \in \tau(q), r_1 \text{ is finite}\} \cup \{r_1 \mid r_1 \in \tau(p), r_1 \text{ is infinite}\}$.

(iv) $\tau(p \cup q) =_{df} \tau(p) \cup \tau(q)$.

(v) $\tau(p^\bullet) =_{df} \{r \mid r \in \tau(p^n) \cup \tau(p^\omega), n \geq 0\}$*, where* $p^0 = \mu$*,* $p^n = \underbrace{p; ...; p}_{n}$ *and* $p^\omega = \underbrace{p; p; ...}_{\infty}$.

(vi) $\tau(\|(p_1, ..., p_n)) =_{df} \{\cap(r_1, ..., r_n) \mid r_1 \in \tau(p_1), ..., r_n \in \tau(p_n)\}$.

Intuitively, $\tau(p)$ defines the language of a program p, i.e., the set of all strings of p.

Based on Def. 3.12 and Def. 3.13, we define the semantics for parallel SEPs as the following definition.

Definition 3.14 (Semantics of Parallel SEPs)**.** *For any programs* $p_1, ..., p_n$*,*

$$val(\|(p_1, ..., p_n)) =_{df} \bigcup_{r_1 \in \tau(p_1), ..., r_n \in \tau(p_n)} val(\cap(r_1, ..., r_n)).$$

We use '$\equiv$' to express that two programs p and q are semantically equivalent, i.e., $p \equiv q$ iff $val(p) = val(q)$.

With the semantics of SEPs, now we introduce the notion of *executable event* in SEP, which is used in the formal analysis of SEPs below.

Definition 3.15 (Executable Events)**.** *Given a trec program* q*, its executable events, denoted as* $Evt(q)$*, is defined as follows:*

(i) $Evt(\mu) =_{df} \emptyset$.

(ii) $Evt(a) =_{df} \{a\}$*, where* $a \in \{\alpha, \varrho\&P?\alpha\}$.

(iii) $Evt(q_1; q_2) =_{df} Evt(q_1) \cup Evt(q_2)$.

(iv) $Evt(\cap(q_1, ..., q_n)) =_{df} Evt(q'; q'')$*, if* $\cap(q_1, ..., q_n) \equiv q'; q''$*,* $Evt(\cap(q_1, ..., q_n)) =_{df} \emptyset$ *otherwise.*

Given a program p*, its executable events are the set of executable events of all of its trec programs, defined as follows:*

$$Evt(p) =_{df} \bigcup_{r \in \tau(p)} Evt(r).$$

An event a *(where* $a \in \{\alpha, \varrho\&P?\beta\}$*) is a current executable event of* p *iff there exists a trec program* $q \in \tau(p)$ *such that* $q \equiv a; q'$.

Intuitively, an executable event of a program is an event that is possibly executed by the program at some instant. A current executable event of a program is an executable event at the current instant.

The next proposition shows that the definitions of *trec* programs: Def. 3.10 and 3.13 are in accord with our intuition about the language of SEPs. It says that the *trec* set of an SEP exactly captures all behaviours of the SEP.

Proposition 3.1 (A Relation between p and $\tau(p)$)**.** *Given an SEP* p*, there is*

$$val(p) = \bigcup_{r \in \tau(p)} val(r).$$

Prop. 3.1 can be proved by induction on the syntactic structure of the program p according to Def. 3.9 - 3.14.

Proof. For the basic case, we take event α for an example, the cases for the skip program μ and the event $\varrho\&P?\alpha$ are similar. Based on Def. 3.13, we have $\tau(\alpha) = \{\alpha\}$, so obviously $val(\alpha) = \bigcup_{r \in \tau(\alpha)} val(r) = val(\alpha)$.

For the inductive case, we take the sequence program $p; q$ for example, other cases are similar. Suppose $val(p) = \bigcup_{r \in \tau(p)} val(r)$ and $val(q) = \bigcup_{r \in \tau(q)} val(r)$ hold. According to Def. 3.9, we have $val(p; q) = val(p) \circ val(q) = (\bigcup_{r \in \tau(p)} val(r)) \circ (\bigcup_{r \in \tau(q)} val(r)) = \bigcup_{r_1 \in \tau(p), r_2 \in \tau(q)} val(r_1) \circ val(r_2) = \bigcup_{r_1 \in \tau(p), r_2 \in \tau(q)} val(r_1; r_2)$. According to Def. 3.13, we immediately obtain the result.

□

Based on the semantics of SEPs we derive some semantic equivalence relations between programs, which play an important role in the rewrite rules for parallel SEPs later given in Section 4.3.

Proposition 3.2. *For any SEPs* p, q, r*,* $p_1, ..., p_n$*, there are equivalence relations between SEPs listed as follows:*

(i) Relations for μ*:* $\mu; p \equiv p \equiv p; \mu$*,* $\mu^\bullet \equiv \mu$*,* $\|(p_1, ..., p_{i-1}, p_i, p_{i+1}, ..., p_n) \equiv \|(p_1, ..., p_{i-1}, p_{i+1}, ..., p_n)$ *(where* $p_i = \mu$*).*

(ii) Associative law for ;: $(p;q);r \equiv p;(q;r)$.

(iii) Commutative, associative laws for $\cup$*:* $p \cup q \equiv q \cup p$, $(p \cup q) \cup r \equiv p \cup (q \cup r)$.

(iv) Distributive law of ; over $\cup$*:* $(p \cup q);r \equiv p;r \cup q;r$, $r;(p \cup q) \equiv (r;p) \cup (r;q)$.

(v) Relation for $\bullet$*:* $p^\bullet \equiv \mu \cup p;p^\bullet$.

(vi) Commutative law for $\|$*:* $\|(p_1, ..., p_i, ..., p_j, ..., p_n) \equiv \|(p_1, ..., p_j, ..., p_i, ..., p_n)$.

(vii) Distributive law of $\|$ *over* $\cup$*:* $\|(p_1, ..., (p_{i,1} \cup p_{i,2}), ..., p_n) \equiv \|(p_1, ..., p_{i,1}, ..., p_n) \cup \|(p_1, ..., p_{i,2}, ..., p_n)$.

All relations in Prop. 3.2 can be directly obtained from the semantics of SEPs. (i)-(iv) and (vi) are obvious. The relations in (i) shows that μ is a neural element for the sequential operator ;, the loop operator $\bullet$ and the parallel operator $\|$. Intuitively, in (v) the loop program $p^\bullet$ executing for 0 or more times can be split into a program executing for 0 time (μ) and a program executing for 1 or more times ($p;p^\bullet$). (vii) is a direct result of Def. 3.14.

The Proof of Prop. 3.2. We only give the proof for (v) as an example, other relations are similar. On the one hand, according to Def. 3.9, we have $val(p^\bullet) = (\bigcup_{n\geq 0} val^n(p)) \cup val^\omega(p) = val^0(p) \cup (\bigcup_{n\geq 1} val^n(p)) \cup val^\omega(p) = val(\mu) \cup (\bigcup_{n\geq 1} val^n(p)) \cup val^\omega(p)$. On the other hand, we have $val(p;p^\bullet) = val(p) \circ val(p^\bullet) = val(p) \circ ((\bigcup_{n\geq 0} val^n(p)) \cup val^\omega(p)) = (val(p) \circ (\bigcup_{n\geq 0} val^n(p))) \cup (val(p) \circ val^\omega(p)) = (\bigcup_{n\geq 0} val(p) \circ val^n(p)) \cup (val(p) \circ val^\omega(p)) = (\bigcup_{n\geq 1} val^n(p)) \cup val^\omega(p)$. So we have $val(p^\bullet) = val(\mu) \cup val(p;p^\bullet) = val(\mu \cup p;p^\bullet)$. □

3.3.2. Consistent SEPs

In synchronous models, due to the basic assumption that events can occur simultaneously at an instant, the behaviour of programs may cause logical inconsistencies [28]. In SEP, similar problem exists because of the synchronous execution mechanism we introduce. We classify the logical inconsistencies in SEP into two types:

(1) At some instant, there are two simultaneous assignments to a single variable or two occurrences of a single signal. This must be avoided since in the combinational event, all events are assumed to be executed simultaneously and there is no notion of *micro steps* with which the order between these events can be defined.

(2) At some instant, an event contradicts some condition of this event. This must be avoided because, intuitively, a signal must be emitted right after its test condition is satisfied.

Based on these two types of logical inconsistencies, we formally define a set of *consistent SEPs* to confine the semantics of SEPs given in Def. 3.9 and 3.12. In this paper, we only focus on the SEPs whose semantics matches the following definition.

Definition 3.16 (Consistent SEPs)**.** *An SEP p is consistent iff it satisfies the following properties:*

(i) For any executable event $a \in Evt(p)$, there exist at most one assignment $x := e$ for each variable x, and at most one signal $\varsigma!e$ for each signal ς.

(ii) For any parallel program $\|(p_1, ..., p_n)$ in p and any executable event $a \in Evt(\|(p_1, ..., p_n))$, for each signal ς in a, if ς appears in an event $\varrho?\alpha'$ in $p_1, ..., p_n$, then $Mat(a, \varrho)$ must hold.

In Def. 3.16, properties (i) and (ii) correspond to the types of logical inconsistencies (1) and (2) respectively.

Example 3.10. *Programs $p_1 = (x := 1|x := 2)$ and $p_2 = (\varsigma!2|\varsigma!3)$ are not consistent, since they do not satisfy Def. 3.16 (i). Program $p_3 = \|(\bar{\varsigma_1}?\varsigma_2, \bar{\varsigma_2}?\varsigma_1)$ is not consistent, since it does not satisfy Def. 3.16 (ii): on the one hand, its current executable event is $(\varsigma_1|\varsigma_2)$ according to Def. 3.12, on the other hand, ς_2 comes from event $\bar{\varsigma_1}?\varsigma_2$, but $(\varsigma_1|\varsigma_2)$ does not satisfy $\bar{\varsigma_1}$.*

In other synchronous languages like Esterel [12], a *constructive semantics* was proposed to preserve the consistency of the program, where an algorithm was developed to check whether a program satisfies the consistent properties (similar to the properties given in Def. 3.16). In this paper, we mainly focus on the CDL and its proof system, so instead of giving a constructive semantics of SEPs, we restrict ourselves to focus only on the consistent SEPs.

3.3.3. Semantics of CDL

In Section 2.1, clock relations are defined under a schedule σ and a configuration $\mathcal{X}_\sigma$. In order to introduce clock relations into dynamic logic we define the semantics of clock relations in the Kripke frame of CDL.

Definition 3.17 (Semantics of Clock Relations in Kripke Frames)**.** *Given a trace tr, the semantics of clock relations $tr \vDash_{ccsl} \xi$ is defined as:*

$$tr \vDash_{ccsl} \xi \text{ iff for each } i \in \mathbb{N}^+, tr(i) \in val(\hbar(\xi)),$$

where the function $\hbar(\xi)$ returns an AFOL formula that describes what should be held at each instant for each clock relation. It is given as follows:

(1) $\hbar(c_1 \subseteq c_2) =_{df} c_1^s = 1 \rightarrow c_2^s = 1$.

(2) $\hbar(c_1 \# c_2) =_{df} c_1^s = 0 \vee c_2^s = 0$.

(3) $\hbar(c_1 \prec c_2) =_{df} c_1^n > c_2^n \vee (c_1^n = c_2^n \rightarrow c_1^s = 0)$.

(4) $\hbar(c_1 \preceq c_2) =_{df} c_1^n \geq c_2^n$.

(5) $\hbar(\curlywedge(Rel_1, ..., Rel_n) =_{df} \bigwedge_{1 \leq i \leq n} \hbar(Rel_i)$.

A trace satisfies a clock relation iff each of its states (from the second state $tr(1)$) satisfies an AFOL formula that exactly captures the corresponding condition of this clock relation in Table 1, which should hold at each instant in a schedule. The semantics of $val(\hbar(\xi))$ is defined in Def. 3.18 below.

From Def. 3.17 we can see a connection between the semantics of clock relations in CCSL (Table 1) and that in the Kripke frame of CDL, described in the following proposition.

Proposition 3.3 ($tr \vDash_{ccsl} \xi$ vs. $\sigma \vDash_{ccsl} \xi$)**.** *Any trace tr that satisfies $tr(0)(c^n) = 0$ and $tr(0)(c^s) = 0$ for any clock c is called an* initial trace*. For any schedule σ, there exists an initial trace, denoted by tr^σ, such that*

(i) $\sigma(0) = \emptyset$.

(ii) for each clock c and $i \geq 0$, $\mathcal{X}_\sigma(c, i) = tr^\sigma(i)(c^n)$.

(iii) for each clock c and $i \geq 0$, $c \in \sigma(i)$ *iff* $tr^\sigma(i)(c^s) = 1$.

More, there is

$$tr^\sigma \vDash_{ccsl} \xi \quad \textit{iff} \quad \sigma \vDash_{ccsl} \xi.$$

Given an initial trace tr_0, if there exists a schedule σ_0 such that $tr_0 = tr^\sigma$, then we often denote this schedule as σ^{tr_0}.

Prop. 3.3 can be proved by induction on the structure of ξ based on Table 1, Def. 3.17 and Def. 3.18.

Proof. For basic cases, we take $c_1 \prec c_2$ as an example, other basic cases are similar. On the one direction, if $tr^\sigma \vDash_{ccsl} c_1 \prec c_2$ holds, we now prove $\sigma \vDash_{ccsl} c_1 \prec c_2$. By Def. 3.17 we have for any $i \in \mathbb{N}^+, tr^\sigma(i) \in val(\hbar(c_1 \prec c_2))$. According to the definition of $\hbar$ in Def. 3.17 and Def. 3.18 below we have $tr^\sigma(i)(c_1^n) > tr(i)(c_2^n) \vee (tr^\sigma(i)(c_1^n) = tr^\sigma(i)(c_2^n) \rightarrow tr^\sigma(i)(c_1^s) = 0)$ holds for any $i \in \mathbb{N}^+$. By the construction of tr^σ, we know that for any $i \geq 0$, $\mathcal{X}_\sigma(c, i) = tr^\sigma(i)(c^n)$ holds, where $c \in \{c_1, c_2\}$. Therefore, according to the semantics of CCSL in Table 1, we get $\sigma \vDash_{ccsl} c_1 \prec c_2$. The proof for the other direction is similar.

The inductive case $\xi = \curlywedge(Rel_1, ..., Rel_n)$ is obvious. □

Prop. 3.3 says that for any schedule σ, we can have an initial trace tr^σ whose behaviour can be exactly captured by σ. Moreover, the satisfaction relation $tr^\sigma \vDash_{ccsl} \xi$ exactly captures the semantics of ξ in CCSL.

When tr is not an initial trace, the schedule σ^{tr} satisfying conditions (i) - (iii) above does not exist. In this case, $tr \vDash_{ccsl} \xi$ actually captures partial information about the satisfaction $\sigma \vDash_{ccsl} \xi$ of some schedule σ, where ρ, one of suffixes of σ, satisfies conditions (ii) - (iii) above.

Fig. 9 shows a corresponding relation between a schedule σ and its corresponding initial trace $tr^\sigma = s_0 s_1 ... s_i ....$ Intuitively, according to the semantics of SEPs, each state of the trace tr^σ exactly represents an instant in the schedule σ. The combinational event executed at the instant i ($i \geq 1$) is α_i.

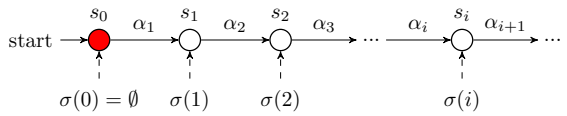

Figure 9: The corresponding relation between trace tr and schedule σ^{tr}

Example 3.11. *For schedule $\sigma = \{\}\{c\}$, its corresponding initial trace is the trace $tr = ss'$ discussed in Example 3.7, where $\mathcal{X}_\sigma(c,0) = tr(0)(c^n) = 0$, $\mathcal{X}_\sigma(d,0) = tr(0)(d^n) = 0$, $\mathcal{X}_\sigma(c,1) = tr(1)(c^n) = 1$, $\mathcal{X}_\sigma(d,1) = tr(1)(d^n) = 0$. $c, d \notin \sigma(0)$ since $tr(0)(c^s) = tr(0)(d^s) = 0$, $c \in \sigma(1)$ since $tr(1)(c^s) = 1$.*

Definition 3.18 (Semantics of CDL Formulae)**.** *Given a set of clocks $\mathcal{C}$ and a set of general variables Var, for any CDL formula ϕ, its semantics is given as a Kripke frame (S, val) defined as follows:*

(i) $val(tt) =_{df} S$.

(ii) $val(E \leq E') =_{df} \{s \mid Eval_s(E) \leq Eval_s(E')\}$.

(iii) $val([p]\xi) =_{df} \{s \mid$ *for all* tr *with* $s = tr_b$ *and* $tr \in val(p), tr \vDash_{ccsl} \xi\}$.

(iv) $val([p]\phi) =_{df} \{s \mid$ *for all* $tr \in val(p)$ *with* $tr_b = s$, *if* tr_e *exists, then* $tr_e \in val(\phi)\}$.

(v) $val(\neg\phi) =_{df} \{s \mid s \notin val(\phi)\}$.

(vi) $val(\phi \wedge \varphi) =_{df} val(\phi) \cap val(\varphi)$.

(vii) $val(\forall x.\phi) =_{df} \{s \mid$ *for each* $v_0 \in \mathbb{Z}, s \in val(\phi[v_0/x])\}$.

We introduce a symbol $\models_{cdl}$ to express the satisfaction relation between a CDL formula ϕ and a state s in set $val(\phi)$. s satisfies ϕ, denoted by $s \models_{cdl} \phi$, is defined as:

$$s \models_{cdl} \phi \text{ iff } s \in val(\phi).$$

If for all state s, $s \models_{cdl} \phi$ holds, then ϕ is said to be valid, denoted as $\models_{cdl} \phi$.

The semantics of CDL formulae is based on states. In (iii), a trace satisfying a clock relation is from the second state of the trace due to Def. 3.17. So state s itself is unrelated to ξ. (iv)-(vii) are similar to the definitions in FODL [20], except that the semantics of SEPs is based on traces. (iv) requires that the trace must be finite, indicating that we only consider the *partial correctness* of ϕ for the program p: it only matters whether ϕ holds on those states on which the program p terminates.

Fig. 10 gives an illustration of $[p]\phi$ and $[p]\xi$, where the snake arrow indicates an executing trace (which could be infinite) of the program. The blue states satisfy $[p]\phi$ (resp. $[p]\xi$), while the red states (resp. traces) satisfy the formula ϕ (resp. the relation ξ).

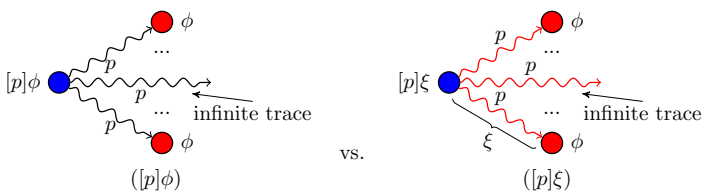

Figure 10: The semantics of formulae $[p]\phi$ and $[p]\xi$

4. The Proof System of CDL

In this section we propose a proof system of CDL, which provides a modular way of transforming a CDL formula into QF-AFOL formulae. As indicated in Fig. 2, the proof system of CDL is based on that of FODL [20]. It consists of the rules for sequential SEPs and the rewrite rules for parallel SEPs. The rules for sequential SEPs are mainly based on those for the program models of FODL. For parallel SEPs, the rewrite rules provide an automatic way of transforming a parallel program into a sequential one.

In Section 4.1 we introduce a concept of *sequent* that is used in the rest of the section as a logical argumentation to express proof rules. In Section 4.2 we build the rules for sequential SEPs, then in Section 4.3 we build the rewrite rules for parallel SEPs and introduce the algorithms for reducing a program in parallel form into a program in sequential form. In Section 4.4 we illustrate how to use this proof system to verify CCSL specifications of synchronous systems by analyzing a simple example.

4.1. Sequent Calculus

A sequent, a concept first proposed by Gentzen [30], is a form to express logical formulae and their proofs. A sequent is of the form

$$\Gamma \Rightarrow \Delta,$$

where Γ, Δ are two finite *multi-sets* of logical formulae. The semantics of sequent $\Gamma \Rightarrow \Delta$ is that of formula

$$\bigwedge_{\phi \in \Gamma} \phi \rightarrow \bigvee_{\varphi \in \Delta} \varphi,$$

meaning that if all formulae of Γ hold, then at least some formula of Δ holds. When Γ or Δ is empty, we use $\cdot$ to denote it.

A rule in sequent calculus is of the form:

$$\frac{\Gamma_1 \Rightarrow \Delta_1 \quad ... \quad \Gamma_n \Rightarrow \Delta_n}{\Gamma \Rightarrow \Delta} \quad .$$

It means that if $\Gamma_1 \Rightarrow \Delta_1$, ..., $\Gamma_n \Rightarrow \Delta_n$ are all true, then $\Gamma \Rightarrow \Delta$ is true. Each $\Gamma_i \Rightarrow \Delta_i$ in the upper part is called a *premise*, while $\Gamma \Rightarrow \Delta$ in the lower part is called a *conclusion*. We use $\frac{\Gamma' \Rightarrow \Delta'}{\Gamma \Rightarrow \Delta}$ to represent a pair of sequent rules: $\frac{\Gamma' \Rightarrow \Delta'}{\Gamma \Rightarrow \Delta}$ and $\frac{\Gamma \Rightarrow \Delta}{\Gamma' \Rightarrow \Delta'}$. It means that $\Gamma \Rightarrow \Delta$ is true iff $\Gamma' \Rightarrow \Delta'$ is true. Sometimes we write $\frac{\varphi}{\phi}$ if for all Γ and Δ, $\frac{\Gamma \Rightarrow \varphi, \Delta}{\Gamma \Rightarrow \phi, \Delta}$ holds. It is easy to prove that $\frac{\varphi}{\phi}$ just means φ implies ϕ. We call Γ and Δ the *contexts* of formula ϕ in sequent $\Gamma \Rightarrow \phi, \Delta$ or $\Gamma, \phi \Rightarrow \Delta$.

The derivation of a sequent forms a *proof tree*, where each node is a sequent. A *derivation* can be expressed in the form: $\frac{\zeta_1, ..., \zeta_n}{\zeta}$ (r), where nodes ζ, ζ_1,...,ζ_n are sequents, r is the name of the rule. We use $\frac{\zeta_1, ..., \zeta_n}{\zeta}$ $(r_1, ..., r_m)$ to express a series of derivations of the rules $r_1, ..., r_m$ $(m \geq 1)$. It means from node ζ, by applying rules $r_1, ..., r_m$, we obtain nodes $\zeta_1, ..., \zeta_n$.

4.2. Proof Rules for Sequential SEPs

The proof rules for sequential SEPs consist of: (1) the special rules for combinational events and formulae of the form $[p]\xi$; (2) the rules inherited from FODL for formulae of the form $[p]\phi$, and other FOL rules.

4.2.1. Special Rules for CDL

Table 3 gives the special rules in CDL, where rules $(\pi[])$, $(\phi[])$, $(\pi\langle\rangle)$, $(\phi\langle\rangle)$, $(P?[])$, $(\pi\mu[])$ and $(\mu[])$ are for combinational events, other rules are for formulae of the form $[p]\xi$. The rules for formulae of the form $\langle p\rangle \sim \xi$ can be derived from the corresponding rules in Table 3. They are given in Table A.11 in Appendix A.2.

In rule $(\pi[])$, we set $\alpha = (c|x := e)$ as an example of combinational events. The ellipses '.....' mean that we omit the discussion about other signals and assignments in α. Intuitively, rule $(\pi[])$ says that under some contexts Γ and Δ, formula $[\alpha]\xi$ holds iff the premise holds, where all dynamic variables (c^n, c^s, x, d_i^s, $1 \leq i \leq n$) are updated with new values according to α, and their old values are stored in a set of new variables V'. The vector equivalence $(x_1, ..., x_n) = (e_1, ..., e_n)$ is a shorthand for the equivalences $x_1 = e_1, ..., x_n = e_n$. $d_1, ..., d_n$ are all clocks not appearing in α. $\mathcal{C}(\alpha)$ and $\mathcal{V}(\alpha)$ represent the set of all clocks and the set of all dynamic variables in α respectively. V' is the set of new variables corresponding to V : for each variable $x \in V$, there is a new variable x' (with respect to $\Gamma, [\alpha]\xi, \Delta$) corresponding to it. The function $\hbar$ (Def. 3.17) maps each clock relation to an AFOL formula which should hold at state s'.

Rules $(\pi\langle\rangle)$, $(\phi[])$ and $(\phi\langle\rangle)$ are similar to rule $(\pi[])$. We omit their explanations here.

Example 4.1. *Given the program* $(\varsigma^c|x := x+1)$ *of Example 3.7, consider a CCSL specification* $[(\varsigma^c|x := x+1)]c \preceq d$ *under the contexts* $\Gamma = \{c^n = 0, c^s = 0, d^n = 0, d^s = 0, x = 2, y = 0\}$ *and* $\Delta = \emptyset$*, by applying rule* $(\pi[])$*, we have the derivation*

$$\frac{\left\{\begin{array}{c}(c^n)' = 0, (c^s)' = 0,\\ d^n = 0, (d^s)' = 0,\\ x' = 2, y = 0\end{array}\right\}, \quad \begin{array}{c}c^n = (c^n)' + 1, c^s = 1, x = x' + 1,\\ d^s = 0\end{array} \Rightarrow c^n \geq d^n}{c^n = 0, c^s = 0, d^n = 0, d^s = 0, x = 2, y = 0 \Rightarrow [(\varsigma^c|x := x+1)]c \preceq d}\ (\pi[])$$

where $(c^n)'$, $(c^s)'$, $(d^s)'$, x' *are new variables corresponding to* c^n, c^s, d^s, x *respectively. The formulae in* Γ *that depend on the old value of each variable is given in braces '{}' in the premise.* y *is a static variable. By rule* $(\pi[])$*, the formula* $[(\varsigma^c|x := x+1)]c \preceq d$ *is transformed into a set of QF-AFOL formulae in the premise.*

Rule $(P?[])$ says that under any contexts, the formula $[P?\alpha]A$ is true iff if P is true, then $[\alpha]A$ is true. When P is false, the program $P?\alpha$ reaches a deadlock, so $[P?\alpha]A$ always holds. Rule $(\pi\mu[])$ says that when a program is a skip program, $[\mu]\xi$ always holds. This is because the skip program μ neither does anything nor consumes time. From another perspective, $tr \vDash_{ccsl} \xi$ always holds for a trace tr with length 1. Rule $(\mu[])$ is obviously sound because the state does not change after the execution of μ.

Rules $(\pi[;])$, $(\pi[\cup])$, $(\pi[\bullet]u)$ and $(\pi[\bullet]i)$ for the formula $[p]\xi$ are modular rules and they are inspired by the rules for the temporal formula $\Box\phi$ proposed in [31]. Rule $(\pi[;])$ means that every trace of $p;q$ satisfies ξ iff every trace of p satisfies ξ, and after p every trace of q satisfies ξ. Rule $(\pi[\cup])$ says that every trace of $p \cup q$ satisfies ξ iff every trace of p and q satisfies ξ. Rule $(\pi[\bullet]u)$ unwinds the operator $\bullet$. It means that to prove the traces of p satisfy ξ we only need to show all traces with length ≥ 2 (i.e., the traces of $p;p^\bullet$) satisfy ξ. Rule $(\pi[\bullet]i)$ states that ξ holds along all traces of $p^\bullet$, iff after all times of repetitions of p (i.e., again, $p^\bullet$), ξ holds along all traces of p. Fig. 11 gives a graphical illustration of these 4 rules, where red traces satisfy ξ, the states are tagged with the formulae that they satisfy.

4.2.2. Rules from FODL

Table 4 gives the rules for formulae of the form $[p]\phi$. They are directly inherited from FODL [20]. The only slight difference is that the loop program $p^\bullet$ in SEP allows p to execute for infinitely many times, which is different from the program p^* in FODL. However, the rules for $p^\bullet$ share the same form as the rules for p^* in FODL because we only consider the partial correctness of the program $p^\bullet$. In other words, it is enough to only consider finite traces of $p^\bullet$.

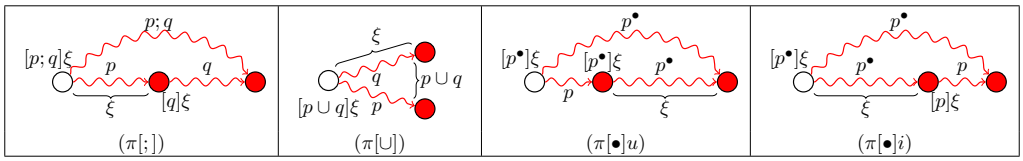

Figure 11: Graphical illustrations of rules $(\pi[;])$, $(\pi[\cup])$, $(\pi[\bullet]u)$ and $(\pi[\bullet]i)$

Other rules for formulae of the form $\langle p\rangle\phi$ can be derived from the corresponding rules in Table 4. They are given in Table A.11 in Appendix A.2.

As shown in Table 4, rule $([;])$ describes that ϕ holds after $p;q$ iff $[q]\phi$ holds after p. Rule $([\cup])$ says ϕ holds after $p \cup q$ iff ϕ holds after p, and also holds after q. Rule $([\bullet]u)$ means that ϕ holds after all times of repetitions of p, iff ϕ holds at the current state, and ϕ holds after $p;p^\bullet$. Fig. 12 gives a graphical illustration of rules $([;])$, $([\cup])$ and $([\bullet]u)$, where each red state satisfies the formula ϕ.

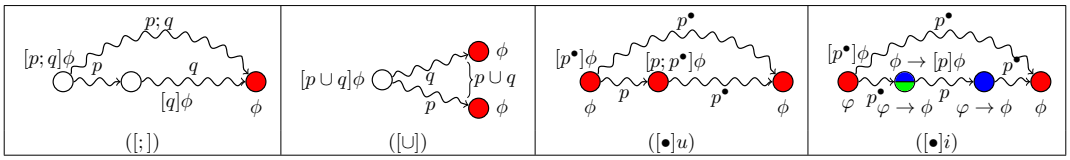

Figure 12: Graphical illustrations of rules $([;])$, $([\cup])$, $([\bullet]u)$ and $([\bullet]i)$

Rules $([]gen)$ and $(\langle\rangle gen)$ strengthen the conclusions by extending the proposition $\phi \to \varphi$ into dynamic situations. Rule $([]gen)$ (resp. $(\langle\rangle gen)$) expresses that if $\phi \to \varphi$ holds under any contexts, then under the contexts Γ and Δ after all (resp. some) executions of p, ϕ implies φ. Fig. 13 gives an intuitive illustration of rule $([]gen)$, where the red states satisfy the formula $\phi \to \varphi$, and the blue state satisfies the formula $[p]\phi \to [p]\varphi$.

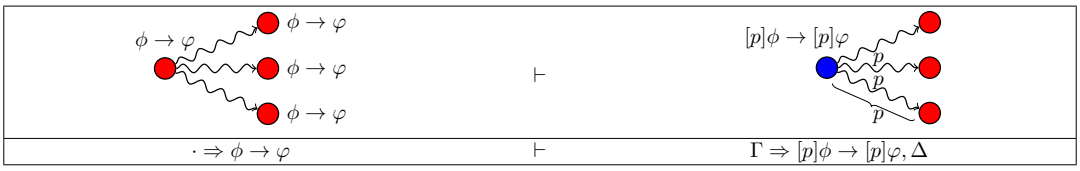

Figure 13: Graphical illustrations of rule $([]gen)$

Rule $([\bullet]ind)$ is the mathematical induction by the number of repetitions of the program p: to prove ϕ holds after all repetitions (including 0) under the given contexts Γ and Δ, we need to prove that under any contexts, if ϕ holds, then it also holds after p. Rule $(\langle\bullet\rangle con)$ is from the Harel's convergence rule in [20] where the integer x indicates the existing number of the repetitions of p. Its meaning is similar to rule $([\bullet]ind)$ and we omit it here. Rules $([\bullet]i)$ and $(\langle\bullet\rangle i)$ are for eliminating the star operator $\bullet$ in practical verification. They can be derived by rules $([\bullet]ind)$ and $(\langle\bullet\rangle con)$ with the generalization rules $([]gen)$ and $(\langle\rangle gen)$ (see [20, 32]). φ is a loop invariant of p. Rule $([\bullet]i)$ says that to prove ϕ holds after all repetitions of p under the given contexts Γ and Δ, we need to prove that there exists an invariant φ such that: (i) φ holds under the contexts Γ and Δ. (ii) Under any contexts, if φ holds, then φ holds after p as well. (iii) Under any contexts, φ implies ϕ. Figure $([\bullet]i)$ of Fig. 12 gives a graphical illustration of rule $([\bullet]i)$, where the red states satisfy the formula ϕ, the blue states satisfy the formula $\varphi \to \phi$, the green state satisfies the formula $\phi \to [p]\phi$. The explanation of rule $(\langle\bullet\rangle i)$ is similar to rule $([\bullet]i)$, where we need a number x to indicate the number of repetitions of p before it terminates. Without x the premise might become too strong so that its falsehood does not imply the falsehood of the conclusion. We omit its explanation here.

$$\frac{\Gamma[V'/V], c^n = (c^n)' + 1, c^s = 1, x = e'[V'/V], ..., (d_1^s, ..., d_n^s) = \underbrace{(0, ..., 0)}_{n} \Rightarrow \hbar(\xi), \Delta[V'/V]}{\Gamma \Rightarrow [\alpha]\xi, \Delta} \; (\pi[])$$

where $\alpha = (\varsigma^c!e \mid x := e' \mid .....).\ \{d_1, ..., d_n\} = \mathcal{C} - \mathcal{C}(\alpha).\ V = \mathcal{V}(\alpha)$, V' is the set of new variables (w.r.t. $\Gamma, [\alpha]\xi, \Delta$) corresponding to V.

$$\frac{\Gamma[V'/V], c^n = (c^n)' + 1, c^s = 1, x = e'[V'/V], ..., (d_1^s, ..., d_n^s) = \underbrace{(0, ..., 0)}_{n} \Rightarrow \phi, \Delta[V'/V]}{\Gamma \Rightarrow [\alpha]\phi, \Delta} \; (\phi[])$$

where $\alpha = (\varsigma^c!e \mid x := e' \mid .....).\ \{d_1, ..., d_n\} = \mathcal{C} - \mathcal{C}(\alpha).\ V = \mathcal{V}(\alpha)$, V' is the set of new variables (w.r.t. $\Gamma, [\alpha]\xi, \Delta$) corresponding to V.

$$\frac{\Gamma[V'/V], c^n = (c^n)' + 1, c^s = 1, x = e'[V'/V], ..., (d_1^s, ..., d_n^s) = \underbrace{(0, ..., 0)}_{n} \Rightarrow \neg\hbar(\xi) \Rightarrow \Delta[V'/V]}{\Gamma \Rightarrow \langle\alpha\rangle \sim \xi \Rightarrow \Delta} \; (\pi\langle\rangle)$$

where $\alpha = (\varsigma^c!e \mid x := e' \mid .....).\ \{d_1, ..., d_n\} = \mathcal{C} - \mathcal{C}(\alpha).\ V = \mathcal{V}(\alpha)$, V' is the set of new variables (w.r.t. $\Gamma, \langle\alpha\rangle\xi, \Delta$) corresponding to V.

$$\frac{\Gamma[V'/V], c^n = (c^n)' + 1, c^s = 1, x = e'[V'/V], ..., (d_1^s, ..., d_n^s) = \underbrace{(0, ..., 0)}_{n} \Rightarrow \phi, \Delta[V'/V]}{\Gamma \Rightarrow \langle\alpha\rangle\phi, \Delta} \; (\phi\langle\rangle)$$

where $\alpha = (\varsigma^c!e \mid x := e' \mid .....).\ \{d_1, ..., d_n\} = \mathcal{C} - \mathcal{C}(\alpha).\ V = \mathcal{V}(\alpha)$, V' is the set of new variables (w.r.t. $\Gamma, \langle\alpha\rangle\xi, \Delta$) corresponding to V.

$$\frac{P \rightarrow [\alpha]A}{[P?\alpha]A} \; (P?[]) \qquad \frac{tt}{[\mu]\xi} \; (\pi\mu[]) \qquad \frac{\phi}{[\mu]\phi} \; (\mu[])$$

where $A \in \{\xi, \phi\}$

$$\frac{[p]\xi \wedge [p][q]\xi}{[p; q]\xi} \; (\pi[;]) \qquad \frac{[p]\xi \wedge [q]\xi}{[p \cup q]\xi} \; (\pi[\cup]) \qquad \frac{[p; p^\bullet]\xi}{[p^\bullet]\xi} \; (\pi[\bullet]u) \qquad \frac{[p^\bullet][p]\xi}{[p^\bullet]\xi} \; (\pi[\bullet]i)$$

Table 3: Rules for combinational events and formulae $[p]\xi$

$\dfrac{[p][q]\phi}{[p;q]\phi}\ ([;])$	$\dfrac{[p]\phi \wedge [q]\phi}{[p \cup q]\phi}\ ([\cup])$	$\dfrac{\phi \wedge [p;p^\bullet]\phi}{[p^\bullet]\phi}\ ([\bullet]u)$
$\dfrac{\cdot \Rightarrow \phi \to \varphi}{\Gamma \Rightarrow [p]\phi \to [p]\varphi, \Delta}\ ([]gen)$	$\dfrac{\cdot \Rightarrow \phi \to \varphi}{\Gamma \Rightarrow \langle p\rangle\phi \to \langle p\rangle\varphi, \Delta}\ (\langle\rangle gen)$	
$\dfrac{\cdot \Rightarrow \phi \to [p]\phi}{\Gamma \Rightarrow \phi \to [p^\bullet]\phi, \Delta}\ ([\bullet]ind)$	$\dfrac{\cdot \Rightarrow \forall x > 0.(\phi(x) \to \langle p\rangle\phi(x-1))}{\Gamma \Rightarrow \exists x \geq 0.\phi(x) \to \exists x \leq 0.\langle p^\bullet\rangle\phi(x), \Delta}\ (\langle\bullet\rangle con)$	
$\dfrac{\Gamma \Rightarrow \varphi, \Delta \quad \cdot \Rightarrow \varphi \to [p]\varphi \quad \cdot \Rightarrow \varphi \to \phi}{\Gamma \Rightarrow [p^\bullet]\phi, \Delta}\ ([\bullet]i)$		
$\dfrac{\Gamma \Rightarrow \exists x \geq 0.\varphi(x), \Delta \quad \cdot \Rightarrow \forall x > 0.(\varphi(x) \to \langle p\rangle\varphi(x-1)) \quad \cdot \Rightarrow \exists x \leq 0.\varphi(x) \to \phi}{\Gamma \Rightarrow \langle p^\bullet\rangle\phi, \Delta}\ (\langle\bullet\rangle i)$		

Table 4: Rules for formulae $[p]\phi$ inherited from FODL

4.2.3. Other FOL Rules

Other FOL rules are listed in Table 5. As indicated in Fig. 1, after a QF-AFOL formula is obtained we can adopt SMT-checking procedure to check its validity. Since the SMT-checking procedure is independent from the CDL proof system itself, we propose an *oracle rule* (o) in our proof system to indicate the termination of the proof. We assume that the validity of this QF-AFOL formula can be checked in a "black box" through an SMT-checking procedure. Other rules comes from the traditional FOL and we omit their explanations.

$\dfrac{\models_{cdl} \bigwedge_{\phi\in\Gamma} \phi \to \bigvee_{\varphi\in\Delta} \varphi}{\Gamma \Rightarrow \Delta}\ (o)$ Γ, Δ are multi-sets of QF-AFOL formulae.		
$\dfrac{}{\Gamma, \phi \Rightarrow \phi, \Delta}\ (ax)$	$\dfrac{\Gamma \Rightarrow \phi, \Delta \quad \Gamma, \phi \Rightarrow \Delta}{\Gamma \Rightarrow \Delta}\ (cut)$	
$\dfrac{\Gamma, \neg\phi \Rightarrow \Delta}{\Gamma \Rightarrow \phi, \Delta}\ (\neg r)$	$\dfrac{\Gamma \Rightarrow \neg\phi, \Delta}{\Gamma, \phi \Rightarrow \Delta}\ (\neg l)$	$\dfrac{\Gamma \Rightarrow \phi, \Delta \quad \Gamma \Rightarrow \varphi, \Delta}{\Gamma \Rightarrow \phi \wedge \varphi, \Delta}\ (\wedge r)$
$\dfrac{\Gamma, \phi, \varphi \Rightarrow \Delta}{\Gamma, \phi \wedge \varphi \Rightarrow \Delta}\ (\wedge l)$	$\dfrac{\Gamma \Rightarrow \phi[x'/x], \Delta}{\Gamma \Rightarrow \forall x\phi, \Delta}\ (\forall r)$	$\dfrac{\Gamma, \forall x\phi, \phi[E/x] \Rightarrow \Delta}{\Gamma, \forall x\phi \Rightarrow \Delta}\ (\forall l)$
$\dfrac{\Gamma \Rightarrow \phi, \varphi, \Delta}{\Gamma \Rightarrow \phi \vee \varphi, \Delta}\ (\vee r)$	$\dfrac{\Gamma, \phi \Rightarrow \Delta \quad \Gamma, \varphi \Rightarrow \Delta}{\Gamma, \phi \vee \varphi \Rightarrow \Delta}\ (\vee l)$	$\dfrac{\Gamma, \phi \Rightarrow \varphi, \Delta}{\Gamma \Rightarrow \phi \to \varphi, \Delta}\ (\to r)$
$\dfrac{\Gamma \Rightarrow \phi, \Delta \quad \Gamma, \varphi \Rightarrow \Delta}{\Gamma, \phi \to \varphi \Rightarrow \Delta}\ (\to l)$	$\dfrac{\Gamma \Rightarrow \phi[E/x], \Delta}{\Gamma \Rightarrow \exists x\phi, \Delta}\ (\exists r)$	$\dfrac{\Gamma, \exists x\phi, \phi[x'/x] \Rightarrow \Delta}{\Gamma, \exists x\phi \Rightarrow \Delta}\ (\exists l)$
where x' is new w.r.t. Γ, ϕ and Δ, $\phi[E/x]$ is admissible.		

Table 5: FOL Rules

4.3. Rewrite Rules for Parallel SEPs

For parallel SEPs, generally if no additional information about the interaction between parallel SEPs is provided, the only way for proving the specifications of parallel SEPs is to first reduce a parallel SEP into a sequential one, then apply the rules for sequential SEPs given in Section 4.2. We propose rewrite rules for parallel SEPs, which associate algorithms to provide an automatic mechanism for program reductions in the proof procedure.

(r)	$\dfrac{\phi\{q\}}{\phi\{p\}}$ if $p \rightsquigarrow q$
$(r \parallel \alpha)$	$\parallel(p_1, ..., p_n) \rightsquigarrow \mathit{Inter}(\parallel(p_1, ..., p_n))$ where all $p_i (1 \leq i \leq n)$ are not in the form $q^\bullet$ or $q^\bullet; r$
$(r \parallel \bullet 1)$	$\parallel(p_1, ..., q^\bullet, ..., p_n) \rightsquigarrow \mathit{Exp}(\parallel(p_1, ..., \star(q^\bullet), ..., p_n))$
$(r \parallel \bullet 2)$	$\parallel(p_1, ..., q^\bullet; r, ..., p_n) \rightsquigarrow \mathit{Exp}(\parallel(p_1, ..., \star(q^\bullet); r, ..., p_n))$

Table 6: Rewrite rules for parallel SEPs

4.3.1. Rewrite Rules

The rewrite rules for parallel SEPs are given in Table 6. The rewrite relation $p \rightsquigarrow q$ is read as "p can be reduced to q". The soundness of the rewrite relation means that if $p \rightsquigarrow q$, then $p \equiv q$.

Rule (r) shows how rewrite rules can be applied in the derivation of CDL formulae. It means that if p can be reduced to q, then it does not change the meaning of the formula ϕ by replacing p with q in ϕ. A *program hole* $\phi\{_\}$ is defined as follows:

$$\phi\{_\} ::= [p\{_\}]\xi \mid [p\{_\}]\varphi \mid \neg\phi\{_\} \mid \phi\{_\} \wedge \varphi \mid \varphi \wedge \phi\{_\} \mid \forall x.\phi\{_\},$$

where φ is a CDL formula, $p\{_\}$ is defined as:

$$p\{_\} ::= _ \mid \varrho\&P?p\{_\} \mid q; p\{_\} \mid p\{_\}; q \mid q \cup p\{_\} \mid p\{_\} \cup q \mid (p\{_\})^\bullet \mid \parallel(q_1, ..., p\{_\}, ..., q_n).$$

'_' represents a place that can be filled by a program.

Rule $(r \parallel \alpha)$ transforms a parallel SEP into a sequential one. It requires that in the parallel SEP, all programs $p_1, ..., p_n$ must not be in the forms $q^\bullet$ and $q^\bullet; r$. Procedure *Inter* is for computing the current executable event (Def. 3.15) of the parallel SEP.

Example 4.2. *Given a parallel SEP* $p = \parallel((\varsigma_r; q_1), (\hat{\varsigma_r}?\alpha_1; q_2))$*, we consider a sequent* $\varphi \Rightarrow [p]c_r \prec c_o$*. Since* $\mathit{Inter}(p) = (\varsigma_r|\alpha_1); \parallel(q_1, q_2)$*, so by applying rule* $(r \parallel \alpha)$ *we can make the derivation*

$$\frac{\varphi \Rightarrow [(\varsigma_r|\alpha_1); \parallel(q_1, q_2)]c_r \prec c_o}{\varphi \Rightarrow [p]c_r \prec c_o} \; (r \parallel \alpha, r)$$

where we reduce p *into a sequential program* $(\varsigma_r|\alpha_1); \parallel(q_1, q_2)$*.*

During the proof of a parallel SEP specification, when there exists loop programs of the forms $q^\bullet$ and $q^\bullet; r$ appearing in the parallel SEP, the proof procedure might not terminate if we do not eliminate the loop operator $\bullet$ of programs $q^\bullet$ and $q^\bullet; r$ in the reduction procedure (as shown in Example 4.3 below). Therefore, we propose rules $(r \parallel \bullet 1)$ and $(r \parallel \bullet 2)$ to handle this problem. Rules $(r \parallel \bullet 1)$ and $(r \parallel \bullet 2)$ deal with the situations where some program among the programs $p_1, ..., p_n$ is of the form $q^\bullet$ or $q^\bullet; r$. In these two rules, an expansion procedure (*Exp*) is built to eliminate the loop operator $\bullet$ (tagged by a symbol $\star$). It returns an equivalent sequential SEP which does not contain the loop programs $q^\bullet$ and $q^\bullet; r$.

Example 4.3. *Consider a sequent* $\Gamma \Rightarrow [\|((\varsigma^{c_1})^\bullet, (\varsigma^{c_2})^\bullet)]c_1 \prec c_2$*, since*

$$Inter(\|((\varsigma^{c_1})^\bullet, (\varsigma^{c_2})^\bullet)) = \varsigma^{c_1}; (\varsigma^{c_1})^\bullet \cup \varsigma^{c_2}; (\varsigma^{c_2})^\bullet \cup (\varsigma^{c_1}|\varsigma^{c_2}); \|((\varsigma^{c_1})^\bullet, (\varsigma^{c_2})^\bullet),$$

by applying rule $(r \parallel \alpha)$*, we have the derivation as follows:*

$$\dfrac{\dfrac{\dfrac{\Gamma \Rightarrow [\varsigma^{c_1}; (\varsigma^{c_1})^\bullet]c_1 \prec c_2 \quad \Gamma \Rightarrow [\varsigma^{c_2}; (\varsigma^{c_2})^\bullet]c_1 \prec c_2 \quad \dfrac{\Gamma \Rightarrow [(\varsigma^{c_1}|\varsigma^{c_2})]c_1 \prec c_2 \quad ② \ \Gamma \Rightarrow [\|((\varsigma^{c_1})^\bullet, (\varsigma^{c_2})^\bullet)]c_1 \prec c_2}{\Gamma \Rightarrow [(\varsigma^{c_1}|\varsigma^{c_2}); \|((\varsigma^{c_1})^\bullet, (\varsigma^{c_2})^\bullet)]c_1 \prec c_2}\ (\pi[;], \wedge r)}{\Gamma \Rightarrow [\varsigma^{c_1}; (\varsigma^{c_1})^\bullet \cup \varsigma^{c_2}; (\varsigma^{c_2})^\bullet \cup (\varsigma^{c_1}|\varsigma^{c_2}); \|((\varsigma^{c_1})^\bullet, (\varsigma^{c_2})^\bullet)]c_1 \prec c_2}\ ([\cup])}{① \ \Gamma \Rightarrow [\|((\varsigma^{c_1})^\bullet, (\varsigma^{c_2})^\bullet)]c_1 \prec c_2}\ (r \parallel \alpha, r)$$

where in this proof tree the formulae at node ② and node ① are the same. Thus the proof procedure will never terminate.

4.3.2. Algorithms in Rewrite Rules

The algorithms we present are in pseudo-code style where we freely use abstract variables and data structures such as sets, functions, etc. We use control statements in bold type like **if**...**then**...**else**..., **return**..., **continue**... and so on. We use '$\leftarrow$' to mean the assignment, and '=' to mean the logical judgment that two objects are equivalent or not. "$/*$ ... $*/$" denotes the remarks. For example, in Algorithm 1 below, $p \leftarrow \|(p_1, ..., p_n)$ means that a program $\|(p_1, ..., p_n)$ is assigned to a program variable named p; In Algorithm 4 below, we use $\Upsilon \leftarrow \Upsilon - \{q\}$ to express removing the element q from the set Υ.

The main function of the procedure *Inter* is to transform a parallel SEP into a sequential one. As Algorithm 1 shows, procedure *Inter* is mainly based on the synchronous communication mechanism defined in Def. 3.12. The main idea behind procedure *Inter* is as follows: (i) (line 2) We first apply procedure *Trans* to each sub program of $u_1, ..., u_n$, and reduce them into one of the forms μ, $p \cup q$, $\alpha; q$ or $\varrho\&P?\alpha; q$; (ii) (lines 2 - 3) if there exists a program of the form $p \cup q$ after the transformation, then we transform the program according to the distributive law of $\|$ in Prop. 3.2; (iii) (lines 5 - 8) if all programs are of the form μ, $\alpha; q$ or $\varrho\&P?\alpha; q$, then like what we did in Def. 3.12 we set a as the current observed event of the program p and return the result of procedure *Com*.

Procedures *Trans* and *Com* are given in Algorithm 2 and 3 respectively. The main function of procedure *Trans* is to reduce a program into one of the forms μ, $p \cup q$, $\alpha; q$ or $\varrho\&P?\alpha; q$ by applying the laws in Prop. 3.2. In Algorithm 2, lines 2-3, line 4, line 5 and lines 6-7 correspond to the laws (i), (ii), (iv) and (v)

Algorithm 1 Procedure Inter

1: **procedure** INTER($p = \|(u_1, ..., u_n)$)
2: let $p_1 \leftarrow$ TRANS(u_1), ..., $p_n \leftarrow$ TRANS(u_n), $p \leftarrow \|(p_1, ..., p_n)$
3: **if** there is an r such that $p = \|(p_1, ..., r, ..., p_n)$ and $r = p_{i,1} \cup p_{i,2}$ **then**
4: **return** INTER($\|(p_1, ..., p_{i,1}, ..., p_n)$)$\cup$INTER($\|(p_1, ..., p_{i,2}, ..., p_n)$)
5: **else** /*all $p_i (1 \leq i \leq n)$ are of the form $\alpha; q$ or $\varrho\&P?\alpha; q$*/
6: find all $p_{k_1}, ..., p_{k_m}$ in $p_1, ..., p_n$ such that $p_{k_1} = a_{k_1}; p'_{k_1}, ..., p_{k_m} = a_{k_m}; p'_{k_m}$ $(a_{k_j} \in \{\alpha_{k_j}, P_{k_j}?\alpha_{k_j}\}, 1 \leq j \leq m)$
7: let $a \leftarrow Mer(a_{k_1}, ..., a_{k_m})$
8: **return** COM($\|(p_1, ..., p_n), a$)

Algorithm 2 Procedure Trans

1: **procedure** TRANS(p)
2: **if** $p = \mu; q$ **then return** TRANS(q)
3: **else if** $p = \mu^\bullet$ **then return** μ
4: **else if** $p = (p_1; p_2); p_3$ **then return** TRANS($p_1; p_2; p_3$)
5: **else if** $p = (p_1 \cup p_2); p_3$ **then return** $p_1; p_3 \cup p_2; p_3$
6: **else if** $p = q^\bullet$ (or $p = q^\bullet; r$) **then**
7: **return** $\mu \cup q; q^\bullet$ (or $r \cup q; q^\bullet; r$)
8: **else if** $p = \|(p_1, ..., p_n)$ (or $p = \|(p_1, ..., p_n); q$) **then**
9: $p \leftarrow$ INTER($\|(p_1, ..., p_n)$), **return** p (or TRANS($p; q$))
10: **else** /*p is of the form α, μ, $\alpha; q$ or $\varrho\&P?\alpha; q$*/
11: **return** p

Algorithm 3 Procedure Com

1: **procedure** $\textsc{Com}(\|(p_1, ..., p_n), a)$
2: **if** there is an $r = p_i$ $(1 \le i \le n)$ in $p_1, ..., p_n$ such that $r = \mu$ **then**
3: **return** $\textsc{Com}(\|(p_1, ..., p_{i-1}, p_{i+1}, ..., p_n), a)$ /*remove r from $p_1, ..., p_n$*/
4: **else if** there exist $r_1, ..., r_m$ in $p_1, ..., p_n$ such that $r_k = \varrho_k \& P_k ? \beta_k; q_k$ $(1 \le k \le m)$ **then**
5: **if** $Mat(a, \varrho_k)$ holds for all $1 \le k \le m$ **then**
6: **for** each k **do** let $r'_k \leftarrow P_k ? Sub(\beta_k, a, \varrho_k); Sub(q_k, a, \varrho_k)$
7: let $a' \leftarrow Mer(P_1 ? Sub(\beta_1, a, \varrho_1), ..., P_m ? Sub(\beta_m, a, \varrho_m), a)$
8: **return** $\textsc{Com}(\|(p_1, ..., r'_1, ..., r'_m, ..., p_n), a')$
9: **else** /*there exists $1 \le k \le m$ such that $\neg Mat(a, \varrho_k)$ holds*/
10: **return** $\ddagger$ /*the program reaches a deadlock*/
11: **else** /*for all $p_1, ..., p_n$, $p_1 = a_1; p'_1, ..., p_n = a_n; p'_n$ $(a_j \in \{\alpha_j, P_j ? \alpha_j\}, 1 \le i \le n)$*/
12: **return** $a; \|(p'_1, ..., p'_n)$

of Prop. 3.2 respectively. If the program is of the form $\|(p_1, ..., p_n)$ or $\|(p_1, ..., p_n); q$ (lines 8-9), then we need to call another procedure *Inter* to further deal with the parallel part.

Procedure *Com* in Algorithm 3 is just an implementation of the function *Com* defined in Def. 3.12. Lines 2-3, lines 5-8, lines 9-10 and lines 11-12 correspond to the conditions (i), (ii)(1), (ii)(2) and (iii) in Def. 3.12 respectively. The main difference is when the program reaches a deadlock, procedure *Com* returns the miracle program $\ddagger$, whose semantics corresponds to an empty set $\emptyset$.

Example 4.4. *Consider the digital filter program DF in Example 3.1, write Fil and Fee as:* $Fil = (\varsigma_r; p_1)^\bullet; \ddagger$ *and* $Fee = (\hat{\varsigma_r} ? \alpha_1; q_1)^\bullet; \ddagger$*, where* $p_1 = \hat{\varsigma_p} ? \epsilon; \hat{\varsigma_p} ? \epsilon; \hat{\varsigma_e} ? \varsigma_o$*,* $q_1 = (P_1 ? \alpha_2 \cup P_2 \alpha_3)^\bullet; f = 1 ? \varsigma_e$*. The execution procedure of Inter(DF) is shown below:*

$$
\begin{aligned}
Inter : DF = \ & \|((\varsigma_r; p_1)^\bullet; \ddagger, (\hat{\varsigma_r} ? \alpha_1; q_1)^\bullet; \ddagger) \\
\longrightarrow_{l2,I} \ & \|(Trans((\varsigma_r; p_1)^\bullet; \ddagger), Trans((\hat{\varsigma_r} ? \alpha_1; q_1)^\bullet; \ddagger)) \\
\longrightarrow_{l6-7,T} \ & \|(\ddagger \cup (\varsigma_r; p_1); (\varsigma_r; p_1)^\bullet; \ddagger, \ddagger \cup (\hat{\varsigma_r} ? \alpha_1; q_1); (\hat{\varsigma_r} ? \alpha_1; q_1)^\bullet; \ddagger) \\
\longrightarrow_{l4,I} \ & Inter(\|(\ddagger, \ddagger)) \cup Inter(\|(\ddagger, (\hat{\varsigma_r} ? \alpha_1; q_1); (\hat{\varsigma_r} ? \alpha_1; q_1)^\bullet; \ddagger)) \cup \qquad (1) \\
& Inter(\|((\varsigma_r; p_1); (\varsigma_r; p_1)^\bullet; \ddagger, \ddagger)) \cup \\
& Inter(\|((\varsigma_r; p_1); (\varsigma_r; p_1)^\bullet; \ddagger, (\hat{\varsigma_r} ? \alpha_1; q_1); (\hat{\varsigma_r} ? \alpha_1; q_1)^\bullet; \ddagger)) \\
\longrightarrow ... \longrightarrow \ & H(\ddagger) \cup a'; \|(p_1; (\varsigma_r; p_1)^\bullet; \ddagger, q_1; (\hat{\varsigma_r} ? \alpha_1; q_1)^\bullet; \ddagger) \qquad (2)
\end{aligned}
$$

where the execution procedure of $Inter(\|((\varsigma_r; p_1); (\varsigma_r; p_1)^\bullet; \ddagger, (\hat{\varsigma_r} ? \alpha_1; q_1); (\hat{\varsigma_r} ? \alpha_1; q_1)^\bullet; \ddagger))$ *is as follows:*

$$
\begin{aligned}
Inter : \ & \|((\varsigma_r; p_1); (\varsigma_r; p_1)^\bullet; \ddagger, (\hat{\varsigma_r} ? \alpha_1; q_1); (\hat{\varsigma_r} ? \alpha_1; q_1)^\bullet; \ddagger) \\
\longrightarrow_{l2,I} \ & \|(Trans((\varsigma_r; p_1); (\varsigma_r; p_1)^\bullet; \ddagger), Trans((\hat{\varsigma_r} ? \alpha_1; q_1); (\hat{\varsigma_r} ? \alpha_1; q_1)^\bullet; \ddagger)) \\
\longrightarrow_{l4,T} \ & \|(\varsigma_r; p_1; (\varsigma_r; p_1)^\bullet; \ddagger, \hat{\varsigma_r} ? \alpha_1; q_1; (\hat{\varsigma_r} ? \alpha_1; q_1)^\bullet; \ddagger) \\
\longrightarrow_{l5-8,I} \ & Com(\|(\varsigma_r; p_1; (\varsigma_r; p_1)^\bullet; \ddagger, \hat{\varsigma_r} ? \alpha_1; q_1; (\hat{\varsigma_r} ? \alpha_1; q_1)^\bullet; \ddagger), a) \qquad (3) \\
\longrightarrow_{l5-8,C} \ & Com(\|(\varsigma_r; p_1; (\varsigma_r; p_1)^\bullet; \ddagger, \alpha_1; q_1; (\hat{\varsigma_r} ? \alpha_1; q_1)^\bullet; \ddagger), a') \qquad (4) \\
\longrightarrow_{l11-12,C} \ & a'; \|(p_1; (\varsigma_r; p_1)^\bullet; \ddagger, q_1; (\hat{\varsigma_r} ? \alpha_1; q_1)^\bullet; \ddagger) \\
\longrightarrow \ & a'; \|(p_1; (\varsigma_r; p_1)^\bullet; \ddagger, q_1; (\hat{\varsigma_r} ? \alpha_1; q_1)^\bullet; \ddagger)
\end{aligned}
$$

The flow $F : X_1 \longrightarrow X_2 \longrightarrow ... \longrightarrow X_n$ *explains how a data* X_1 *evolves as the procedure* F *executes.* $X \longrightarrow_{lx-y,F} Y$ $(F \in \{I, T, C\})$ *means that from* X*, by executing the code from line* x *to line* y *of the procedure* F*, we obtain* Y*. The symbols* I, T, C *stand for procedures Inter, Trans and Com respectively. For example, from* (3)*, by lines 5-8 of procedure Com, we obtain* (4)*, where* $a = \varsigma_r$ *and* $a' = (\varsigma_r | \alpha_1)$*. At* (1) *we see that the program DF is split into 4 parts, linked by* $\cup$*. From* (1)*, after several steps of executing Inter recursively, we obtain* (2) *as the final result. Here we only focus on the detailed procedure of the last part* $Inter(\|((\varsigma_r; p_1); (\varsigma_r; p_1)^\bullet; \ddagger, (\hat{\varsigma_r} ? \alpha_1; q_1); (\hat{\varsigma_r} ? \alpha_1; q_1)^\bullet; \ddagger))$*, since other 3 parts:* $Inter(\|(\ddagger, \ddagger))$*,* $Inter(\|(\ddagger, (\hat{\varsigma_r} ? \alpha_1; q_1); (\hat{\varsigma_r} ? \alpha_1; q_1)^\bullet; \ddagger))$ *and* $Inter(\|((\varsigma_r; p_1); (\varsigma_r; p_1)^\bullet; \ddagger, \ddagger))$ *return miracle programs as they all*

Algorithm 4 Procedure Exp

1: **procedure** Exp(p) /*$p = \|(q_1, ..., p_i = \star(q^\bullet); r, ..., q_n)$*/
2: $\Xi \leftarrow \emptyset$ /*programs already treated*/
3: $\Upsilon \leftarrow p$ /*programs remained to be treated*/
4: $\mathcal{E} \leftarrow \emptyset$ /*the set of equations*/
5: **while** $\Upsilon \neq \emptyset$ **do**
6: choose a program $q \in \Upsilon$, $\Upsilon \leftarrow \Upsilon - \{q\}$, $\Xi \leftarrow \Xi \cup \{q\}$
7: $q' \leftarrow$ Inter(q) /*$q' = t_1 \cup ... \cup t_n$ $(n \geq 1)$, each t_i is of the form a or $a; q''$, where $a \in \{\alpha, \varrho\&P?\alpha\}$*/
8: $\mathcal{E} \leftarrow \mathcal{E} \cup \{q \equiv q'\}$ /*build equation and add it to $\mathcal{E}$*/
9: **for** each t_i in $q' = t_1 \cup ... \cup t_n$ $(1 \leq i \leq n)$ **do**
10: **if** $t_i = a; q'' \wedge Tag(q'') \wedge q'' \notin \Xi$ **then**
11: $\Upsilon \leftarrow \Upsilon \cup \{q''\}$ /*further reduce other programs not treated before*/
12: **return** Solv($\mathcal{E}$)

begin with a program of the form ff?α. For example, for the first part, we have Inter$(\ddagger, \ddagger) = \ddagger$. Generally, we denote any miracle program as $H(\ddagger)$.

Procedure *Exp* (in Algorithm4) is mainly based on the Brzozowski's method for transforming a regular language into an equivalent finite automaton [33]. It continuously reduces the parallel program until a set of equations is formed. By solving this equations using Arden's rule (Prop. 4.1 below), procedure *Exp* returns the sequential program in which the target loop operator (tagged by $\star$) has been eliminated.

The main idea behind *Exp* is as follows: (i) (lines 2-4) We initialize the sets Ξ, Υ and $\mathcal{E}$; (ii) (lines 6-8) we reduce the current program q into the form $q' = t_1 \cup ... \cup t_n$ by procedure *Inter*, and we build an equation $q \equiv q'$; (iii) (lines 9-11) for any t_i of the form $a; q''$, if q'' still contains the target loop operator inside and has not been reduced yet, we further reduce q''; (iv) (line 12) we solve the set of equations $\mathcal{E}$ in procedure *Solv*. The function *Tag*(p) returns *true* if the target loop program is in p.

Example 4.5. *Following Example 4.4, we consider the execution procedure of Exp(DF) (where DF = $\|(\star(\varsigma_r; p_1)^\bullet; \ddagger, (\hat{\varsigma_r}?\alpha_1; q_1)^\bullet; \ddagger)$) as follows:*

$$
\begin{aligned}
& Exp : \emptyset \\
\longrightarrow_{l5-11,E}\ & w_1 \equiv H_1(\ddagger) \cup (\varsigma_r|\alpha_1); w_2 \\
\longrightarrow_{l5-11,E}\ & w_2 \equiv H_2(\ddagger) \cup P_1?\alpha_2; w_3 \cup P_2?\alpha_3; w_3 \\
\longrightarrow_{l5-11,E}\ & w_3 \equiv \ddagger \cup P_1?\alpha_2; w_4 \cup P_2?\alpha_3; w_4 \\
\longrightarrow_{l5-11,E}\ & w_4 \equiv H_3(\ddagger) \cup f = 1?(\varsigma_o|\varsigma_e); w_1
\end{aligned}
$$

where programs w_1, w_2, w_3, w_4 are listed in the following table:

$w_1 = DF$	$w_2 = \|(p_1; (\varsigma_r; p_1)^\bullet; \ddagger, q_1; (\hat{\varsigma_r}?\alpha_1; q_1)^\bullet; \ddagger)$
$w_3 = \|(\varsigma_p?\epsilon; \hat{\varsigma_e}?\varsigma_o; (\varsigma_r; p_1)^\bullet; \ddagger, q_1; (\hat{\varsigma_r}?\alpha_1; q_1)^\bullet; \ddagger)$	$w_4 = \|(\hat{\varsigma_e}?\varsigma_o; (\varsigma_r; p_1)^\bullet; \ddagger, q_1; (\hat{\varsigma_r}?\alpha_1; q_1)^\bullet; \ddagger)$

In the flow above, at each step the data represents the equation Exp generated at the current loop. The symbol E stands for Exp. For example, at the first step, by executing lines 5-11 of procedure Exp, we obtain the equation $w_1 \equiv H_1(\ddagger) \cup (\varsigma_r|\alpha_1); w_2$. The program $H_1(\ddagger) \cup (\varsigma_r|\alpha_1); w_2$ is just the result returned by Inter(DF) in Example 4.4. The explicit structure of the miracle programs $H_1(\ddagger)$, $H_2(\ddagger)$ and $H_3(\ddagger)$ are omitted because their semantics are just equivalent to $\ddagger$.

Procedure *Solv* describes the process of solving the equations obtained from procedure *Exp*. It turns out to be a standard procedure for solving the algebra equations using Arden's rule [34]. In SEP, Arden's rule can be stated as the following proposition.

Proposition 4.1 (Arden's Rule in SEP)**.** *Given any SEPs p and q with $q \not\equiv \mu$,*

$$X \equiv q^\bullet; p \text{ is the only solution of } X \equiv p \cup q; X.$$

Algorithm 5 Procedure *Solv*

1: **procedure** SOLV($\mathcal{E}$)

2: equations in $\mathcal{E}$ are listed below, where p_{ij} $(1 \leq i, j \leq n)$ is of the form $a_1 \cup ... \cup a_o$, $a_k \in \{\alpha, P?\alpha\}$ $(1 \leq k \leq o)$:

$$w_1 \equiv \gamma_1 \cup p_{11}; w_1 \cup p_{12}; w_2 \cup ... \cup p_{1n}; w_n \quad (1)$$
$$w_2 \equiv \gamma_2 \cup p_{21}; w_1 \cup p_{22}; w_2 \cup ... \cup p_{2n}; w_n \quad (2)$$
$$...$$
$$w_n \equiv \gamma_n \cup p_{n1}; w_1 \cup p_{n2}; w_2 \cup ... \cup p_{nn}; w_n \quad (n)$$

3: **for** each k, $k = n, n-1, ..., 2, 1$ **do**

4: substitute $w_{k+1}, ..., w_n$ in equation (k) and transform it into the form $w_k \equiv p \cup q; w_k$.

5: apply Arden's rule, eliminate w_k on the right side of $w_k \equiv p \cup q; w_k$ and get $w_k \equiv q^\bullet; p$.

6: **return** w_1

Proof Sketch. The proof of Prop. 4.1 is quite straight forward because according to the syntax of SEPs (Def. 3.1), SEP is in fact a sub-language of an *omega language* whose syntax is just the same as the syntax of SEPs except that we define p^* and p^ω instead of $p^\bullet$. p^* is the finite loop program as we saw in FODL while the infinite word p^ω represents the program that executes p for infinite times. With p^* and p^ω, the infinite loop program $p^\bullet$ in SEP can be expressed as: $p^\bullet = p^* \cup p^\omega$. The omega language follows the theory of the omega algebra proposed in [35], according to which it is not hard to see that the programs $q^*; p$ and $(q^* \cup q^\omega); p$ are the least and greatest solutions of the equation $X \equiv p \cup q; X$. Thus except for $q^\bullet; p$ (note that $q^\bullet \equiv q^* \cup q^\omega$), we cannot find other solutions between $q^*; p$ and $(q^* \cup q^\omega); p$ in SEPs. □

Example 4.6. *Continuing the Example 4.5, the flow of solving the equations obtained in Exp(DF) is described as follows:*

$$\begin{aligned}
& Solv : \emptyset, k = 5 \\
\longrightarrow_{l4-5,S}\ & w_4 \equiv H_3 \cup u_1; w_1, k = 4 && (1)\\
\longrightarrow_{l4-5,S}\ & w_3 \equiv \ddagger \cup (a_1 \cup a_2); H_3 \cup (a_1 \cup a_2); u_1; w_1, k = 3 && (2)\\
\longrightarrow_{l4-5,S}\ & w_2 \equiv H_4 \cup (a_1 \cup a_2); (a_1 \cup a_2); u_1; w_1, k = 1 \\
\longrightarrow_{l4-5,S}\ & w_1 \equiv H_1 \cup u_2; H_4 \cup u_2; (a_1 \cup a_2); (a_1 \cup a_2); u_1; w_1, k = 0 && (3)\\
\longrightarrow_{l4-5,S}\ & w_1 \equiv (u_2; (a_1 \cup a_2); (a_1 \cup a_2); u_1)^\bullet; (H_1 \cup u_2; H_4), k = 0 && (4)
\end{aligned}$$

where $a_1 = P_1?\alpha_2$, $a_2 = P_2?\alpha_3$, $u_1 = (f = 1)?(\varsigma_o|\varsigma_e)$, $u_2 = (\varsigma_r|\alpha_1)$, $H_4 = H_2 \cup (a_1 \cup a_2); \ddagger \cup (a_1 \cup a_2); (a_1 \cup a_2); H_3$.

The flow shows the value of k *and the equation dealt with in each loop on lines* $3-6$ *in Solv. The symbol* S *stands for Solv. For example, from (1) to (2), after executing lines 4-5 of Solv, we obtain the equation* $w_3 \equiv ...$ *by substituting* w_4 *with* $H_3 \cup u_1; w_1$. *From (3) to (4), we obtain the final program* w_1 *by applying Arden's rule (Prop. 4.1) to (3). Note that* $H_1 \cup u_2; H_4$ *is a miracle program, since* $H_1 \cup u_2; H_4 \equiv \ddagger$. *The program can be further simplified before returned, and we finally get*

$$Exp(DF) = (u_2; (a_1 \cup a_2); (a_1 \cup a_2); u_1)^\bullet; \ddagger.$$

4.4. An Example

At the end of this section, we give a complete example of verifying a CCSL specification of a synchronous system in CDL. In the digital filter system given in Example 3.1, consider a CCSL specification expressed in a CDL formula as follows:

$$I \rightarrow [DF]c_r \prec c_o.$$

It means that if I holds, then all executing traces of the program DF satisfy the clock relation $c_r \prec c_o$. I is the initial condition of clock-related variables of the program, let

$$I = \bigwedge_{c \in \{c_e, c_p, c_r, c_o\}} (c^n = 0 \wedge c^s = 0).$$

This specification is illustrated as the red arrows in Fig. 6, which means only after the "Ready" message has been sent, the output can be generated through port o. It is a simple unbounded CCSL specification, which captures an important safety property in synchronous systems.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\bigwedge_{\varphi\in\Gamma_2}\varphi\to\hbar(c_r\prec c_o)}{\Gamma_2\Rightarrow\hbar(c_r\prec c_o)}\,(o)}{\varphi_1\Rightarrow[(\varsigma_r|\alpha_1)]c_r\prec c_o}\,(\pi[])\quad \dfrac{\dfrac{\dfrac{\dfrac{\

encode them as program observers since they can generate new clocks while observing existing clocks from the systems [36, 37]. Fig. 15 shows clock definition $c \triangleq c_1 + c_2$, as a program observer, interacts with the system program: at each instant, it observes clocks c_1 and c_2 from the system program, and triggers clock c if either clock c_1 or clock c_2 ticks. In this section, we mainly propose an encoding of clock definitions as SEPs.

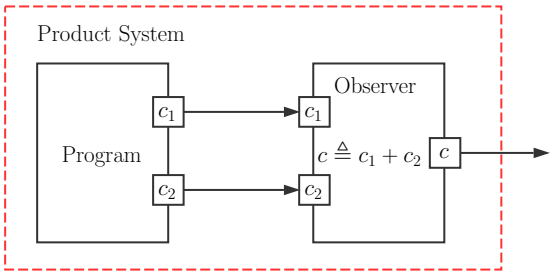

Figure 15: Clock definition $c \triangleq c_1 + c_2$ as a program observer

CDL provides a natural way to express clock definitions as SEPs since it contains signals as primitives. Table 7 lists the encoding for each clock definition *cdf*, denoted as cdf^E. Generally, the program observer has the form:

$$\alpha_1; p^{\bullet}; \alpha_2,$$

where α_1 is an initial event that initializes local variables used in the observer, p is the main loop body responsible for modeling the semantics of the clock definition, α_2 is called a *final event* of the program.

The syntactic sugar $p_{wait} = wait\ \varrho_1 \& P_1\ do\ (\alpha_1; p_1)\ ::\ \varrho_2 \& P_2\ do\ (\alpha_2; p_1)\ ::\ ...\ ::\ \varrho_n \& P_n\ do\ (\alpha_n; p_n)$ is defined as

$$p_{wait} =_{df} (\varrho ? \epsilon)^{\bullet}; (\varrho_1 \& P_1 ? \alpha_1; p_1 \cup \varrho_2 \& P_2 ? \alpha_2; p_2 \cup ... \cup \varrho_n \& P_n ? \alpha_n; p_n),$$

where $\varrho =_{df} Neg(\varrho_1 \veebar ... \veebar \varrho_n)$, the function $Neg(\varrho)$ returns the negation of the condition ϱ, which is given as:

(1) $Neg(\hat{\varsigma}) =_{df} \bar{\varsigma}$, $Neg(\bar{\varsigma}) =_{df} \hat{\varsigma}$;

(2) $Neg(\varrho_1 \wedge \varrho_2) =_{df} Neg(\varrho_1) \barwedge Neg(\varrho_2)$;

(3) $Neg(\varrho_1 \vee \varrho_2) =_{df} Neg(\varrho_1) \veebar Neg(\varrho_2)$.

The program p_{wait} says that at each instant, it checks whether one of the conditions $\varrho_1 \& P_1$,...,$\varrho_n \& P_n$ holds and executes the corresponding branches $\alpha_1; p_1$,...,$\alpha_n; p_n$. If none of these conditions hold, then the program simply waits for a unit of time and continues the judgement at the next instant. The syntactic sugar $p_{ite} = if\ \varrho_1 \& P_1\ then\ (\alpha_1; p_1)\ else\ (\beta; q)$ is defined as

$$p_{ite} =_{df} \varrho_1 \& P_1 ? \alpha_1; p_1 \cup P ? \beta; q,$$

where $P = \neg(P_1 \vee ... \vee P_n)$. It says that if the condition $\varrho_1 \& P_1$ holds, then $\alpha_1; p_1$ proceeds, otherwise $\beta; q$ proceeds.

As shown in Table 7, the program of Union (resp. Intersection) checks if the signal ς^{c_1} or (resp. and) ς^{c_2} emits (resp. emit) at each instant, if so, it emits the signal ς^{c}, otherwise it waits for a unit of time. The program of Strict Sample first waits for the signal ς^{c_1}, then it produces ς^{c} while the signal ς^{c_2} emits. The program of Sample behaves as Strict Sample, except that if the signals ς^{c_1} and ς^{c_2} emit simultaneously, the signal ς^{c} emits immediately. In the program of Interruption we need a local variable x to indicate when the loop ends. When the signal ς^{c_1} emits and the signal ς^{c_2} does not, the program emits ς^{c}. Once the signal ς^{c_2} emits, x is set to 1 and the loop ends. In the program of Periodicity, the variable x is used to record the

1.	$(c \triangleq c_1 + c_2)^E =_{df} \epsilon; (wait\ \varsigma^{\hat{c}_1} \veebar \varsigma^{\hat{c}_2}\ do\ \varsigma^c)^\bullet; \ddagger$	(Union)
2.	$(c \triangleq c_1 * c_2)^E =_{df} \epsilon; (wait\ \varsigma^{\hat{c}_1} \barwedge \varsigma^{\hat{c}_2}\ do\ \varsigma^c)^\bullet; \ddagger$	(Intersection)
3.	$(c \triangleq c_1 \blacktriangleright c_2)^E =_{df} \epsilon; (wait\ \varsigma^{\hat{c}_1}\ do\ \epsilon; wait\ \varsigma^{\hat{c}_2}\ do\ \varsigma^c)^\bullet; \ddagger$	(Strict Sample)
4.	$(c \triangleq c_1 \rhd c_2)^E =_{df} \epsilon; (wait\ \varsigma^{\hat{c}_1} \barwedge \varsigma^{\bar{c}_2}\ do\ q\ ::\ \varsigma^{\hat{c}_1} \barwedge \varsigma^{\hat{c}_2}\ do\ \varsigma^c)^\bullet; \ddagger$ where $q = \epsilon; wait\ \varsigma^{\hat{c}_2}\ do\ \varsigma^c$	(Sample)
5.	$(c \triangleq c_1 \curvearrowright c_2)^E =_{df} x := 0; (wait\ \varsigma^{\hat{c}_1} \barwedge \varsigma^{\bar{c}_2} \& x = 0\ do\ \varsigma^c\ ::\ \varsigma^{\hat{c}_2} \& x = 0\ do\ x := 1)^\bullet; x = 1?\epsilon$	(Interruption)
6.	$(c \triangleq c' \propto n)^E =_{df} x := n; (wait\ \varsigma^{\hat{c'}} \& x = 0\ do\ (\varsigma^c \vert x := n)\ ::\ \varsigma^{\hat{c'}} \& x > 0\ do\ x := x - 1)^\bullet; \ddagger$	(Periodicity)
7.	$(c \triangleq c' \$ n)^E =_{df} x := n; (wait\ \varsigma^{\hat{c'}} \& x = 0\ do\ \varsigma^c\ ::\ \varsigma^{\hat{c'}} \& x > 0\ do\ x := x - 1)^\bullet; \ddagger$	(Delay)
8.	$(c \triangleq c_1 \wedge c_2)^E =_{df} x := 0; (wait\ \varsigma^{\hat{c}_1} \barwedge \varsigma^{\bar{c}_2}\ do\ q_1\ ::\ \varsigma^{\hat{c}_1} \barwedge \varsigma^{\hat{c}_2}\ do\ \varsigma^c\ ::\ \varsigma^{\bar{c}_1} \barwedge \varsigma^{\hat{c}_2}\ do\ q_2))^\bullet; \ddagger$ where $q_1 = if\ x \geq 0\ then\ (\varsigma^c \vert x := x + 1)\ else\ x := x + 1,$ $q_2 = if\ x \leq 0\ then\ (\varsigma^c \vert x := x - 1)\ else\ x := x - 1$	(Infimum)
9.	$(c \triangleq c_1 \vee c_2)^E =_{df} x := 0; (wait\ \varsigma^{\hat{c}_1} \barwedge \varsigma^{\bar{c}_2}\ do\ q_1\ ::\ \varsigma^{\hat{c}_1} \barwedge \varsigma^{\hat{c}_2}\ do\ \varsigma^c\ ::\ \varsigma^{\bar{c}_1} \barwedge \varsigma^{\hat{c}_2}\ do\ q_2)^\bullet; \ddagger$ where $q_1 = if\ x \leq 0\ then\ (\varsigma^c \vert x := x + 1)\ else\ x := x + 1,$ $q_2 = if\ x \geq 0\ then\ (\varsigma^c \vert x := x - 1)\ else\ x := x - 1$	(Supremum)

Table 7: Encoding of clock definitions in CDL

remaining times of the occurrences of $\varsigma^{c'}$ before the occurrence of ς^c. It is initialized by n. At each instant when the signal $\varsigma^{c'}$ emits, x is decreased by 1. When $x = 0$, the program emits c. In the program of Delay, x is used to keep the remaining times of the occurrences of $\varsigma^{c'}$ before an occurrence of ς^c. At the beginning when the signal $\varsigma^{c'}$ emits, x is decreased by 1. When $x = 0$, ς^c emits whenever $\varsigma^{c'}$ emits. The program of Infimum and the program of Supremum are symmetrical. In the program of Infimum, we need a local variable x to record the difference between the number of the occurrences of ς^{c_1} and ς^{c_2}. At each instant the signal ς^c emits iff the signal that occurs slower emits. For example, if the number of the occurrences of ς^{c_1} is more than that of ς^{c_2} (i.e., $x \geq 0$), then ς^c emits in q_2 when ς^{c_2} emits. The situation for the program of Supremum is similar. We omit the details here.

At last, as indicated in Fig. 1, we show that any general CCSL specification can be fully captured by a CDL formula of the form $I \wedge I_c \rightarrow [\|(p, q_1, ..., q_n)]\xi$.

Definition 5.1 (Representation of CCSL specifications in CDL)**.** *Given an SEP p and a CCSL specification $\langle \mathcal{C}, \mathbf{Cdf}, \mathbf{Rel} \rangle$ where $\mathbf{Cdf} = \{cdf_1, ..., cdf_m\}$, $\mathbf{Rel} = \{Rel_1, ..., Rel_n\}$, then the verification problem of whether p satisfies this specification can be expressed as a CDL formula:*

$$\phi = I \wedge I_c \rightarrow [\|(\epsilon; p, cdf_1^E, ..., cdf_m^E)]\ \curlywedge\ (Rel_1, ..., Rel_n),$$

where $I = \bigwedge_{c \in \mathcal{C}(p)} (c^n = 0 \wedge c^s = 0)$ is an initial condition of the clock-related variables of the program p, $I_c = \bigwedge_{c \in \mathcal{C}} (c^n = 0 \wedge c^s = 0)$ is an initial condition of the clock-related variables of the specification $\langle \mathcal{C}, \mathbf{Cdf}, \mathbf{Rel} \rangle$. Because of the initial event in cdf_i^E, we add an idle event ϵ before p.

The initial condition $I \wedge I_c$ restricts us to only consider the satisfaction of **Rel** by all initial traces of $\|(\epsilon; p, cdf_1^E, ..., cdf_m^E)$. The proof of the formula ϕ reflects the satisfaction of the CCSL specification by all behaviours of p. Note that in Def. 5.1, actually $I \wedge I_c = \bigwedge_{c \in \mathcal{C}(p) \cup \mathcal{C}} (c^n = 0 \wedge c^s = 0)$.

Example 5.1. *Consider another specification of the program DF in Example 3.1:*

$$\langle \mathcal{C}, \{c_1 \triangleq c_r \,\$\, 1, c_2 \triangleq c_p \propto 1\}, \{c_2 \prec c_1\}\rangle,$$

where $\mathcal{C} = \{c_p, c_r, c_1, c_2\}$, c_1 *and* c_2 *are new clocks generated by the clock definitions. It means that only after receiving two pixels, the new "Ready" message is sent (as indicated as blue arrows in Fig. 6). In CDL, this specification can be encoded as the following formula:*

$$\bigwedge_{c \in \mathcal{C} \cup \mathcal{C}(DF)} (c^n = 0 \wedge c^s = 0) \rightarrow [\|(\epsilon; DF, (c_1 \triangleq c_r \,\$\, 1)^E, (c_2 \triangleq c_p \propto 1)^E]c_1 \prec c_2,$$

where $\mathcal{C}(DF) = \{c_e, c_o, c_p, c_r\}$.

6. Soundness and Relative Completeness of CDL

In this section we analyze the soundness and relative completeness of the proof system of CDL. As an extension of FODL, except for the rules for the special primitives in CDL, other rules are mainly inherited from FODL and traditional FOL. We mainly focus on the soundness of those rules special in CDL. The proof of the relative completeness of CDL is mainly inspired from that of FODL in [20, 25], where a main theorem forms the proof skeleton, while the conditions of the main theorem describe the technical details of the proof. In the proof of CDL we propose a similar main theorem (Theorem 6.2) with a set of modified conditions (the conditions (i) - (iv) in Theorem 6.2). In the following we mainly analyze the critical points of the proof that are different from that of FODL, and we leave the details in Appendix A.

Before the analysis we first introduce the notion of *deductive relation* in CDL. For a CDL formula ϕ and a multi-set Φ of formulae, $\Phi \vdash_{cdl} \phi$ iff the sequent $\Phi \Rightarrow \phi$ can be derived using rules in Table 3, 4, 5 and 6. If Φ is empty, we also write $\vdash_{cdl} \phi$.

6.1. Soundness

The soundness of the proof system of CDL is stated as the following theorem.

Theorem 6.1 (Soundness of $\vdash_{cdl}$)**.** *Given any CDL formula* ϕ*, if* $\vdash_{cdl} \phi$*, then* $\models_{cdl} \phi$*.*

To prove this theorem, we need to show that each rule $\dfrac{\Gamma_1 \Rightarrow \Delta_1 \quad ... \quad \Gamma_n \Rightarrow \Delta_n}{\Gamma \Rightarrow \Delta}$ in the proof system is sound in the sense that:

$$\text{if } \models_{cdl} \bigwedge_{\varphi_1 \in \Gamma_1} \varphi_1 \rightarrow \bigvee_{\varphi_2 \in \Delta_1} \varphi_2, ..., \models_{cdl} \bigwedge_{\varphi_1 \in \Gamma_n} \varphi_1 \rightarrow \bigvee_{\varphi_2 \in \Delta_n} \varphi_2, \text{ then}$$
$$\models_{cdl} \bigwedge_{\varphi_1 \in \Gamma} \varphi_1 \rightarrow \bigvee_{\varphi_2 \in \Delta} \varphi_2.$$

For the rules for sequential SEPs, in Section 4.2 we have given an intuitive meaning for each rule and their relations to the corresponding rules in FODL. The soundness of all FOL rules in Table 5 is straightforward. The key to prove rules $(\pi[;])$, $(\pi[\cup])$, $(\pi[\bullet]u)$ and $(\pi[\bullet]i)$ is to prove a temporal property possessed by clock relations (see Prop. A.1 in Appendix A.1). The proofs of rules $(\pi[\bullet]u)$, $(\pi[\bullet]i)$, $([\bullet]u)$, $([\bullet]ind)$, $(\langle\bullet\rangle con)$, $([\bullet]i)$ and $(\langle\bullet\rangle i)$ are inspired from their counterparts in FODL. As we have discussed in Section 4.2, since we consider only partial correctness of the program, $[p^\bullet]\phi$ actually means the same thing as $[p^*]\phi$ in FODL (see Prop. A.2 in Appendix A.2). The soundness of other rules can be directly obtained by the semantics of CDL, some of their proofs are given in Appendix A.1.

In the rules of parallel SEPs, rule (r) can be proved by induction on the structure of the program hole $p\{_\}$. The correctness of the program reductions in procedure *Inter* can be directly proved according to the semantics of parallel SEPs (Def. 3.12 and 3.14) and the equivalence relations between programs (Prop. 3.2). As for *Exp*, to prove $p \equiv Exp(p)$ we only need to prove Arden's rule in SEPs (Prop.4.1), which is straight forward as analyzed in Section 4.3.

6.2. Relative Completeness

As CDL contains AFOL in itself, which is generally incomplete due to Gödel's incompleteness theorem [38], we consider the completeness of CDL relative to AFOL. In the analysis of the relative completeness we assume that the proof system contains all tautologies in the traditional AFOL. We use $\Phi \vdash^{+}_{cdl} \phi$ to mean that $\Phi \Rightarrow \phi$ can be derived in the system that consists of all rules in Table 3, 4, 5 and 6, and all tautologies in AFOL as axioms.

The proof skeleton of the relative completeness of CDL is stated as the following theorem. It mainly follows the main theorem of [25], but is extended with the condition (iv), which describes a similar property as the condition (ii) for formulae of the forms $[p]\xi$ and $\langle p\rangle \sim \xi$.

We sometimes write $\varphi^\flat$ to stress that φ is an AFOL formula.

Theorem 6.2 (Relative Completeness of $\vdash_{cdl}$)**.** *For any formula ϕ, if $\models_{cdl} \phi$, then $\vdash^{+}_{cdl} \phi$ holds if the following conditions are true:*

(i) For any CDL formula ϕ, ϕ is expressible in AFOL.

(ii) For any AFOL formulae $\varphi^\flat$ and $\phi^\flat$, if $\models_{cdl} \varphi^\flat \to op\,\phi^\flat$, then $\vdash^{+}_{cdl} \varphi^\flat \to op\,\phi^\flat$.

(iii) For any formulae φ and ϕ, if $\vdash^{+}_{cdl} \varphi \to \phi$ then $\vdash^{+}_{cdl} op\,\varphi \to op\,\phi$.

(iv) For any AFOL formulae $\varphi^\flat$ and $\phi^\flat$, if $\models_{cdl} \varphi^\flat \to [p]\xi$, then $\vdash^{+}_{cdl} \varphi^\flat \to [p]\xi$; if $\models_{cdl} \varphi^\flat \to \langle p\rangle \sim \xi$, then $\vdash^{+}_{cdl} \varphi^\flat \to \langle p\rangle \sim \xi$.

where $op \in \{[p], \langle p\rangle, \forall x, \exists x\}$, $x \in Var$.

The proofs of the main body and each condition of Theorem 6.2 are given in Appendix A.2. Having the conditions $(i) - (iv)$, the proof of the main body of Theorem 6.2 is direct and mainly follows [25]. In condition (i), to prove the expressibility of CDL formulae, we need to first prove the expressibility of formulae of the form $[p]\phi^\flat$ (see Lemma A.5). In FODL, the expressibility of a formula $[p]\phi^\flat$ can be obtained by the expressibility of the program p and induction on the syntactic structure of the formula. However in CDL, SEP is not a regular language, it contains the infinite loop operator $\bullet$ and the parallel operator $\|$, which makes it difficult to prove its expressibility in a direct way. To solve this problem we introduce a finite fragment of SEPs called *Finite-SEPs* (see Def. A.1). In order to show formula $[p]\phi^\flat$ (with an SEP p) is expressible, we first show that all Finite-SEPs are expressible (see Lemma A.3), then show how the case of an SEP can be reduced to the case of its corresponding Finite-SEP based on the relation between SEPs and Finite-SEPs (see Prop. A.2). The core idea behind the proof of condition (ii) is mainly based on [20, 25]. It proceeds by induction on the syntactic structure of program. Condition (iii) is straightforward according to the proof system of CDL. The proof of condition (iv) is similar to that of (ii), except that the special forms $[p]\xi$ and $\langle p\rangle \sim \xi$ need to be treated in a special way.

7. The Mechanization of CDL in Coq

In this section we discuss the mechanization of CDL in Coq. Currently, we only consider the mechanization of a fragment of CDL, which consists of all formulae that contain no parallel SEPs and all rules for sequential SEPs (i.e., the rules in Table 3, 4 and 5). In the discussion below, we call this fragment of CDL *sequential CDL* (SCDL). The implementation of the rewrite rules for parallel SEPs (i.e., the rules in Table 6) in Coq is our future work.

The main idea behind the mechanization is to use Coq as a programming language to define the language of SCDL as its types and functions and to implement the proof rules of SCDL as its theorems. Currently, we have not considered implementing the semantics of SCDL. We only define the syntax of SCDL and define the proof rules of SCDL as axioms in Coq without proving the their validity. These axioms act as a translator that transforms a given SCDL formula into a set of QF-AFOL formulae by applying suitable proof tactics in Coq, just as the procedure of the deduction shown in Section 4.4.

In the following subsections we give a brief introduction to the mechanization of SCDL in Coq. We mainly give the ideas and show some crucial points in the mechanization process. We give some critical definitions but omit the details of others. We have posted our current implementation on the website `https://github.com/antitaboo/CDL`, where interested readers can find more details about it.

7.1. A Brief Introduction to Coq

In this section we give a brief introduction to the theorem prover Coq. We only give informal explanations for some primitives of the language in Coq which is necessary for understanding the Coq code presented in this paper. Readers can refer to [22] for a more comprehensive introduction.

7.1.1. The language of Coq

The core language of Coq (called *Gallina*) is based on the theory of a typed $\lambda-$calculus called *Calculus of Inductive Constructions*. It is a type of higher-order functional programming languages provided with the ability to define inductive types.

The expression in Coq mainly consists of identifiers, function applications and $\lambda-$abstractions. Since in the Coq code of this paper only identifiers and function applications are involved, we omit the introduction to $\lambda-$abstractions. Identifiers are simple names declared in Coq. For example, we can declare an identifier x to express the integer number 1 (see Section 7.2.1 below). In Coq, the identifiers 0, 1, 2, ... are predefined to represent natural numbers $0, 1, 2, ...$ Function applications has the form: func a, where parameter a is applied to function func. The application is left associative, so term func a b c is read as ((func a) b) c. For example, we can have a function add 1 2 to express the addition of two numbers 1 and 2.

In Coq, every term has a unique type. We use t : A to represent that a term t has a type A. A *simple type* can be an *atomic type*, or an *arrow type* of the form A −> B, where A and B are simple types. Atomic types are made of single identifiers, such as nat and Prop as explained below in Section 7.1.2. nat −> nat and nat −> Prop are simple types. Arrow types are needed when we perform function application. For example, if a : A and func : A −> B, then we have (func a) : B.

In Coq, we also consider a more general type, called *dependent type*, of the form forall (x : A), t, where t : B, A and B are simple or dependent types. It means that for all x of the type A, t has the type B. t is an arbitrary term in which x might or might not appear. For example, forall (x : nat), (x = 2) −> (x > 1) is a dependent type in Coq, where (x = 2) −> (x > 1) has the type Prop. If x does not appear in t, then the type A and B are independent and forall (x : A), t is equivalent to A −> B.

7.1.2. Types nat and Prop

nat and Prop are predefined types in Coq.

The type nat represents the set of natural numbers $\{0, 1, 2, ...\}$. We often use identifiers 0, 1, 2, ... to express the elements of nat. The arithmetic operators, such as the addition (+), the substitution (−) and the multiplication (∗), and the relations, such as less-than relation (<), less-than-or-equal-to relation (<=) and equal-to relation (=) are predefined as functions in Coq. For example, in Coq we can have terms 1 + 2, 2 ∗ x and 1 < 2.

The type Prop represents the set of all propositions in Coq. Propositions can be expressed by a higher-order logic with logical connectives such as negation (∼), conjunction (/\), disjunction (\/), implication (−>), universal quantifier (forall) and existential quantifier (exists). For example, we can have a proposition forall (x : nat), (x = 2) −> (x > 1) in Coq, where x = 2 and x > 1 are two propositions of the type Prop. Note that the universal quantifier forall and the implication −> are just the same symbols for constructing arrow types and dependent types introduced above.

7.2. Proof and Proof Tactics

In Coq, theorem proving and type inference are the same thing. Propositions are expressed as dependent types. For example, the dependent type forall (x : nat), (x = 2) −> (x > 1) in Section 7.1.1 are also a proposition as indicated in Section 7.1.2. Given a type T : Prop, which is also a proposition, the process of proving T is in fact the process of finding an instance a such that a : T by applying type inference rules.

a is also called a proof of the proposition T. When a : T is declared in Coq by the command Definition or Inductive (discussed below in Section 7.2.1), a is also called an *axiom*. We use the command Theorem to define a proposition to be proved, in the form Theorem p : t, where p is the name of the theorem, t has the type Prop. For example, Theorem Reflexivity : forall (x : nat), x = x.

The process of proving a proposition can be implemented in Coq by using proof tactics. Different tactics correspond to using different type inference rules and proof strategies. Tactic apply is a basic tactic. Given an axiom a : A -> B, if we want to prove the proposition B, by applying the command apply a we only need to prove the proposition A. Tactic split is for eliminating the conjunction connective /\. If we want to prove a proposition A /\ B, by applying the command split we only need to prove the propositions A and B. Tactic intros N is for eliminating the universal quantifier forall. If we want to prove a proposition forall (x : T), t, by applying the command intros a, we obtain a new identifier a : T and only need to prove the proposition t[a/x]. Here t[a/x] means the term obtained by the substitution of x by a. Tactic omega is for automatically solving a Presburger arithmetic formula.

7.2.1. Commands Definition, Inductive and type list

In Coq, to declare an identifier in the local environment we can use the command Definition. We can write Definition (x : A) := t to declare an identifier x as a term t of type A. For example, Definition (x : nat) := 1 declares an identifier x as the integer number 1.

In Coq, a type can be inductively defined with the command Inductive. The general form of this command is:

```
Inductive T (x1 : T1) ... (xn : Tn) : O :=
        | C1 : t1
        ...
        | Cm : tm.
```

or

```
Inductive T : T1 -> ... -> Tn -> O :=
        | C1 : t1
        ...
        | Cm : tm.
```

where T is the type to be defined, its type is T1 -> ... -> Tn -> O. C1, ..., Cm are the *constructors* of the type T, while t1, ..., tm are their types. The constructors tell how the type can be constructed in an inductive way.

For example, the type nat can be defined as Fig. 16 (a), where the type of nat is the type Set. The definition says that (i) the number 0 has the type nat, and (ii) if any number n has the type nat, then S n has the type nat.

```
Inductive nat : Set :=
        | 0 : nat
        | S : nat -> nat.
```

(a)

```
Inductive list (A : Type) : Set :=
        | nil : list A
        | cons : A -> list A -> list A.
```

(b)

```
Inductive ev : nat -> Prop :=
        | ev_0 : ev 0
        | ev_SS : forall (n : nat), (ev n) -> (ev (n + 2)).
```

(c)

Figure 16: Examples of inductive types

Another example is the type list (shown as Fig. 16 (b)), whose type is the arrow type Type -> Set. The type list is a *polymorphic type*, it receives a parameter A of the type Type and returns a type list A. The definition of the type list says that given any A of the type Type, the type list A can be defined by the constructors nil and cons in the following way: (i) the empty list nil has the type list A; (ii) given any element a of the type A and a list l, then cons a l is a list of the type list A. In Coq, we often use the notation a1 :: a2 :: ... :: an :: nil to express a list cons a1 (cons a2 (... (cons an nil))).

With the command Inductive one can also define a predicate of the type T1 -> ... -> ... Tn -> Prop. When defining a predicate, the constructors C1, ..., Cm of the types t1, ..., tm can also be understood as axioms according to Section 7.2. Fig. 16 (c) shows an example, where we define a predicate ev : nat -> Prop in an inductive way: (i) ev 0 holds, and (ii) if ev n holds for any n : nat, then ev (n + 2) holds. Here ev_0 and ev_SS are two axioms. The proposition ev n describes that n is an even number.

7.3. The Mechanization of SCDL in Coq

Our mechanization consists of two steps: (1) first we define the syntax of SCDL in Coq; (2) then we define the proof system of SCDL in Coq.

The main idea behind the mechanization is that: each expression in SCDL is defined as an inductive type (introduced in Section 7.2.1); the whole proof system of SCDL is constructed through defining sequent as a predicate type, where each constructor corresponds to a rule of the proof system.

7.3.1. The Syntax of SCDL in Coq

From the discussion above in Section 7, we know that in SCDL formulae no parallel SEPs are considered. Therefore we consider a subset of SEPs where no parallel operators appear, we call them *pure sequential SEPs* (ps-SEPs). Since there is no parallel operators in ps-SEPs, we neither need to consider the signal test condition ϱ because it can only appear in a parallel program according to Def. 3.1. The syntax of ps-SEPs is defined just as Def. 3.1 but with '$\varrho\&P?\alpha$' replaced by '$P?\alpha$' and with '$\|(p_1, ..., p_n)$' being removed.

```
1  (* variable name*)
2  Inductive Var := var : nat -> Var.
3
4  (* expression e *)
5  Inductive e_exp :=
6    enumC : nat -> e_exp |
7    evarC : Var -> e_exp |
8    eplusC : e_exp -> e_exp -> e_exp |
9    eminusC : e_exp -> e_exp -> e_exp |
10   emulC : e_exp -> e_exp -> e_exp |
11   edivC : e_exp -> e_exp -> e_exp.
12
13 (* condition P *)
14 Inductive P_exp :=
15   PtrueC : P_exp |
16   PlteC : e_exp -> e_exp -> P_exp |
17   PnegC : P_exp -> P_exp |
18   PandC : P_exp -> P_exp -> P_exp |
19   (*unnecesarry expression*)
20   PfalseC : P_exp |
21   PltC : e_exp -> e_exp -> P_exp |
22   PgtC : e_exp -> e_exp -> P_exp |
23   PgteC : e_exp -> e_exp -> P_exp |
24   PeqC : e_exp -> e_exp -> P_exp |
25   PorC : P_exp -> P_exp -> P_exp |
26   PimpC : P_exp -> P_exp -> P_exp.
27
28 (* event element *)
29 Inductive EvtElement :=
30   sigC : Var -> e_exp -> EvtElement |
31   assC : Var -> e_exp -> EvtElement.
32
33 (* event *)
34 Definition Evt := list EvtElement.
35
36 (* syntax of SEP *)
37 Inductive SEP_exp : Type :=
38   skip : SEP_exp |
39   evtC : Evt -> SEP_exp |
40   tstC : P_exp -> Evt -> SEP_exp |
41   seqC : SEP_exp -> SEP_exp ->
          SEP_exp |
42   choC : SEP_exp -> SEP_exp ->
          SEP_exp |
43   loopC : SEP_exp -> SEP_exp.
44
45 (* idle event *)
46 Definition idle : Evt := nil .
```

Term	Notation in Coq	Meaning in SCDL	Term	Notation in Coq	Meaning in SCDL
var n	x, y, z, c,...	names x, y, z, c...	enumC n	n	integer n in e
evarC v	v	a variable in e	eplusC e1 e2	e1 '+ e2	expression $e_1 + e_2$ in e
PtrueC	'tt	formula tt in P	PlteC e1 e2	e1 '<= e2	expression $e_1 \leq e_2$ in e
PnegC P	'~ P	formula $\neg P$ in P	PandC P1 P2	P1 '/\ P2	formula $P_1 \wedge P_2$ in P
sigC s e	s ! e	signal $\varsigma!e$	assC x e	x :=' e	assignment $x := e$
skip		skip program μ	evtC (b1 :: ... :: bn :: nil)	@{b1 | ... | bn}	combinational event $\{Cmb_1, ..., Cmb_n\}$
tstC P a	P ? a	test event $P?a$	seqC p1 p2	p1 ; p2	program $p_1; p_2$
choC p1 p2	p1 U p2	program $p_1 \cup p_2$	loopC p	p **	program $p^\bullet$

Table 8: The definition of the syntax of ps-SEPs in Coq

Table 8 shows the definition of the syntax of ps-SEPs in Coq. It is defined as an inductive type SEP_exp [3] (line 37) in the Coq code. The syntax of other components of ps-SEPs, i.e., the expression e, the condition P and the combinational event α, are defined as inductive types e_exp (line 5), P_exp (line 14) and Evt (line 34) respectively. The diagram in the below of Table 8 gives the correspondences between the constructors of these inductive types in Coq and their meanings in SCDL. For most constructors we define notations to improve their readabilities, in forms as close as possible to the forms of their correspondences in SCDL.

As indicated in Table 8, the set of all general variable names and all clock names in SCDL are defined as the inductive type Var in Coq, whose elements are constructed by the constructor var with natural numbers (nat). For simplicity, currently we do not distinguish general variable names and clock names in the Coq code. One advantage of using natural numbers instead of using strings to define names is that it becomes much easier to generate new names. By default, in Coq we use notations x, y, z and k as shorthands for

[3] For the convenience of extending our programs in the future, we still name ps-SEP as 'SEP' in our Coq code.

terms var 1, var 2, var 3 and var 4 respectively. They denote the general variable names x, y, z, k in SCDL respectively. We use notations c, d, c1 - c4 as shorthand for terms var 5, var 6, var 7 - var 10 respectively. They denote the clock names c, d, c_1 - c_4 in SCDL respectively. For simplicity we also do not distinguish clock names and signal names and consider that they are the same thing in the Coq code. Actually, due to Def. 3.2 we assume there is always a bijection $SigM$ between them.

In the type e_exp, terms enumC n and evarC v express a number n and a variable in the expression e respectively. They can be simply written as n and v in Coq respectively. For example, term evarC (var 1) expresses the variable x in e, it can be simply written as x in Coq, where x is a shorthand of variable var 1 as discussed above.

In the inductive type P_exp, for convenience of expressing we also define the constructors for unnecessary expressions (such as false ff, the less-than expression $e_1 \leq e_2$, etc) (lines 20 - 26) which can be expressed by the expressions in P defined in Def. 3.1. In the diagram of Table 8 we omit the explanations for their constructors (such as PfalseC, PltC, etc). We also omit the explanations for the constructors eminusC, emulC and edivC in e_exp (lines 9 - 11).

In the inductive type Evt, the type list is defined in Section 7.2.1.

```
1  (* CCSL clock relation *)
2  Inductive CRel :=
3    crSubClC : Var -> Var -> CRel |
4    crExclC : Var -> Var -> CRel |
5    crPrecC : Var -> Var -> CRel |
6    crCausC : Var -> Var -> CRel.
7
8  (* clock-relation term in SCDL formulae
       *)
9  Inductive rel :=
10   rRelC : CRel -> rel |
11   rConjC : list CRel -> rel.
12
13 (* expression E *)
14 Inductive E_exp :=
15   EvarC : Var -> E_exp |
16   ECntClkC : Var -> E_exp |
17   EStClkC : Var -> E_exp |
18   EnumC : nat -> E_exp |
19   EplusC : E_exp -> E_exp -> E_exp |
20   (* unecessary expression *)
21   EmulC : E_exp -> E_exp -> E_exp |
22   EminusC : E_exp -> E_exp -> E_exp |
23   EdivC : E_exp -> E_exp -> E_exp.
24
25 (* syntax of SCDL formula *)
26 Inductive CDL_exp :=
27   cdl_trueC : CDL_exp |
28   cdl_lteC : E_exp -> E_exp ->
         CDL_exp |
29   cdl_box1C : SEP_exp -> rel ->
         CDL_exp |
30   cdl_box2C : SEP_exp -> CDL_exp ->
         CDL_exp |
31   cdl_negC : CDL_exp -> CDL_exp |
32   cdl_andC : CDL_exp -> CDL_exp ->
         CDL_exp |
33   cdl_forallC : Var -> CDL_exp ->
         CDL_exp |
34   (*unnecessary expressions*)
35   cdl_falseC : CDL_exp |
36   cdl_ltC : E_exp -> E_exp -> CDL_exp
         |
37   cdl_gtC : E_exp -> E_exp -> CDL_exp
         |
38   cdl_gteC : E_exp -> E_exp ->
         CDL_exp |
39   cdl_eqC : E_exp -> E_exp -> CDL_exp
         |
40   cdl_dia1C : SEP_exp -> rel ->
         CDL_exp |
41   cdl_dia2C : SEP_exp -> CDL_exp ->
         CDL_exp |
42   cdl_orC : CDL_exp -> CDL_exp ->
         CDL_exp |
43   cdl_impC : CDL_exp -> CDL_exp ->
         CDL_exp |
44   cdl_existsC : Var -> CDL_exp ->
         CDL_exp.
```

Symbol	Notation in Coq	Meaning in SCDL	Symbol	Notation in Coq	Meaning in SCDL
crSubClC c1 c2	c1 sub c2	clock relation $c_1 \subseteq c_2$	crExclC c1 c2	c1 # c2	clock relation $c_1 \# c_2$
crPrecC c1 c2	c1 << c2	clock relation $c_1 \prec c_2$	crCausC c1 c2	e1 <<= e2	clock relation $e_1 \preceq e_2$
rRelC r	r	a clock relation in formulae	rConjC (r1 :: ... :: rn :: nil)	/\{r1,...,rn}	term $\curlywedge(Rel_1, ..., Rel_n)$
EvarC v	v	a variable in E	ECntClkC c	n(c)	variable c^n in E
EStClkC c	s(c)	variable c^s in E	EnumC n	n	integer n in E
EplusC e1 e2	e1 +' e2	expression $e_1 + e_2$ in E	cdl_trueC	tt'	true tt in E
cdl_lteC e1 e2	e1 <=' e2	expression $e_1 \leq e_2$ in E	cdl_box1C p rel	[[p]]' rel	formula $[p]\xi$
cdl_box2C p phi	[[p]] phi	formula $[p]\phi$	cdl_negC phi	~' phi	formula $\neg\phi$
cdl_andC phi1 phi2	phi1 /\' phi2	formula $\phi_1 \wedge \phi_2$	cdl_forallC x phi	all x, e	formula $\forall x.\phi$

Table 9: The definition of the syntax of SCDL formulae in Coq

Table 9 shows the definition of the syntax of SCDL formulae in Coq. It is defined as the inductive type CDL_exp (line 26). The other components of SCDL formulae, i.e., the CCSL clock relation Rel, the clock-relation term ξ and the expression E are defined as the inductive types CRel (line 2), rel (line 9) and E_exp (line 14) respectively. The diagram in the below of Table 9 gives the notations of the constructors of these types and their explanations in SCDL. For the same reason mentioned above, we also define the constructors for unnecessary expressions in types E_exp (lines 21 - 23) and CDL_exp (lines 35 - 44), and their explanations are omitted in the diagram of Table 9.

In the type rel, term rRelC r expresses a clock relation in the expression E. They can be simply written as r in Coq, similar explanations can be made for term EvarC v and term EnumC n in the type E_exp. For

example, term rRelC (c1 sub c2) expresses the relation $c_1 \subseteq c_2$ in the expression E, it can be simply written as c1 sub c2 in Coq. The type list is defined in Section 7.2.1.

Note that in the Coq code, the constructors of the arithmetic operators (e.g., $+$, $\cdot$, etc) and relations (e.g., $\leq$, $<$, etc) in the expression E and the constructors of the logical terms (i.e., tt and ff) and connectives (e.g., $\neg$, $\wedge$, etc) in SCDL formulae are different from those in the expression e and the condition P, though we use the same syntactic symbols to express them in Def. 3.1 and 3.4. In Coq, their notations differ from each other according to the positions of the prime symbol ''. For example, in e the plus operator is expressed as ''+' in Coq, while in E, the plus operator is expressed as '+''.

```
1  (* a small example *)
2  (* general variables x, f*)
3  Definition f : Var := (var 11).
4  Definition Y : Var := (var 12).
5
6  (* clock *)
7  Definition e : Var := (var 13).
8  Definition p : Var := (var 14).
9  Definition r : Var := (var 15).
10 Definition o : Var := (var 16).
11
12 (* program p2 *)
13 Definition P1 : P_exp := (x '= 2) '/\ (f '= 0).
14 Definition P2 : P_exp := (x '< 2) '/\ (f '= 0).
15
16 Definition a2 : Evt := { p ! 0 | f :=' 1 }.
17 Definition a3 : Evt := { p ! 0 | x :=' x '+ 1 }.
18
19 Definition p2 : SEP_exp := P1 ? a2 U P2 ? a3.
```

Term in Coq	Meaning in SCDL	Term in Coq	Meaning in SCDL	Term in Coq	Meaning in SCDL
f	variable f	Y	variable $(c_r^n)'$	e	clock c_e
p	clock c_p/signal ς_p	r	clock c_r/signal ς_r	o	clock $c_o$signal ς_o

Table 10: The definition of program p_2 in Coq

Example 7.1. *We consider the formula $[p_2]c_r \prec c_o$ in the sequent of the node ⑬ of the deduction procedure of $I \rightarrow [DF]c_r \prec c_o$ (Fig. 14). The program p_2 can be defined in the Coq code shown in Table 10, where the meaning of each variable term is explained in the diagram below. As discussed above, the name* x *is defined as $var1$. Note that as mentioned above, a clock and its corresponding signal can be expressed with the same name. The program p_2 is defined as the term* p2 *(line 19).*

With the program p2*, the formula $[p_2]c_r \prec c_o$ can be described as a* CDL_exp *expression:*

```
[[ p2 ]]' r << o.
```

7.3.2. The Proof System of SCDL in Coq

The proof system of CDL proposed in Section 4 is based on the sequent of the form $\Gamma \Rightarrow \Delta$ (defined in Section 4.1). However, such an argumentation is not suitable for making deductions in Coq because it is highly non-deterministic: each time any formula in the sets Γ and Δ can be chosen as a target formula according to which rule is chosen to be applied. To solve this problem we make two augmentations when we implement sequent in Coq:

(1) We define the multi-sets Γ and Δ as ordered lists, instead of sets. In this way we can determine which formula will be dealt with next, rather than non-deterministically choosing one.

(2) We introduce the notion of *target place* in the sequent. A target place can either be empty or store a formula that should be focused in the current sequent. To support this new feature we need to introduce new proof rules in the proof system of SCDL for removing the current formula from the place and adding the next formula to the place.

Besides (1) and (2) above, in order to make the form of the proof rules in Coq as simple as possible, for each sequent we also keep the information about the current set of variable names and clock names, the set of all clocks and the set of all clocks that do not tick at the current instant.

Based on the above discussion, formally, a sequent can be defined in Coq as a structure:

$$\Gamma, \mathfrak{p}_1 \Rightarrow \mathfrak{p}_2, \Delta \ /\!/ \ \mathfrak{V}, C, ntC,$$

where $\Gamma, \mathfrak{p}_1 \Rightarrow \mathfrak{p}_2, \Delta$ is a sequent (defined in Def. 4.1) with two sets $\mathfrak{p}_1$ and $\mathfrak{p}_2$ called 'target places' on both of its sides. $\mathfrak{p}_1$ and $\mathfrak{p}_2$ can contain at most one formula. $\mathfrak{V}, C, ntC$ provide auxiliary information about a sequent, they are separated from a sequent with a symbol $/\!/$. $\mathfrak{V}$ represents the current set of variable names and clock names, C represents the set of all clocks and ntC represents the set of all clocks that do not tick at the current instant.

Appendix B lists a part of code for the definition of the proof system of SCDL in Coq. The sets Γ and Δ are defined as two lists with the type list Var (i.e., the types Gamma and Delta at lines 3 - 4) in Coq. Target places are defined as the inductive type place (line 7). A target place can be either empty (expressed by the term empty) or can be a CDL formula phi (expressed by the term exp phi).

In Coq, a sequent is defined as a predicate (line 15):

```
Seq : Gamma -> place -> place -> Delta -> (list Var) - > (list Var) -> (list Var) -> Prop.
```

Each sequent Seq T p1 p2 D V C ntC is a proposition of the type Prop, where T, p1, p2, D, V, C, ntC represent components Γ, $\mathfrak{p}_1$, $\mathfrak{p}_2$, Δ, $\mathfrak{V}$, C, ntC respectively. As indicated in Section 7.2.1, the sequent is defined in an inductive way by its constructors — as the axioms that capture the proof rules of SCDL in Coq. In another word, we define a sequent according to whether it can be proved by these axioms in an inductive way. Once we have defined the sequent, we finish constructing the proof system of SCDL.

In this paper, we only list the constructors for some rules in Appendix B (lines 18 - 183) as an example to explain how the proof system of SCDL is defined in Coq. Currently, we only implement all rules for one direction, i.e., the direction from the premises to the conclusion. Only these rules are applied for transforming a dynamic CDL formula into QF-AFOL formulae. Rules for the other direction only matter when concerning the relative completeness of the logic and we leave their implementations in our future work.

Generally, all types of the constructors are of the form:

```
forall ... (some parameters),
           (Seq T1 p11 p12 D1 V1 C1 ntC1) * ... * (Seq Tn pn1 pn2 Dn Vn Cn ntCn)
           ->
           Seq T p1 p2 D V C ntC.
```

where $* \in \{/\backslash, \backslash/\}$. It represents the proof rule $\dfrac{\Gamma_1, \mathfrak{p}_{11} \Rightarrow \mathfrak{p}_{12}, \Delta_1 ... \Gamma_n, \mathfrak{p}_{n1} \Rightarrow \mathfrak{p}_{n2}, \Gamma_n}{\Gamma, \mathfrak{p}_1 \Rightarrow \mathfrak{p}_2, \Delta}$ when $*$ is $/\backslash$, and represents n proof rules $\dfrac{\Gamma_1, \mathfrak{p}_{11} \Rightarrow \mathfrak{p}_{12}, \Delta_1}{\Gamma, \mathfrak{p}_1 \Rightarrow \mathfrak{p}_2, \Delta}, ..., \dfrac{\Gamma_n, \mathfrak{p}_{n1} \Rightarrow \mathfrak{p}_{n2}, \Delta_n}{\Gamma, \mathfrak{p}_1 \Rightarrow \mathfrak{p}_2, \Delta}$ when $*$ is $\backslash/$. We use the notation

```
< | T , p1 ==> p2 , D // V , C , ntC | >
```

to express the sequent Seq T p1 p2 D V C ntC in Coq (line 183).

The constructors r_Test_all_rel_int (line 111), r_PiCho_all_int (line 123), r_o_int (line 137), r_andR_int (line 152), r_impR_int (line 170) correspond to the proof rules $(P?[])$, $(\pi[\cup])$, (o), $(\wedge r)$, $(\rightarrow r)$ respectively.

As mentioned in the augmentation (2) above at the beginning of Section 7.3.2, we need to propose special rules for removing a formula from and adding a formula to a target place. The constructors r_placeR_rmv_int (line 18) (resp. r_placeL_rmv_int (line 29)) is for removing a formula from the target place $\mathfrak{p}_2$ (resp. $\mathfrak{p}_1$) and adding it to the nail of the list Δ (resp. Γ). For example, the type of the constructor r_placeR_rmv_int means that to prove Seq T pls1 (exp phi) D V C ntC, we need to prove Seq T pls1 empty (addNail D phi) V C ntC, where formula phi is removed from the target place (so that the target place becomes empty) and added to D. The function addNail adds the formula phi to the nail of the list D. Two other special constructors are r_placeR_add_int and r_placeL_add_int (not shown in Appendix B), they are for adding a formula from the head of the list Δ or Γ to the target place $\mathfrak{p}_2$ or $\mathfrak{p}_1$.

The constructors r_Pi_all_int1 (line 43), r_Pi_all_int2 (line 56), r_Pi_all_int_idle1 (line 79) and r_Pi_all_int_idle2 (line 90) are for the single rule $(\pi[])$ (similarly, there are constructors for the rule $(\phi[])$, which are not shown in Appendix B). Recall that in $(\pi[])$, the combinational event α is dealt with as a whole in one derivation.

However in Coq, to implement this rule as one single constructor is not convenient. To make it as simple as possible we split the combinational event into its atomic events, and we build constructors to deal with the execution of a single atomic event. The constructors r_Pi_all_int1 and r_Pi_all_int2 deal with the executions of the assignments and signals respectively. When all events are dealt with in the combinational event, we need to build constructors for dealing with its end. The constructors r_Pi_all_int_idle1 and r_Pi_all_int_idle2 are defined for this purpose. The constructor r_Pi_all_int_idle2 deals with all clocks that do not tick at the current instant (i.e. the set ntC): according to ($\pi[]$), the value of their clock-related variables ($d_1^s, ..., d_n^s$) should be set to 0. The constructor r_Pi_all_int_idle1 finally eliminates the combinational event once all clocks in ntC have been dealt with. In these 4 constructors, the function NewId is for generating new variable names, the functions of the notations e [E 'subs-E' u] and T [E 'subs-I' u] are for the substitutions of different expressions. We omit their details here.

```
1  (* Gamma2 *)
2  Definition Gamma2 : Gamma := (Y >=' n(o)) :: (n(r) =' Y +' 1) ::
3    (s(r) =' 1) :: (x =' 1) :: (f =' 0) :: (s(e) =' 0) ::
4    (s(p) =' 0) :: (s(o) =' 0) :: nil .
5
6  Theorem example : <| Gamma2 , empty ==>
7    (exp ([[ p2 ]]' r << o)) , nil //
8    (x :: f :: Y :: e :: p :: r :: o :: nil ), (* V *)
9    (e :: p :: r :: o :: nil ), (* C *)
10   (e :: p :: r :: o :: nil ) (* ntC *)
11          |>.
12
13 Proof.
14   apply r_PiCho_all_int .
15   apply r_andR_int . split .
16   Focus 2.
17   - apply r_Test_all_rel_int .
18     apply r_impR_int .
19     apply r_placeL_rmv_int .
20     apply r_Pi_all_int2 ;cbv.
21     apply r_Pi_all_int1 ;cbv.
22     apply r_Pi_all_int_idle2 ;apply r_Pi_all_int_idle2 ;apply
           r_Pi_all_int_idle2 ; cbv.
23     apply r_Pi_all_int_idle1 ; cbv.
24     apply r_placeR_rmv_int .
25     apply r_o_int . cbn.
26     intros H1 H2 H3.
27     omega.
28 Admitted.
```

```
1 subgoal
H1, H2, H3 : Var -> nat
____________________________________(1/1)
(H3 (var 16) = 0 /\
 H3 (var 15) = 0 /\
 H3 (var 13) = 0 /\
 H1 (var 1) = H1 (var 19) + 1 /\
 H2 (var 14) = H1 (var 17) + 1 /\
 H3 (var 14) = 1 /\
 H1 (var 12) >= H2 (var 16) /\
 H2 (var 15) = H1 (var 12) + 1 /\ H1 (var 21) = 1 /\ H1 (var 19) = 1 /\ H1 (var 11) = 0 /\ H1 (var 20) = 0 /\ H1 (var 18) = 0 /\
H1 (var 22) = 0 /\ (H1 (var 19) < 2 /\ H1 (var 11) = 0) /\ True) /\ True ->
False \/ (H2 (var 15) > H2 (var 16) \/ (H2 (var 15) = H2 (var 16) -> H3 (var 15) = 0)) \/ False
|
```

Figure 17: The proof of sequent $\Gamma_2 \Rightarrow [p_2]c_r \prec c_o$ in Coq

Example 7.2. *We consider the derivation from the node ⑬ of the deduction procedure of $I \rightarrow [DF]c_r \prec c_o$ (Fig. 14). The problem for proving the sequent $\Gamma_2 \Rightarrow [p_2]c_r \prec c_o$ is described as a* theorem *example (line 6) in the Coq code shown in Fig. 17, where the program p2 and the names f, Y, e, p, r, o have been shown in Table 10.*

Following the deduction procedure shown in Fig. 14, we conduct a proof in Coq by starting the statement Proof. (line 13). Here for simplicity we only perform the deduction of the branch from the node ⑬ to the node ⑫ in Fig. 14. By applying the tactic apply to the constructors (as axioms in Coq) in the same order as their corresponding rules that are applied on this branch (lines 14 - 26), we transform the formula [[p2]]' r << o into a QF-AFOL formula shown in the below of Fig. 17. This formula contains uninterpreted functions H1, H2, H3 that map a name to a natural number in Coq. However, these uninterpreted functions can actually be eliminated since their parameters never appear alone in the formula. So this formula is in fact a Presburger arithmetic formula.

At last, by applying the tactic omega (line 27), we solve the formula and prove the branch.

8. Related Work

Previous approaches [6, 7, 14, 9, 15] for the verification of CCSL specifications were mainly based on model checking, where a CCSL specification was encoded as a target model, and reachability analysis was

made on the product of the system model and the model of the CCSL specification. Different encoding principles were proposed according to different types of target models, such as timed automata [6, 7, 15], data flow graphs [14], Esterel programs [9], etc. When a CCSL specification is unbounded, a bound needs to be set to avoid the enumeration of an infinite number of states (e.g. in [9]). Our approach is based on a method combining theorem proving and SMT checking, which provides a unified verification framework under which both bounded and unbounded CCSL specifications can be directly analyzed.

Another aspect of formal analysis of CCSL is to find a schedule (or a set of schedules) of a given CCSL specification where no system models were involved [2, 17, 39]. The earliest approach [2] combined BDD-based boolean solving and the rewrite relations on clock expressions, while the recent method in [39] was based on the rewriting logic of Maude. In [17], the search of a schedule was conducted by solving a logical formula that encodes CCSL specifications through an SMT-checking procedure. Compared with [17], the proposed CDL provides a natural way for capturing CCSL specifications and a modular way for transforming the CDL formula into QF-AFOL formulae, which are easier and more efficient for an SMT checker to solve.

Dynamic logic, first proposed by V.R. Pratt [40], is a formalism for modeling and reasoning about program specifications. The syntax and semantics of CDL are largely based on those of FODL [25] and those of its extension to concurrency [29]. Process logic [41] combines the features of both dynamic logic and temporal logic. In process logic, formulae of the form $[p]\Box\phi$ mean that all execution traces of p satisfy the temporal formula $\Box\phi$. The semantics of SEPs and the formulae of the form $[p]\xi$ are mainly inherited from the semantics of the formulae of the form $[p]\Box\phi$ in process logic. Differential Dynamic Temporal Logic [32, 31] (DDTL) is a dynamic logic for the specification and verification of hybrid systems. In DDTL, programs support a continuous time model with differential equations embedded into it to express the physical dynamics of hybrid systems. Compared with it, the SEPs of CDL support a discrete time model with a synchronous execution mechanism to specify the behaviours of synchronous systems. The rules $(\pi[;])$, $(\pi[\cup])$, $(\pi[\bullet]u)$ and $(\pi[\bullet]i)$ in Table 3 are largely inspired from the corresponding rules for formulae of the form $[p]\Box\phi$ in DDTL.

There are many other variations of dynamic logic for reasoning about different systems, such as Java Card Dynamic Logic [42] for Java programs, Object-Oriented Dynamic Logic [43] for object-oriented programs, Differential Dynamic Logic [44] for hybrid systems, etc. To our knowledge, CDL is the first dynamic logic for synchronous systems.

9. Conclusion and Future Work

In this paper, we propose a novel dynamic logic — CDL — for the verification of CCSL specifications in synchronous systems. We define the syntax and semantics of CDL, and we build a proof system for CDL to semi-automatically transform a CDL formula into QF-AFOL formulae, which can be efficiently checked through an SMT-checking procedure. We show how CCSL specifications can be expressed in CDL by encoding each clock definition as an SEP. We analyze the soundness and completeness of CDL, and we give complete proofs for critical theorems and lemmas in Appendix A. Finally we mechanize a part of CDL using Coq, and illustrate how CDL formulae can be proved in Coq with a small example. Throughout this paper we illustrate our viewpoint through a simple example, the digital filter system.

Future work will focus on two aspects:

1. In this paper, the CDL we propose supports the parallel model of synchronous systems. However, the proof system of CDL does not support the verification of parallel model in a modular way: we verify a parallel program by reducing it into an equivalent sequential program. The modular verification for parallel model is important for SEPs because unlike traditional imperative programs, parallel is a nature of synchronous systems. Next we will focus on providing a rely-guarantee-style [45] compositional proof system for parallel SEPs. An automatic (or semi-automatic) method could be proposed for generating suitable "rely" and "guarantee" predicates for each clock relation and clock definition. Such a proof system could greatly enhance the applicability of CDL in synchronous systems.

2. We will also consider a full mechanization of CDL in Coq. With it we can consider more practical applications in order to see more practical potentials of this logic.

10. Acknowledgements

This work has been partialy supported by the French government, through the EUR DS4H Investments in the Future project managed by the National Research Agency (ANR) with the reference number ANR-17-EURE-0004.

References

[1] F. Mallet, Clock constraint specification language: specifying clock constraints with UML/MARTE, Innovations in Systems and Software Engineering 4 (3) (2008) 309–314.

[2] C. André, Syntax and semantics of the clock constraint specification language (CCSL), Research Report RR-6925, INRIA (2009).

[3] OMG, UML profile for MARTE: Modeling and analysis of real-time embedded systems, Tech. rep., OMG (2011).

[4] OMG, UML Profile for Schedulability, Performance, and Time Specification, v1.1, 2005, available at `https://www.omg.org/spec/SPTP/1.1`.

[5] D. Du, P. Huang, K. Jiang, F. Mallet, pCSSL: A stochastic extension to MARTE/CCSL for modeling uncertainty in cyber physical systems, Science of Computer Programming 166 (2018) 71 – 88.

[6] J. Suryadevara, C. Seceleanu, F. Mallet, P. Pettersson, Verifying MARTE/CCSL mode behaviors using UPPAAL, in: Software Engineering and Formal Methods (SEFM), Vol. 8137 of Lecture Notes in Computer Science (LNCS), Springer Berlin Heidelberg, 2013, pp. 1–15.

[7] L. Yin, F. Mallet, J. Liu, Verification of MARTE/CCSL time requirements in Promela/SPIN, in: International Conference on Engineering of Complex Computer Systems (ICECCS), IEEE Computer Society, 2011, pp. 65–74.

[8] B. Chen, X. Li, X. Zhou, Model checking of MARTE/CCSL time behaviors using timed I/O automata, Journal of Systems Architecture - Embedded Systems Design 88 (2018) 120–125.

[9] C. André, F. Mallet, Specification and verification of time requirements with CCSL and Esterel, in: ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), ACM, 2009, pp. 167–176.

[10] H. Yu, J.-P. Talpin, L. Besnard, T. Gautier, H. Marchand, P. Guernic, Polychronous controller synthesis from MARTE CCSL timing specifications, in: Ninth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMPCODE2011), Cambridge, 2011, pp. 21 – 30.

[11] A. Benveniste, P. L. Guernic, C. Jacquemot, Synchronous programming with events and relations: the signal language and its semantics, Science of Computer Programming 16 (2) (1991) 103 – 149.

[12] G. Berry, G. Gonthier, The Esterel synchronous programming language: design, semantics, implementation, Science of Computer Programming 19 (2) (1992) 87 – 152.

[13] F. Mallet, R. de Simone, Correctness issues on MARTE/CCSL constraints, Science of Computer Programming 106 (2015) 78 – 92.

[14] F. Mallet, J. Deantoni, C. André, R. de Simone, The clock constraint specification language for building timed causality models - application to synchronous data flow graphs, Innovations in Systems and Software Engineering 6 (2010) 99–106.

[15] Y. Zhang, F. Mallet, Y. Chen, Timed automata semantics of spatial-temporal consistency language STeC, in: International Symposium on Theoretical Aspects of Software Engineering (TASE), IEEE, 2014, pp. 201–208.

[16] F. Mallet, J.-V. Millo, R. de Simone, Safe CCSL specifications and marked graphs, in: ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE), IEEE, 2013, pp. 157–166.

[17] M. Zhang, F. Mallet, H. Zhu, An SMT-based approach to the formal analysis of MARTE/CCSL, in: ICFEM, Vol. 10009 of Lecture Notes in Computer Science (LNCS), 2016, pp. 433–449.

[18] M. Zhang, Y. Ying, Towards SMT-based LTL model checking of clock constraint specification language for real-time and embedded systems, in: ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), ACM, 2017, pp. 61–70.

[19] C. A. R. Hoare, An axiomatic basis for computer programming, Commun. ACM 12 (10) (1969) 576–580.

[20] D. Harel, D. Kozen, J. Tiuryn, Dynamic logic, SIGACT News 32 (1) (2001) 66–69.

[21] S. A. Cook, Soundness and completeness of an axiom system for program verification, SIAM Journal on Computing 7 (1) (1978) 70–90.

[22] Y. Bertot, P. Castéran, Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions, Texts in Theoretical Computer Science. An EATCS Series, Springer, 2004.

[23] C. Barrett, P. Fontaine, C. Tinelli, The SMT-LIB standard: Version 2.6, Tech. rep., Department of Computer Science, The University of Iowa, available at `www.SMT-LIB.org` (2017).

[24] T. Nipkow, L. C. Paulson, M. Wenzel, Isabelle/HOL — A Proof Assistant for Higher-Order Logic, Vol. 2283 of Lecture Notes in Computer Science (LNCS), Springer, 2002.

[25] D. Harel, First-Order Dynamic Logic, Vol. 68 of Lecture Notes in Computer Science (LNCS), Springer, 1979.

[26] A. Blass, Y. Gurevich, Inadequacy of computable loop invariants, ACM Transactions on Computational Logic 2 (1) (2001) 1–11.

[27] Y. Zhang, H. Wu, Y. Chen, F. Mallet, Embedding CCSL into dynamic logic: A logical approach for the verification of CCSL specifications, in: Formal Techniques for Safety-Critical Systems (FTSCS), Vol. 1008 of Communications in Computer and Information Science (CCIS), Springer International Publishing, 2019, pp. 101–118.

[28] G. Berry, The Constructive Semantics of Pure Esterel, 1999, available at `http://www.Esterel.org`.

[29] D. Peleg, Communication in concurrent dynamic logic, Journal of Computer and System Sciences 35 (1) (1987) 23–58.

[30] G. Gentzen, Untersuchungen über das logische schließen, Ph.D. thesis, NA Göttingen (1934).

[31] A. Platzer, A temporal dynamic logic for verifying hybrid system invariants, in: Logical Foundations of Computer Science (LFCS), Vol. 4514 of Lecture Notes in Computer Science (LNCS), Springer, 2007, pp. 457–471.

[32] A. Platzer, Logical Analysis of Hybrid Systems - Proving Theorems for Complex Dynamics, Springer, 2010.

[33] J. A. Brzozowski, Derivatives of regular expressions, Journal of the ACM 11 (4) (1964) 481–494.

[34] D. N. Arden, Delayed-logic and finite-state machines, in: SWCT (FOCS), IEEE Computer Society, 1961, pp. 133–151.

[35] M. R. Laurence, G. Struth, Omega algebras and regular equations, in: Relational and Algebraic Methods in Computer Science (RAMICS), Vol. 6663 of Lecture Notes in Computer Science (LNCS), Springer, 2011, pp. 248–263.

[36] C. André, F. Mallet, Specification and verification of time requirements with CCSL and Esterel, in: ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), ACM, 2009, pp. 167–176.

[37] F. Mallet, Automatic generation of observers from MARTE/CCSL, in: IEEE International Symposium on Rapid System Prototyping (RSP), IEEE, 2012, pp. 86–92.

[38] K. Gödel, Über formal unentscheidbare sätze der principia mathematica und verwandter systeme, Monatshefte für Mathematik und Physik 38 (1) (1931) 173–198.

[39] M. Zhang, F. Dai, F. Mallet, Periodic scheduling for MARTE/CCSL: Theory and practice, Science of Computer Programming 154 (2018) 42 – 60.

[40] V. R. Pratt, Semantical considerations on Floyd-Hoare logic, in: Annual IEEE Symposium on Foundations of Computer Science (FOCS), IEEE Computer Society, 1976, pp. 109–121.

[41] D. Harel, D. Kozen, R. Parikh, Process logic: Expressiveness, decidability, completeness, Journal of Computer and System Sciences 25 (2) (1982) 144–170.

[42] K. Rustan, M. Leino, Verification of Object-Oriented Software. The KeY Approach, Vol. 4334 of Lecture Notes in Computer Science (LNCS), Springer, 2007.

[43] B. Beckert, A. Platzer, Dynamic logic with non-rigid functions, in: U. Furbach, N. Shankar (Eds.), Automated Reasoning, Vol. 4130 of Lecture Notes in Computer Science (LNCS), Springer Berlin Heidelberg, 2006, pp. 266–280.

[44] A. Platzer, Differential dynamic logic for verifying parametric hybrid systems, in: International Conference on Theorem Proving with Analytic Tableaux and Related Methods (TABLEAUX), Vol. 4548 of Lecture Notes in Computer Science (LNCS), Springer Berlin Heidelberg, 2007, pp. 216–232.

[45] C. B. Jones, Specification and design of (parallel) programs, in: IFIP Congress, 1983, pp. 321–332.

[46] C. Leary, A Friendly Introduction to Mathematical Logic, Milne Library, SUNY Geneseo, 2015.

A. The Proof of Soundness and Relative Completeness of CDL

A.1. The Proof of Theorem 6.1

We only give the proofs for some rules in Table 3, for the rest of the rules see an analysis in Section 6.1.

Theorem 6.1. 1. For rules $(\pi[])$, $(\alpha[])$, $(\pi\langle\rangle)$ and $(\phi\langle\rangle)$, we take rule $(\alpha[])$ for example, others are similar. By the definition of the rule: $\dfrac{\Gamma' \Rightarrow \Delta'}{\Gamma \Rightarrow \Delta}$ in Section 4.1, we need to prove the following two propositions:

(i) If $\forall s \in S, s \models_{cdl} (\bigwedge_{\varphi_1 \in \Gamma} \varphi_1[V'/V] \wedge P_1) \rightarrow (P_2 \vee \bigvee_{\varphi_2 \in \Delta} \varphi_2[V'/V])$, then $\forall s \in S, s \models_{cdl} \bigwedge_{\varphi_1 \in \Gamma} \varphi_1 \rightarrow (P_3 \vee \bigvee_{\varphi_2 \in \Delta} \varphi_2)$.

(ii) If $\forall s \in S, s \models_{cdl} \bigwedge_{\varphi_1 \in \Gamma} \varphi_1 \rightarrow (P_3 \vee \bigvee_{\varphi_2 \in \Delta} \varphi_2)$, then $\forall s \in S, s \models_{cdl} (\bigwedge_{\varphi_1 \in \Gamma} \varphi_1[V'/V] \wedge P_1) \rightarrow (P_2 \vee \bigvee_{\varphi_2 \in \Delta} \varphi_2[V'/V])$.

where $P_1 = (c^n = (c^n)' + 1 \wedge c^s = 1 \wedge x = e'[V'/V] \wedge ..... \wedge \bigwedge_{1 \leq i \leq n} d_i^s = 0)$, $P_2 = \phi$, $P_3 = [(\varsigma^c!e|x := e'|.....)]\phi$. α, $d_1, ..., d_n$, V, V' are defined as in rule $(\phi[])$ of Table 4.

For (i), for any $s \in S$, if $s \models_{cdl} \bigwedge_{\varphi_1 \in \Gamma} \varphi_1$, we need to show that $s \models_{cdl} P_3 \vee \bigvee_{\varphi_2 \in \Delta} \varphi_2$. We construct an s' such that

$$\begin{cases} s'(c^n) = s(c^n) + 1, s'(c^s) = 1, & \\ s'(x) = Eval_s(e'), & \\ ..... & \\ s'(d_1^s) = 0, ..., s'(d_n^s) = 0, & \\ s'(z') = s(z), & \text{for each new variable } z' \in V' \\ s'(y) = s(y), & \text{for other variable } y \notin V \end{cases} \tag{A.1}$$

Since all variables in V' are new variables in the contexts Γ and Δ, we can assume that for any $z' \in V'$, $s(z') = s'(z')$ holds. Because if it does not, actually we can consider a new state s'', whose value differs from s only on those variables in V'. Obviously $s'' \models_{cdl} \bigwedge_{\varphi_1 \in \Gamma} \varphi_1$. If $s'' \models_{cdl} P_3 \vee \bigvee_{\varphi_2 \in \Delta} \varphi_2$, we also have $s \models_{cdl} P_3 \vee \bigvee_{\varphi_2 \in \Delta} \varphi_2$.

Since $s \models_{cdl} \bigwedge_{\varphi_1 \in \Gamma} \varphi_1$, from (A.1) we can get $s' \models_{cdl} \bigwedge_{\varphi_1 \in \Gamma} \varphi_1[V'/V] \wedge P_1$. So $s' \models_{cdl} P_2 \vee \bigvee_{\varphi_2 \in \Delta} \varphi_2[V'/V]$ from the assumption of (i). The situation for $s' \models_{cdl} \bigvee_{\varphi_2 \in \Delta} \varphi_2[V'/V]$ is obvious, because if so then $s \models_{cdl} \bigvee_{\varphi_2 \in \Delta} \varphi_2$. If $s' \models_{cdl} P_2$, since from (A.1) easy to see $ss' \in val((\varsigma^c!e|x := e'|.....))$, according to the semantics of P_3 in Def. 3.9 we have $s \models_{cdl} P_3$.

For (ii), for any $s \in S$, if $s \models_{cdl} \bigwedge_{\varphi_1 \in \Gamma} \varphi_1[V'/V] \wedge P_1$, we need to show that $s \models_{cdl} P_2 \vee \bigvee_{\varphi_2 \in \Delta} \varphi_2[V'/V]$.

To this end we construct an s' such that

$$\begin{cases} s'(c^n) = s((c^n)'), s'(c^s) = s((c^s)'), & \\ s'(x) = s(x'), & \\ \ldots\ldots & \\ s'(d_1^s) = s((d_1^s)'), \ldots, s'(d_n^s) = s((d_n^s)'), & \\ s'(z') = s(z'), & \text{for each new variable } z' \in V' \\ s'(y) = s(y), & \text{for other variable } y \notin V \end{cases} \tag{A.2}$$

where $(c^n)'$, $(c^s)'$, x',.....,$(d_1^s)'$,...,$(d_n^s)'$ are the new variables in V' corresponding to c^n, c^s, x,.....,d_1^s,...,d_n^s in V. Since $\models_{cdl} \bigwedge_{\varphi_1 \in \Gamma} \varphi_1[V'/V]$, from (A.2) we get $s' \models_{cdl} \bigwedge_{\varphi_1 \in \Gamma} \varphi_1$. So $s' \models_{cdl} P_3 \vee \bigvee_{\varphi_2 \in \Delta} \varphi_2$ from the assumption of (ii). If $s' \models_{cdl} \bigvee_{\varphi_2 \in \Delta} \varphi_2$, then obviously $s \models_{cdl} \bigvee_{\varphi_2 \in \Delta} \varphi_2[V'/V]$ from (A.2). If $s' \models_{cdl} P_3$, since $s \models_{cdl} P_1$, from (A.2) easy to see that $s's \in val(\alpha)$. According to the semantics of P_3 in Def. 3.9 we have $s \models_{cdl} P_2$.

2. For rule $(P?[])$, we consider $A = \phi$ for example. It is sufficient to prove that for all state $s \in S$, $s \models_{cdl} P \to [p]\phi$ iff $s \models_{cdl} [P?p]\phi$. For the 'if' direction, since $s \models_{cdl} [P?p]\phi$, based on the semantics of $[P?p]\phi$ given in Def. 3.18, we know for any finite trace $tr \in val(P?p)$ starting from s, $tr_e \in val(\phi)$ holds. But from Def. 3.9, $tr \in val(P?p)$ means if $tr_b \in val(P)$, then $tr \in val(p)$. So if $s \models_{cdl} P$, we have $s \models_{cdl} [p]\phi$. For the 'only if' direction, the condition when $s \not\models_{cdl} P$ is trivial, since in this case there exists no trace $tr \in val(P?p)$ starting from s, so $s \in val([P?p]\phi)$ always holds. If $s \models_{cdl} P$ and $s \models_{cdl} [p]\phi$, we know that any trace $tr \in val(p)$ starting from s is a trace of $val(P?p)$ too. So clearly $s \models_{cdl} [P?p]\phi$.

   According to Def. 3.17, a trace with length 1 always satisfies a term ξ, thus rule $(\pi\mu[])$ is sound.

   In rule $(\mu[])$, for any $s \in S$, since $val(\mu) = S$, thus $s \in val(\mu)$. So we have $s \models_{cdl} \phi$ iff $s \models_{cdl} [\mu]\phi$.

3. The soundness of rule $(\pi[\cup])$ is obvious, we omit its proof.

   Rule $(\pi[\bullet]u)$ holds because of Def. 3.17, traces with length 1 always satisfy a relation ξ, so it is enough to consider traces with length ≥ 2, which amounts to the traces of $p; p^\bullet$.

   The proofs of rule $(\pi[;])$ and rule $(\pi[\bullet]i)$ require Prop. A.1 and Prop. A.2 given below.

   For rule $(\pi[;])$, we prove for any $s \in S$, $s \models_{cdl} [p;q]\xi$ iff $s \models_{cdl} [p]\xi \wedge [p][q]\xi$. On the one hand, for any trace $tr \in val(p;q)$ with $tr_b = s$, by the semantics of $p;q$ (Def. 3.9) easy to see that there exists $tr_1 \in val(p)$, $tr_2 \in val(q)$ such that $tr = tr_1 \circ tr_2$. Because $s \models_{cdl} [p]\xi$ and $s \models_{cdl} [p][q]\xi$ hold, so $tr_1 \models_{cdl} \xi$ and $tr_2 \models_{cdl} \xi$ hold. From Prop. A.1 we know $tr \vDash_{ccsl} \xi$. So $s \models_{cdl} [p;q]\xi$. On the other hand, for any trace $tr \in val(p)$ with $tr_b = s$ and trace $tr' \in val(q)$ with $tr'_b = tr_e$, let $tr'' = tr \circ tr'$, easy to see that $tr'' \in val(p;q)$. Since $s \models_{cdl} [p;q]\xi$, there is $tr'' \models_{cdl} \xi$. Again from Prop. A.1 we can get $tr \models_{cdl} \xi$ and $tr' \models_{cdl} \xi$ hold. Therefore $s \models_{cdl} [p]\xi \wedge [p][q]\xi$.

   For rule $(\pi[\bullet]i)$, we prove for any $s \in S$, $s \models_{cdl} [p^\bullet]\xi$ iff $s \models_{cdl} [p^\bullet][p]\xi$. On the one hand, for any trace $tr \in val(p^\bullet)$ with $tr_b = s$, we consider three situations. (1) If the length of tr is 1, then obviously $tr \vDash_{ccsl} \xi$ due to Def. 3.17. (2) If there is a constant $n \geq 1$ such that $tr = tr_1 \circ tr_2 \circ \ldots \circ tr_n \in val(p^n)$, where $tr_1, \ldots, tr_{n-1} \in val(p)$ is finite, $tr_n \in val(p)$, since $s \models_{cdl} [p^\bullet][p]\xi$, for any trace $tr' \in val(p^\bullet)$ with $tr'_b = s$ and any trace tr'' with $tr''_b = tr'_e$, $tr'' \vDash_{ccsl} \xi$ holds. Let $tr' = \mu$, $tr'' = tr_1$, $tr' = tr_1$, $tr'' = tr_2$, $tr' = tr_1 \circ tr_2$, $tr'' = tr_3$, ..., $tr' = tr_1 \circ \ldots \circ tr_{n-1}$ and $tr'' = tr_n$ respectively, then we can get $tr_1 \vDash_{ccsl} \xi$, $tr_2 \vDash_{ccsl} \xi$, ..., $tr_n \vDash_{ccsl} \xi$ respectively. From Prop. A.1 $tr \vDash_{ccsl} \xi$ is immediately obtained. (3) If $tr = \underbrace{tr_1 \circ tr_2 \circ \ldots}_{\infty}$, where $tr_i \in val(p)$ $(i \in \mathbb{N}^+)$ is finite, then according to Prop. A.2, the proof can be attributed to situation (2). In all three situations (1), (2), (3)above, we have $s \models_{cdl} [p^\bullet]\xi$. On the other hand, for any trace $tr \in val(p^\bullet)$ with $tr_b = s$, and trace $tr' \in val(p)$ with $tr' = tr_e$, let

$tr'' = tr \circ tr'$. Because $tr' \in val(p^\bullet; p)$ and $s \models_{cdl} [p^\bullet]\xi$, $tr'' \vDash_{ccsl} \xi$ holds. From Prop. A.1 there are $tr \vDash_{ccsl} \xi$ and $tr' \vDash_{ccsl} \xi$, so $s \models_{cdl} [p^\bullet][p]\xi$.

□

Proposition A.1 (Temporal Property of ξ)**.** *Any clock relation ξ satisfies the following property: for any traces tr, tr_1 and tr_2 such that $tr = tr_1 \circ tr_2$, where tr_1 is finite, there is*

$$tr \vDash_{ccsl} \xi \quad \textit{iff} \quad tr_1 \vDash_{ccsl} \xi \textit{ and } tr_2 \vDash_{ccsl} \xi.$$

Prop. A.1. According to Def. 3.17, $tr \vDash_{ccsl} \xi$ iff for any $i \in \mathbb{N}^+$, $tr(i) \models_{cdl} \hbar(\xi)$. Thus for any $i \in \mathbb{N}^+$, if $i \leq |tr_1|$, then $tr_1(i) \models_{cdl} \hbar(\xi)$, if $i > |tr_1|$, then $tr_2(i) \models_{cdl} \hbar(\xi)$, which means $tr_1 \vDash_{ccsl} \xi$ and $tr_2 \vDash_{ccsl} \xi$ hold. □

A.2. The Proof of Theorem 6.2

In the following proofs, we use $\phi^\flat$ to stress that ϕ is an AFOL formula.

The main body of Theorem 6.2. For any formula ϕ, by the soundness of the FOL rules (in Table 5), we can convert ϕ into a conjunctive normal form: $C_1 \wedge ... \wedge C_n$. Each clause C_i is a disjunction of literals: $C_i = l_{i,1} \vee ... \vee l_{i,m_i}$, where $l_{i,j}$ $(1 \leq i \leq n, 1 \leq j \leq m_i)$ is an atomic CDL formula, or its negation. By the FOL rules, it is sufficient to prove that for each clause C_i, $\models_{cdl} C_i$ implies $\vdash^+_{cdl} C_i$. We proceed by induction on the sum n, of the number of the appearances of $[p]$ and $\langle p \rangle$ and the number of quantifiers $\forall x$ and $\exists x$ prefixed to non-AFOL formulae in C_i.

If $n = 0$, there are no appearances of $[p]$ and $\langle p \rangle$ in C_i, so C_i is an AFOL formula, by condition (i) of Theorem 6.2, we obtain $\vdash^+_{cdl} C_i$.

Suppose $n > 0$, it is sufficient to consider the following cases:

$$C_i = \varphi_1 \vee op\, \varphi_2, C = \varphi \vee [p]\xi, C = \varphi \vee \langle p \rangle \sim \xi$$

where $op \in \{[p], \langle p \rangle, \forall x, \exists x\}$.

If $C_i = \varphi_1 \vee op\, \varphi_2$, which is equivalent to $\neg\varphi_1 \to op\, \varphi_2$, by condition (i), there exist $\phi^\flat_1$ and $\phi^\flat_2$ such that $\models_{cdl} \phi^\flat_1 \leftrightarrow \neg\varphi_1$ and $\models_{cdl} \phi^\flat_2 \leftrightarrow \varphi_2$. Then by the soundness of the FOL rules $\models_{cdl} \phi^\flat_1 \to op\, \phi^\flat_2$ holds. By condition (ii) we have

$$\vdash^+_{cdl} \phi^\flat_1 \to op\, \phi^\flat_2. \tag{A.3}$$

Since in $\phi^\flat_1 \leftrightarrow \neg\varphi_1$ and $\phi^\flat_2 \leftrightarrow \varphi_2$ the sum is strictly less than n, by inductive hypothesis we can get

$$\vdash^+_{cdl} \neg\varphi_1 \to \phi^\flat_1 \tag{A.4}$$

and $\vdash^+_{cdl} \phi^\flat_2 \to \varphi_2$ hold. By condition (iii) we know that

$$\vdash^+_{cdl} op\, \phi^\flat_2 \to op\, \varphi_2 \tag{A.5}$$

holds. Based on (A.3), (A.4), (A.5) and the FOL rules in Table 5 we can conclude that $\vdash^+_{cdl} \neg\varphi_1 \to op\, \varphi_2$.

If $C_i = \varphi \vee [p]\xi$, we prove the completeness of the formula $\neg\varphi \to [p]\xi$. By condition (i) of Theorem 6.2 there exists an AFOL formula $\phi^\flat_0$ such that $\models_{cdl} \phi^\flat \leftrightarrow \neg\varphi$. Note that in $\phi^\flat \leftrightarrow \neg\varphi$ the sum is strictly less than n, by inductive hypothesis we have $\vdash^+_{cdl} \neg\varphi \to \phi^\flat$. By condition (iv), $\vdash^+_{cdl} \phi^\flat \to [p]\xi$ holds. Thus we have $\vdash^+_{cdl} \neg\varphi \to [p]\xi$.

Similarly for the case $C_i = \varphi \vee \langle p \rangle \sim \xi$. □

Before giving the proofs of Theorem 6.2 (i), (ii) and (iii), we define a finite fragment of SEP, called 'Finite-SEP'.

Definition A.1 (Finite-SEP)**.** *The finite fragment of SEP, called 'Finite-SEP', is defined as follows:*

$$p ::= \mu \mid \alpha \mid \varrho \& P ? \alpha \mid p; p \mid p \cup p \mid p^* \mid \ \|(p_1, ..., p_n),$$

where μ, α, ϱ and P are defined as in Def. 3.1.

The semantics of Finite-SEPs is defined the same as in Def. 3.9 and 3.14, except for p^*, whose semantics is given as below:

$$val(p^*) =_{df} \bigcup_{n \geq 0} val^n(p).$$

In CDL formulae, there is an important relationship between an SEP and its finite fragment. We state it as the next proposition. Intuitively, it says that for any specification of a program that a CDL formula can describe, it is enough to only consider the finite fragment of this program.

Proposition A.2 (Relation between SEP and Finite-SEP in CDL)**.** *For any SEP p, we denote the Finite-SEP program, which is formed by replacing all loop programs of the form $q^\bullet$ in p with program q^*, as $finite(p)$. For any term ξ and CDL formula ϕ, the following two relations hold:*

$$\models_{cdl} [finite(p)]\xi \leftrightarrow [p]\xi \ \ and \ \ \models_{cdl} [finite(p)]\phi \leftrightarrow [p]\phi.$$

The proof of Prop A.2 relies on the following Lemmas A.1 and A.2.

Prop.A.2. We take $[p]\xi$ as an example, $[p]\phi$ is similar. On the one hand, if $\models_{cdl} [p]\xi$, for any trace $tr \in val(finite(p))$, according to Lemma A.1 (i) there is $tr \in val(p)$, so $tr \models_{ccsl} \xi$. On the other hand, if $\models_{cdl} [finite(p)]\xi$, for any trace $tr \in val(p)$, we analyze two situations. (1) If tr is finite, then according to Lemma A.1 (ii) we have $tr \in val(finite(p))$, so $tr \models_{ccsl} \xi$. (2) If tr is infinite, suppose $tr \not\models_{ccsl} \xi$, then according to Def. 3.17, there must exist a state s in tr such that $s \models_{cdl} \neg\hbar(\xi)$. According to Lemma A.2, there exists a finite trace tr' in p containing s, so $tr' \not\models_{ccsl} \xi$. Since tr' is also a trace of $finite(p)$ (Lemma A.1 (ii)), but this contradicts the assumption $\models_{cdl} [finite(p)]\xi$. Thus $tr \models_{ccsl} \xi$. □

The lemmas used in the proof of Prop. A.2 above is given below. Lemma A.1 says Finite-SEP exactly captures all finite behaviour of SEP. Lemma A.2 reveals that for any SEP and one of its traces, all finite prefixed traces of this trace are traces of this SEP.

Lemma A.1. *The semantics of any SEP p and the semantics of its corresponding finite fragment $finite(p)$ satisfy the following propositions:*

(i) $val(finite(p)) \subseteq val(p)$.

(ii) For any trace $tr \in val(p)$, if tr is finite, then $tr \in val(finite(p))$.

Lemma A.1. We proceed by induction on the structure of program p. In sequential SEPs, the only non-trivial case is $p = q^\bullet$. For parallel programs of the form $p = \|(p_1, ..., p_n)$, we can first reduce it into an equivalent sequential program by applying the rewrite rules in Table 6, then we prove this sequential program by induction.

If $p = q^\bullet$, by the definition of the function *finite*, there is $finite(p) = finite(q)^*$. In proposition (i), for any trace tr of $finite(q)^*$, if $|tr| = 1$ then obviously $tr \in val(q^\bullet)$ because $val(\mu) \subseteq val(q^\bullet)$. If $|tr| > 1$, then there must exist an $n \geq 1$ such that $tr = tr_1 \circ tr_2 \circ ... \circ tr_n$, where $tr_1, ..., tr_n \in val(finite(q))$. By inductive hypothesis we know $tr_1 \in val(q)$, ..., $tr_n \in val(q)$. So by Def. 3.9 we obtain $tr \in val(q^n) \subset val(q^\bullet)$. In proposition(ii), for any finite trace tr of $q^\bullet$, if $|tr| = 1$ obviously $tr \in val(q^\bullet)$. If $|tr| > 1$, there must be a number $n \geq 1$ such that $tr = tr_1 \circ tr_2 \circ ... \circ tr_n$, where $tr_1, ..., tr_n \in val(q)$ must be finite. According to the inductive hypothesis there is $tr_1, ..., tr_n \in val(finite(q))$, so $tr \in val(finite(q)^n) \subset val(finite(q))$.

□

Lemma A.2. *Given an SEP p and a trace tr of p, for any state s in tr, there exists a finite trace of p that contains s.*

Lemma A.2. We proceed by induction on the structure of p. Same as in the proof of Lemma A.1, the only non-trivial case is $p = q^\bullet$. We omit the case when p is a parallel program since it can be reduced into a sequential one.

If $p = q^\bullet$, let tr be any trace of p. The case when tr is finite is trivial. So we assume tr is infinite. According to the semantics of $q^\bullet$, tr can be one of the two forms as follows: (1) $tr = tr_1 \circ tr_2 \circ ...$, where $tr_i \in val(q)$ is finite ($i \in \mathbb{N}^+$); (2) there exists a number $n \geq 1$ such that $tr = tr_1 \circ ... \circ tr_n$, where $tr_1, ..., tr_{n-1}$ is finite, $tr_n \in val(q)$. If a state s is in one of the finite trace tr_i ($i \in \mathbb{N}^+$), we let $tr' = tr_1 \circ ... \circ tr_i$. Obviously tr' is a finite trace, and we have $tr' \in val(q^i) \subset val(q^\bullet)$. If a state s is in trace tr_n of form (2), then by inductive hypothesis there exists a finite trace $tr'' \in val(q)$ that contains s. Now let $tr' = tr_1 \circ ... \circ tr_{n-1} \circ tr''$. Obviously tr' is finite, and more, we have $tr' \in val(q^n) \subset val(q^\bullet)$.

□

In order to prove Theorem 6.2 (i), we first show any Finite-SEP is expressible in AFOL (Lemma A.3), then based on the expressibility of Finite-SEPs, we prove any formula $[p]\phi^\flat$ (where p is a Finite-SEP) is expressible (Lemma A.5). At last we obtain the proof of Theorem 6.2 (i).

The next lemma shows every Finite-SEP (except for μ) can be expressed as an AFOL formula. The skip program μ is special, since in fact it has no behaviour.

Lemma A.3 (Construction of Formula $A_p(\vec{v}, \vec{v}')$)**.** *For any Finite-SEP p, if $p \neq \mu$, then we can construct an AFOL formula $A_p(\vec{v}, \vec{v}')$, where $\vec{v} = \mathcal{V}(p)$, $\vec{v}'$ is a vector of new variables corresponding to $\vec{v}$. It satisfies the following conditions:*

(i) For any state s and vector $\vec{x}_0$, if $s \models_{cdl} A_p(\vec{v}, \vec{x}_0)$, then there exists a trace $tr \in val(p)$ such that $tr_b = s$ and $tr_e(\vec{v}) = \vec{x}_0$.

(ii) For any trace $tr \in val(p)$, $tr_b \models_{cdl} A_p(\vec{v}, \vec{v}')[tr_e(\vec{v})/\vec{v}']$ holds.

Given a state s and a vector $\vec{v} = \langle v_1, ..., v_n \rangle$, we define $s(\vec{v}) =_{df} \langle s(v_1), ..., s(v_n) \rangle$.

Before proving Lemma A.3, we need to introduce the concept of Gödel encoding. We use this concept to help us build AFOL formulae for finite loop program p^*, just like what has been done in [20, 32]. Here we use a lemma to illustrate that some concepts we need in the proof of Lemma A.3 can be described by formulae constructed by Gödel encoding. And we assume the existence of these formulae, without giving any hint about how to construct them. The details of the construction is outside of the scope of this paper, interesting readers can refer to [46] for more details.

Lemma A.4 (Gödel Encoding)**.** *In AFOL, there exist ways to encode a vector $(x_1, ..., x_n)$ (where $x_1, ..., x_n \in \mathbb{Z}$) as a unique number $Z \in \mathbb{Z}$, denoted as $Z = (x_1, ..., x_n)^{\mathfrak{G}}$. Z is called the* Gödel number *of $(x_1, ..., x_n)$. The ways of encoding are called* Gödel encodings.

Let $Z = (x_1, ..., x_n)^{\mathfrak{G}}$. In AFOL, a formula $IEl(Z, i, y)$ can be constructed through Gödel encoding in order to judge: whether the ith element of $(x_1, ..., x_n)$ is y, i.e., $IEl(Z, i, x)$ is true iff $(x_1, ..., x_n)_i = y$. A formula $Len(Z, N)$ can be constructed through Gödel encoding in order to judge: whether the length of vector $(x_1, ..., x_n)$ is N, i.e., $Len(Z, N)$ is true iff $N = n$.

Readers can refer to [46] for an explicit definition of *IEl* and *Len* given above[4].

Lemma A.3. We proceed by induction on the structure of the Finite-SEP p. We only consider the cases when p semantically equals to the following forms. Programs in other forms can be first reduced into the following forms by laws in Prop. 3.2 and rewrite rules in Table 6.

1. If $p \equiv P?\alpha$ or $p \equiv \alpha$, we take $p \equiv P?\alpha$ as an example, $p \equiv \alpha$ is similar. Without the lose of generality set $\alpha = (\varsigma^c!e|x := e')$, then we define

$$A_\alpha(\vec{v}, \vec{v}') = P \wedge (c^n)' = c^n + 1 \wedge (c^s)' = 1 \wedge x' = e' \wedge \bigwedge_{1 \leq i \leq n} (d_i^s)' = 0,$$

[4]where the corresponding names are "*IthElement*" and "*Length*" respectively.

where $\vec{v} = \{c^n, c^s, d_1^s, ..., d_n^s\}, \{d_1, ..., d_n\} = \mathcal{C} - \mathcal{C}(\alpha)$. $\vec{v}' = \{(c^n)', (c^s)', (d_1^s)', ..., (d_n^s)'\}$ is the vector of new variables corresponding to $\vec{v}$.

For any state s and vector $\vec{x}_0$, if $s \models_{cdl} A_\alpha(\vec{v}, \vec{x}_0)$, let

$$\begin{cases} s'(\vec{v}) = \vec{x}_0, \\ s'(z) = s(z), \quad \text{other variable } z \end{cases}$$

then there is $s \models_{cdl} P \wedge s'(c^n) = c^n + 1 \wedge s'(c^s) = 1 \wedge s'(x) = e' \wedge \bigwedge_{1 \le i \le n} s'(d_i^s) = 0$. According to the semantics of $P?\alpha$ (Def. 3.9), we can get $ss' \in val(P?\alpha)$.

On the other hand, for any trace $uu' \in val(P?\alpha)$, by the semantics of $P?\alpha$ we have:

(1) $u \in val(P)$;

(2) $u'(c^n) = u(c^n) + 1 \wedge u'(c^s) = 1$ and $u'(x) = Eval_u(e')$;

(3) $u'(d^n) = u(d^n) \wedge u'(d^s) = 0$ for other $d \in \mathcal{C}$ not in α;

(4) $u'(z) = u(z)$ for other variable z not in α.

Thus easy to get $u \models_{cdl} A_\alpha(\vec{v}, \vec{v}')[u'(\vec{v})/\vec{v}']$.

2. If $p \equiv q \cup r$, let

$$A_p(\vec{v}, \vec{v}') = A_q(\vec{v}, \vec{v}') \vee A_r(\vec{v}, \vec{v}'),$$

where $\vec{v} = \mathcal{V}(q \cup r)$, $\vec{v}'$ is a vector of new variables corresponding to $\vec{v}$. In the formula $A_q(\vec{v}, \vec{v}')$ (the same for $A_r(\vec{v}, \vec{v}')$), we can assume $\vec{v} = \mathcal{V}(q \cup r) = \mathcal{V}(q)$. Because if $\mathcal{V}(q) \subset \vec{v}$, we can actually build a new formula as $A'_q(\vec{v}, \vec{v}') = A_q(\vec{v}_q, \vec{v}'_q) \wedge \bigwedge_{x \in \vec{v} - \vec{v}_q, x' \in \vec{v} - \vec{v}'_q} x = x'$, where in $A_q(\vec{v}_q, \vec{v}'_q)$ we have $\vec{v}_q = \mathcal{V}(q)$. For some variable x not appearing in q, in $A'_q(\vec{v}, \vec{v}')$ there is $x = x'$. x' is a new variable corresponding to x. Then we can consider $A_p(\vec{v}, \vec{v}') = A'_q(\vec{v}, \vec{v}') \vee A'_r(\vec{v}, \vec{v}')$. If $q \equiv \mu$ (the same for $r \equiv \mu$), let $A_q(\vec{v}, \vec{v}') = \bigwedge_{x \in \vec{v}, x' \in \vec{v}'} x = x'$.

For any state s and vector $\vec{x}_0$, if $s \models_{cdl} A_p(\vec{v}, \vec{x}_0)$, then according to the soundness of the FOL rules (see Table 5) there is $s \models_{cdl} A_q(\vec{v}, \vec{x}_0)$ or $s \models_{cdl} (A_r(\vec{v}, \vec{x}_0)$ holds. Assume $s \models_{cdl} A_q(\vec{v}, \vec{x}_0)$, by inductive hypothesis there exists a trace $tr \in val(q)$ such that $tr_b = s$ and $tr_e(\vec{v}) = \vec{x}_0$. So by Def. 3.9 obviously $tr \in val(q \cup r)$.

On the other hand, for any trace $tr \in val(q \cup r)$, based on Def. 3.9 there is $tr \in val(q)$ or $tr \in val(r)$. Assume $tr \in val(q)$, by inductive hypothesis we know $tr_b \models_{cdl} A_q(\vec{v}, \vec{v}')[tr_e(\vec{v})/\vec{v}']$. So $tr_b \models_{cdl} A_q(\vec{v}, \vec{v}')[tr_e(\vec{v})/\vec{v}'] \vee A_r(\vec{v}, \vec{v}')[tr_e(\vec{v})/\vec{v}']$, i.e., $tr_b \models_{cdl} A_p(\vec{v}, \vec{v}')[tr_e(\vec{v})/\vec{v}']$.

3. If $p \equiv q; r$ and $q \equiv \mu$, then define $A_p = A_r$, similarly for $r \equiv \mu$. Now we assume $q \not\equiv \mu$ and $r \not\equiv \mu$. Let

$$A_p(\vec{v}, \vec{v}') = \exists \vec{z}.(A_q(\vec{v}, \vec{v}')[\vec{z}/\vec{v}'] \wedge A_r(\vec{v}, \vec{v}')[\vec{z}/\vec{v}]),$$

where $\vec{v} = \mathcal{V}(q \cup r)$, $\vec{v}'$ is the vector of new variables corresponding to $\vec{v}$. $\vec{z}$ is the vector of new variables corresponding to $\vec{v}$ and $\vec{v}'$. Like in the case $p \equiv q \cup r$ above, In the formula $A_q(\vec{v}, \vec{v}')$ (the same for $A_r(\vec{v}, \vec{v}')$), we can assume $\vec{v} = \mathcal{V}(q \cup r) = \mathcal{V}(q)$. Because if not so, let $\vec{v}_q = \mathcal{V}(q)$, we can build a new formula $A'_q(\vec{v}, \vec{v}') = A_q(\vec{v}_q, \vec{v}'_q) \wedge \bigwedge_{x \in \vec{v} - \vec{v}_q, x' \in \vec{v}' - \vec{v}'_q} x = x'$, then we can consider $A_p(\vec{v}, \vec{v}') = \exists \vec{z}.(A'_q(\vec{v}, \vec{v}')[\vec{z}/\vec{v}'] \wedge A'_r(\vec{v}, \vec{v}')[\vec{z}/\vec{v}])$ instead.

For any state s and vector $\vec{x}_0$, if $s \models_{cdl} A_p(\vec{v}, \vec{x}_0)$, then there exists a vector $\vec{z}_0$ such that

$$s \models_{cdl} A_q(\vec{v}, \vec{x}_0)[\vec{z}_0/\vec{x}_0] \text{ and } s \models_{cdl} A_r(\vec{v}, \vec{x}_0)[\vec{z}_0/\vec{v}]. \tag{A.6}$$

From (A.6) and inductive hypothesis we know there exists a trace $tr \in val(q)$ such that

$$tr_b = s \text{ and } tr_e(\vec{v}) = \vec{z}_0. \tag{A.7}$$

Let $s' = tr_e$. Because $\vec{v} = \mathcal{V}(q)$, so for other variable $x \notin \vec{v}$ we always have $s'(x) = s(x)$. According to the semantics of q and (A.6) there is $s' \models_{cdl} A_r(\vec{v}, \vec{x}_0)$. By inductive hypothesis, there exists a trace $tr' \in val(r)$ such that

$$tr'_b = s' \text{ and } tr'_e(\vec{v}) = \vec{x}_0. \tag{A.8}$$

From (A.7) and (A.8) there exists a trace $tr'' = tr \circ tr' \in val(q; r)$ that satisfies $tr''_b = s$ and $tr''_e(\vec{v}) = \vec{x}_0$. On the other hand, given a trace $tr \in val(q; r)$, based on the semantics of SEPs (Def. 3.9), there exist traces $tr' \in val(q)$ and $tr'' \in val(r)$ such that $tr = tr' \circ tr''$. Let $s = tr'_b$, $s'' = tr'_e = tr''_b$ and $s' = tr''_e$. By inductive hypothesis, we have

$$s \models_{cdl} A_q(\vec{v}, \vec{v}')[s''(\vec{v})/\vec{v}'] \text{ and } s'' \models_{cdl} A_r(\vec{v}, \vec{v}')[s'(\vec{v})/\vec{v}']. \tag{A.9}$$

Let $\vec{z}_0 = s''(\vec{v})$, from (A.9), easy to see that

$$s \models_{cdl} A_q(\vec{v}, \vec{z}_0) \text{ and } s \models_{cdl} A_r(\vec{z}_0, \vec{v}')[s'(\vec{v})/\vec{v}']. \tag{A.10}$$

By (A.10) and the soundness of the FOL rules in Table 5, we get

$$s \models_{cdl} \exists \vec{z}.(A_q(\vec{v}, \vec{v}')[\vec{z}/\vec{v}'] \wedge A_r(\vec{v}, \vec{v}')[\vec{z}/\vec{v}][s'(\vec{v})/\vec{v}']),$$

i.e., $s \models_{cdl} A_p(\vec{v}, \vec{v}')[s'(\vec{v})/\vec{v}']$.

4. If $p \equiv q^*$ (and $q \not\equiv \mu$), let

$$A_p(\vec{v}, \vec{v}') = \exists n > 0.Iter(n) \vee Iter(0), \tag{A.11}$$

where we define $Iter(0) = \bigwedge_{x \in \vec{v}, x' \in \vec{v}'} x = x'$. $\vec{v} = \mathcal{V}(p)$, $\vec{v}'$ is the vector of new variables corresponding to $\vec{v}$. $Iter(n)(n \geq 1)$ is defined as an AFOL formula based on the Gödel encoding:

$$\begin{aligned}
&Iter(n) = \exists Z.\exists V_1 V_2.(\\
&Len(Z, n+1) \wedge IEl(Z, 1, V_1) \wedge IEl(Z, n+1, V_2) \wedge Len(V_1, N) \wedge Len(V_2, N) \wedge \\
&\quad \bigwedge_{k=0}^{N} IEl(V_1, k, \vec{v}_k) \wedge \bigwedge_{k=0}^{N} IEl(V_2, k, \vec{v}'_k) \wedge \forall (1 \leq i \leq n).\exists Y_1 Y_2.(\\
&\quad\quad IEl(Z, i, Y_1) \wedge IEl(Z, i+1, Y_2) \wedge Len(Y_1, N) \wedge Len(Y_2, N) \wedge \exists \vec{X}_1 \vec{X}_2.(\\
&\quad\quad\quad \bigwedge_{k=1}^{N} IEl(Y_1, k, \vec{X}_{1,k}) \wedge \bigwedge_{k=1}^{N} IEl(Y_2, k, \vec{X}_{2,k}) \wedge A_q(\vec{X}_1, \vec{X}_2)\\
&\quad\quad)\\
&\quad)\\
&),
\end{aligned} \tag{A.12}$$

where $N = |\vec{v}|$, Z is a Gödel number, V_1, V_2, Y_1, Y_2 are variables, $\vec{X}_1$ and $\vec{X}_2$ are vectors of variables. Formulae *IEl* and *Len* are defined in Lemma A.4. $Iter(0)$ expresses the behaviour of program q^* when it executes 0 time (i.e., the behaviour of μ). Intuitively, formula $Iter(n)$ $(n \geq 1)$ just captures the meaning of the formula $A_{q^n}(\vec{v}, \vec{v}')$:

$$A_{q^n}(\vec{v}, \vec{v}') = \exists \vec{z}_1 \vec{z}_2 ... \vec{z}_{n-1}.(A_q(\vec{v}, \vec{z}_1) \wedge A_q(\vec{z}_1, \vec{z}_2) \wedge ... \wedge A_q(\vec{z}_{n-1}, \vec{v}')).$$

In $Iter(n)$, Z is the Gödel number represents the vector $(W_1, ..., W_{n+1})$ of length $n+1$, i.e., $Z = (W_1, ..., W_{n+1})^{\mathfrak{G}}$. $W_1, ..., W_{n+1}$ are Gödel numbers. W_1 and W_{n+1} represent vectors $\vec{v}$ and $\vec{v}'$ in

$A_{q^n}(\vec{v},\vec{v}')$ respectively, i.e., $W_1 = \vec{v}^{\mathfrak{G}}$ and $W_{n+1} = \vec{v'}^{\mathfrak{G}}$. $W_2, ..., W_n$ represent vectors $\vec{z}_1$, ..., $\vec{z}_{n-1}$ in $A_{q^n}(\vec{v},\vec{v}')$ respectively, i.e., $W_2 = \vec{z}_1^{\mathfrak{G}}$, $W_3 = \vec{z}_2^{\mathfrak{G}}$, ..., $W_n = \vec{z}_{n-1}^{\mathfrak{G}}$. Variables V_1 and V_2 represent the Gödel number W_1 and W_{n+1} respectively. Variable i traverses all elements in the vector $(W_1, ..., W_{n+1})$. For each value of i, variables Y_1 and Y_2 represent the Gödel numbers W_i and W_{i+1} respectively. The length of vectors $\vec{X}_1$ and $\vec{X}_2$ is N. They represent the element of the vector represented by the numbers W_i and W_{i+1} respectively.

If we take n as a constant, then actually $A_{q^n}(\vec{v},\vec{v}')$ is an AFOL formula. From (A.12), Lemma A.4 and the soundness of the FOL rules (see Table 5), it is not hard to prove (omitted here) $Iter(n)$ is logical equivalent to $A_{q^n}(\vec{v},\vec{v}')$, i.e.,

$$\models_{cdl} Iter(n) \leftrightarrow A_{q^n}(\vec{v},\vec{v}') \tag{A.13}$$

Now we prove conditions (i) and (ii) of Lemma A.3. For any state s, if $s \models_{cdl} A_p(\vec{v},\vec{x}_0)$, there are two cases: (1) $s \models_{cdl} Iter(0)(\vec{v},\vec{x}_0)$; (2) $s \models_{cdl} (\exists n.Iter(n))(\vec{v},\vec{x}_0)$. In case (1), there is $s \models_{cdl} \bigwedge_{x\in\vec{v},y\in\vec{x}_0} x = y$, i.e., $s(\vec{v}) = \vec{x}_0$. Let $tr = s \in val(\mu) \subseteq val(p)$, obviously, tr satisfies $tr_b = s$ and $tr_e(\vec{v}) = \vec{x}_0$. In case (2), since $s \models_{cdl} (\exists n.Iter(n))(\vec{v},\vec{x}_0)$, there exists a number $n_0 \geq 1$ such that $s \models_{cdl} Iter(n_0)(\vec{v},\vec{x}_0)$. From (A.13), we also have $s \models_{cdl} A_{q^{n_0}}(\vec{v},\vec{x}_0)$, but this means there exists vectors $\vec{z}_1 ... \vec{z}_{n-1}$ such that

$$s \models_{cdl} A_q(\vec{v},\vec{z}_1) \wedge A_q(\vec{z}_1,\vec{z}_2) \wedge ... \wedge A_q(\vec{z}_{n-1},\vec{x}_0). \tag{A.14}$$

Based on (A.14) and inductive hypothesis, below we construct a trace $tr \in val(q^{n_0})$ that satisfies $tr_b = s$ and $tr_e(\vec{v}) = \vec{x}_0$. From (A.14) we know $s \models_{cdl} A_q(\vec{v},\vec{z}_1)$. By inductive hypothesis, there exists a trace $tr_1 \in val(q)$ such that $tr_{1,b} = s$ and $tr_{1,e}(\vec{v}) = \vec{z}_1$. Let $s_1 = tr_{1,e}$, according to the semantics of q (Def. 3.9), s_1 equals to s except for those values in $\vec{v}$. By $s \models_{cdl} A_q(\vec{z}_1,\vec{z}_2)$ in (A.14), there is $s_1 \models_{cdl} A_q(\vec{v},\vec{z}_2)$. Again by inductive hypothesis there exists a trace $tr_2 \in val(q)$ such that $tr_{2,b} = s_1$ and $tr_{2,e}(\vec{v}) = \vec{z}_2$. Let $s_2 = tr_{2,e}$,..., continuing this process as what we just did for s_1, finally we see that there exist $s_1 = tr_{1,e}$,...,$s_{n-1} = tr_{n-1,e}$ and $tr_1, ..., tr_n \in val(q)$ such that:

$$\begin{aligned}
&tr_{1,b} = s, tr_{1,e}(\vec{v}) = \vec{z}_1,\\
&tr_{2,b} = s_1, tr_{2,e}(\vec{v}) = \vec{z}_2,\\
&...,\\
&tr_{n-1,b} = s_{n-2}, tr_{n-1,e}(\vec{v}) = \vec{z}_{n-1},\\
&tr_{n,b} = s_{n-1}, tr_{n,e}(\vec{v}) = \vec{x}_0.
\end{aligned}$$

Let $tr = tr_1 \circ tr_2 \circ ... \circ tr_n \in val(q^{n_0}) \subset val(q^*)$, then obviously tr satisfies $tr_b = s$ and $tr_e(\vec{v}) = \vec{x}_0$.

On the other hand, for any trace $tr \in val(q^*)$, according to the semantics of q^* in Def. A.1, there are two cases: (1) $|tr| = 1$, which means $tr \in val(\mu)$; (2) $tr \in val(p;p^*)$. In case (1), let $s = tr$. It is easy to see that $s \models_{cdl} Iter(0)(\vec{v},\vec{v}')[s(\vec{v})/\vec{v}']$, i.e., $s \models_{cdl} \bigwedge_{x\in\vec{v},x'\in\vec{v}'} x = x'[s(\vec{v})/\vec{v}']$ holds. Thus we have $s \models_{cdl} A_p(\vec{v},\vec{v}')[s(\vec{v})/\vec{v}']$. In case (2), there exist trace $tr_1, ..., tr_m \in val(q)$ $(m \geq 1)$ such that $tr = tr_1 \circ tr_2 \circ ... \circ tr_m$. By inductive hypothesis, for each trace tr_k $(1 \leq k \leq m)$, there is $tr_{k,b} \models_{cdl} A_q(\vec{v},\vec{v}')[tr_{k,e}(\vec{v})/\vec{v}']$. Observe that $tr_{1,e}(\vec{v}) = tr_{2,b}(\vec{v})$, by Def. 3.9 we see that $tr_{2,b}$ agrees with $tr_{1,b}$ on all values but those in $\vec{v}$. So by $tr_{2,b} \models_{cdl} A_q(\vec{v},\vec{v}')[tr_{2,e}(\vec{v})/\vec{v}']$ we have $tr_{1,b} \models_{cdl} A_q(tr_{1,e}(\vec{v}),\vec{v}')[\vec{v}_{2,e}(\vec{v})/\vec{v}']$. Note that $tr_{2,e}(\vec{v}) = tr_{3,b}(\vec{v})$, similarly based on Def. 3.9 and $tr_{3,b} \models_{cdl} A_q(\vec{v},\vec{v}')[tr_{3,e}(\vec{v})/\vec{v}']$ there is $tr_{1,b} \models_{cdl} A_q(tr_{2,e}(\vec{v}),\vec{v}')[tr_{3,e}(\vec{v})/\vec{v}']$, ..., continuing this procedure, finally, since $tr_{1,e}(\vec{v}) = tr_{2,b}(\vec{v})$, ..., $tr_{m-1,e}(\vec{v}) = tr_{m,b}(\vec{v})$, we have:

$$\begin{aligned}
&tr_{1,b} \models_{cdl} A_q(\vec{v},\vec{v}')[tr_{1,e}(\vec{v})/\vec{v}'],\\
&tr_{1,b} \models_{cdl} A_q(tr_{1,e}(\vec{v}),\vec{v}')[tr_{2,e}(\vec{v})/\vec{v}'],\\
&...,\\
&tr_{1,b} \models_{cdl} A_q(tr_{m-1,e}(\vec{v}),\vec{v}')[tr_{m,e}(\vec{v})/\vec{v}'].
\end{aligned}$$

Hence $tr_{1,b} \models_{cdl} A_q(\vec{v}, tr_{1,e}(\vec{v})) \wedge A_q(tr_{1,e}(\vec{v}), tr_{2,e}(\vec{v})) \wedge ... \wedge A_q(tr_{m-1,e}(\vec{v}), \vec{v}')[tr_{m,e}(\vec{v})/\vec{v}']$. Let $\vec{z}_1 = tr_{1,e}(\vec{v})$, ..., $\vec{z}_{m-1} = tr_{m-1,e}(\vec{v})$. By the soundness of the FOL rules in Table 5 there is $tr_{1,b} \models_{cdl} \exists \vec{z}_1 ... \vec{z}_{m-1}.(A_q(\vec{v}, \vec{z}_1) \wedge A_q(\vec{z}_1, \vec{z}_2) \wedge ... \wedge A_q(\vec{z}_{m-1}, \vec{v})[tr_{m,e}(\vec{v})/\vec{v}'])$. From (A.13) we obtain the result: $tr_b \models_{cdl} A_p(\vec{v}, \vec{v}')[tr_e(\vec{v}')/\vec{v}']$.

□

Note that in the proof of Lemma A.3, $\exists x > 0.A_{q^n}(\vec{v}, \vec{v}')$ is not an AFOL formula, because the variable n appears in the subscript of '$\vec{z}_n$'. This is also why we need Gödel encoding to construct formula $Iter(n)$.

Based on Lemma A.3, we prove that all formulae of the form $[p]\phi^\flat$ with p being a Finite-SEP are expressible in AFOL.

Lemma A.5 (Expressibility of $[p]\phi^\flat$)**.** *For any CDL formula of the form $[p]\phi^\flat$ where p is a Finite-SEP, there exists an AFOL formula $\varphi^\flat$ such that*

$$\models_{cdl} \varphi^\flat \leftrightarrow [p]\phi^\flat.$$

Lemma A.5. We construct $\varphi^\flat$ as: $\varphi^\flat = \forall \vec{x}.(A_p(\vec{v}, \vec{x}) \rightarrow \phi^\flat[\vec{x}/\vec{v}])$, where $A_p(\vec{v}, \vec{x})$ is the formula constructed in Lemma A.3. Now we prove

$$\models_{cdl} \varphi^\flat \leftrightarrow [p]\phi^\flat.$$

This is equivalent to prove for any state s, $s \models_{cdl} \forall \vec{x}.(A_p(\vec{v}, \vec{x}) \rightarrow \phi^\flat[\vec{x}/\vec{v}])$ iff $s \models_{cdl} [p]\phi^\flat$.

On the one hand, suppose for any state s, $\forall \vec{x}.(A_p(\vec{v}, \vec{x}) \rightarrow \phi^\flat[\vec{x}/\vec{v}])$ holds. We need to prove that for any trace $tr \in val(p)$ with $tr_b = s$, $tr_e \models_{cdl} \phi^\flat$ holds. Let $s' = tr_e$. By Lemma A.3 (ii) we have $s \models_{cdl} A_p(\vec{v}, \vec{v}')[s'(\vec{v})/\vec{v}']$. But this means $s \models_{cdl} A_p(\vec{v}, \vec{s}'(\vec{v}))$. Because $s \models_{cdl} \forall \vec{x}.(A_p(\vec{v}, \vec{x}) \rightarrow \phi^\flat[\vec{x}/\vec{v}])$, so $s \models_{cdl} \phi^\flat[s'(\vec{v})/\vec{v}]$ holds. But this is to say $s' \models_{cdl} \phi^\flat$, because tr is a trace of p and according to Def. 3.9, the evaluation of s' on all variables of $\phi^\flat$ only differs from s on set $\vec{v}$.

On the other hand, suppose $s \models_{cdl} [p]\phi^\flat$, we need to prove for any vector $\vec{x}_0$, if $s \models_{cdl} A_p(\vec{v}, \vec{x}_0)$, then $s \models_{cdl} \phi^\flat[\vec{x}_0/\vec{v}]$. From Lemma A.3 (i) we know there exists a trace $tr \in val(p)$ such that $tr_b = s$ and $tr_e(\vec{v}) = \vec{x}_0$. Let $s' = tr_e$. Since $s \models_{cdl} [p]\phi^\flat$, by Def. 3.18 there is $s' \models_{cdl} \phi$. But based on the semantics of p, s' differs from s only on the set $\vec{v}$. So we have $s \models_{cdl} \phi[\vec{x}_0/\vec{v}]$. □

Based on Lemmas A.3 and A.5, we now prove Theorem 6.2 (i). According to Lemma A.2, to prove the expressibility of a CDL formula that contains SEPs, we only need to transform SEPs into their corresponding Finite-SEPs in this formula through the function *finite*, then prove the expressibility of this transformed formula.

Theorem 6.2 (i). We proceed by induction on the structure of the formula ϕ. In sequential SEPs, the only non-trivial cases are $\phi = [p]\phi_1^\flat$ and $\phi = [p]\xi$. We consider both cases at the same time. For any parallel SEP of the form $\|(p_1, ..., p_n)$, we can reduce it into an equivalent sequential SEP by the rewrite rules in Table 6.

1. If $p = \mu$, then according to the soundness of rules in Table 3, there is $\models_{cdl} tt \leftrightarrow [\mu]\xi$. According to the soundness of rule $(\pi\mu[])$ in Table 4, we have $\models_{cdl} \phi_1^\flat \leftrightarrow [\mu]\phi_1^\flat$.

2. If $p = \alpha$, according to the definition of $[p]\xi$ in Def. 3.18 we know $\models_{cdl} [\alpha]\hbar(\xi) \leftrightarrow [\alpha]\xi$. But $\hbar(\xi)$ is an AFOL formula, so from Lemma A.5 we know that $[\alpha]\hbar(\xi)$ is expressible.

   Because of Lemma A.5, the case for $[\alpha]\phi_1^\flat$ is obvious.

3. If $p = P?\alpha$, by the soundness of rule $(P?[])$ in Table 3 there is $\models_{cdl} P \rightarrow [\alpha]\xi \leftrightarrow [P?\alpha]\xi$. By inductive hypothesis $[\alpha]\xi$ is expressible, so $P \rightarrow [\alpha]\xi$ is also expressible.

   Similar for the case of $\phi = [P?\alpha]\phi_1^\flat$.

4. If $p = q \cup r$, by the soundness of rule $(\pi[\cup])$ in Table 3 there is $\models_{cdl} [q]\xi \wedge [r]Rel \leftrightarrow [q \cup r]\xi$. By inductive hypothesis, $[q]\xi$ and $[r]\xi$ are expressible, so $[q]\xi \wedge [r]\xi$ is also expressible.

   Similar for the situation when $\phi = [q \cup r]\phi_1^\flat$.

5. If $p = q; r$, then by the soundness of rule $(\pi[;])$ in Table 3 we can get $\models_{cdl} [q]\xi \wedge [q][r]\xi \leftrightarrow [q; r]\xi$. According to inductive hypothesis, $[q]\xi$ and $[r]\xi$ are all expressible. Let $\models_{cdl} \varphi_1^\flat \leftrightarrow [r]\xi$, by inductive hypothesis we see that $[q]\varphi_1^\flat$ is also expressible. Thus $[q]\xi \wedge [q][r]\xi$ is expressible.

   The case for $\phi = [q; r]\phi_1^\flat$ is similar.

6. If $p = q^\bullet$, we first consider the case when $\phi = [q^\bullet]\phi_1^\flat$. According to Prop. A.2, we only need to consider if $[\mathit{finite}(q)^*]\phi_1^\flat$ is expressible. However from Lemma A.5 we know $[\mathit{finite}(q)^*]\phi_1^\flat$ is expressible.

   As for $[q^\bullet]\xi$, by the soundness of rule $(\pi[\bullet]i)$ in Table 3, $\models_{cdl} [q^\bullet][q]\xi \leftrightarrow [q^\bullet]\xi$ holds. By inductive hypothesis, easy to see $[q]\xi$ is expressible. Let $\models_{cdl} \varphi_1 \leftrightarrow [q]\xi$. $[q^\bullet]\varphi_1$ belongs to case considered above.

□

Based on Lemma A.3 and Theorem 6.2 (i), now we prove Theorem 6.2 (ii). The main idea is based on [20, 25]: by making use of the expressibility of formula CDL and the soundness of proof system $\vdash^+_{cdl}$, we make an induction on the structure of p.

Theorem 6.2 (ii). We proceed by induction on the structure of p. In $\varphi^\flat \to op\,\phi^\flat$, when op is $\forall x$ or $\exists x$, the proof is obvious. This is because $\varphi^\flat \to op\,\phi^\flat$ itself is an AFOL formula, so there must be $\vdash^+_{cdl} \varphi^\flat \to op\,\phi^\flat$.

We first consider the case when op is $[p]$.

1. If $p = \mu$, by the soundness of rule $(\mu[])$ (see Table 3) and the FOL rules (see Table 5) there is $\models_{cdl} (\varphi^\flat \to \phi^\flat) \leftrightarrow \varphi^\flat \to [\mu]\phi^\flat$. Since $\varphi^\flat \to \phi^\flat$ is an AFOL formula, obviously $\vdash^+_{cdl} \varphi^\flat \to \phi^\flat$. By rule $(\mu[])$ and the FOL rules we immediately obtain: $\vdash^+_{cdl} \varphi^\flat \to [\mu]\phi^\flat$.

2. If $p = \alpha$, without the lose of generality let $\alpha = (\varsigma^c!e|x := e')$. Because $\models_{cdl} \varphi^\flat \to [\alpha]\phi^\flat$, so by the soundness of rule $(\phi[])$ and the FOL rules there is $\models_{cdl} \varphi^\flat[V'/V] \wedge A \to \phi^\flat$, where formula

$$A = (c^n = (c^n)' + 1 \wedge c^s = 1 \wedge x = e'[V'/V] \wedge \bigwedge_{1 \le i \le n} (d_i^n = (d_i^n)' \wedge d_i^s = 0)).$$

   Since $\varphi^\flat[V'/V] \wedge A \to \phi^\flat$ is an AFOL formula, $\vdash^+_{cdl} \varphi^\flat[V'/V] \wedge A \to \phi^\flat$ holds. By rule $(\phi[])$ and the FOL rules we obtain $\vdash^+_{cdl} \varphi^\flat \to [\alpha]\phi^\flat$.

3. If $p = P?\alpha$, according to the soundness of rule $(P?[])$ and the FOL rules there is $\models_{cdl} \varphi^\flat \to (P \to [\alpha]\phi^\flat) \leftrightarrow \varphi^\flat \to [\alpha]\phi^\flat$. So $\models_{cdl} \varphi^\flat \to \neg P \vee \varphi^\flat \to [\alpha]\phi^\flat$. Since $\varphi^\flat \to \neg P$ is an AFOL formula, we have

$$\vdash^+_{cdl} \varphi^\flat \to \neg P. \tag{A.15}$$

   By inductive hypothesis we have

$$\vdash^+_{cdl} \varphi^\flat \to [\alpha]\phi^\flat. \tag{A.16}$$

   From (A.15) and (A.16), applying the FOL rules (see Table 5) we can get $\vdash^+_{cdl} \varphi^\flat \to (P \to [\alpha]\phi^\flat)$. By rule $(P?[])$ and the FOL rules there is $\vdash^+_{cdl} \varphi^\flat \to [P?\alpha]\phi^\flat$.

4. If $p = q \cup r$, according to the soundness of rule $([\cup])$ and the FOL rules we know $\models_{cdl} \varphi^\flat \to ([q]\phi^\flat \wedge [r]\phi^\flat) \leftrightarrow \varphi^\flat \to [q \cup r]\phi^\flat$, i.e., $\models_{cdl} \varphi^\flat \to [q]\phi^\flat$ and $\models_{cdl} \varphi^\flat \to [r]\phi^\flat$. By inductive hypothesis, we have $\vdash^+_{cdl} \varphi^\flat \to [q]\phi^\flat$ and $\vdash^+_{cdl} \varphi^\flat \to [r]\phi^\flat$ hold. By rule $([\cup])$ and the FOL rules we conclude that $\vdash^+_{cdl} \varphi^\flat \to [q \cup r]\phi^\flat$.

5. If $p = q; r$, according to the soundness of rule $([;])$ and the FOL rules we can get

$$\models_{cdl} \varphi^\flat \to ([q][r]\phi^\flat) \leftrightarrow \varphi^\flat \to [q; r]\phi^\flat. \tag{A.17}$$

By Theorem 6.2 (i), there is an AFOL formula φ_1 such that $\models_{cdl} \varphi_1 \leftrightarrow [r]\phi^\flat$. So from (A.17) we have $\models_{cdl} \varphi^\flat \rightarrow [q]\varphi_1$. By inductive hypothesis we then have

$$\vdash^+_{cdl} \varphi_1 \rightarrow [r]\phi^\flat,\ \vdash^+_{cdl} \varphi^\flat \rightarrow [q]\varphi_1. \tag{A.18}$$

Applying rule ($[]gen$) to the left formula of (A.18), we obtain $\vdash^+_{cdl} [q]\varphi_1 \rightarrow [q][r]\phi^\flat$. Comparing it with the right formula of (A.18), we have $\vdash^+_{cdl} \varphi^\flat \rightarrow [q][r]\phi^\flat$. At last, by applying rule ($[;]$) and other FOL rules the result is obtained.

6. If $p = q^\bullet$, by Theorem 6.2 (i) there exists an AFOL formula $\phi^\flat_0$ such that

$$\models_{cdl} \phi^\flat_0 \leftrightarrow [q^\bullet]\phi^\flat. \tag{A.19}$$

Because $\models_{cdl} \varphi^\flat \rightarrow [q^\bullet]\phi^\flat$, we also have

$$\models_{cdl} \varphi^\flat \rightarrow \phi^\flat_0. \tag{A.20}$$

From (A.19), by rule ($[\bullet]u$) and rule ($[;]$), it is easy to see

$$\models_{cdl} \phi^\flat_0 \leftrightarrow [q^\bullet]\phi^\flat \leftrightarrow \phi^\flat \wedge [q;q^\bullet]\phi^\flat \leftrightarrow \phi^\flat \wedge [q][q^\bullet]\phi^\flat \leftrightarrow \phi^\flat \wedge [q]\phi^\flat_0.$$

From the serial logical equivalences above we can see that

$$\models_{cdl} \phi^\flat_0 \rightarrow [q]\phi^\flat_0 \text{ and } \models_{cdl} \phi^\flat_0 \rightarrow \phi^\flat. \tag{A.21}$$

By inductive hypothesis, (A.20) and (A.21) there are

$$\vdash^+_{cdl} \varphi^\flat \rightarrow \phi^\flat_0,\ \vdash^+_{cdl} \phi^\flat_0 \rightarrow [q]\phi^\flat_0 \text{ and } \vdash^+_{cdl} \phi^\flat_0 \rightarrow \phi^\flat. \tag{A.22}$$

Based on (A.22), by applying rule ($[\bullet]i$) we get $\vdash^+_{cdl} \varphi^\flat \rightarrow [q^\bullet]\phi^\flat$.

7. If $p = \|(q_1, ..., q_n)$, by applying the rewrite rules in Table 6, we can reduce p into a sequential program p', i.e., $p \rightsquigarrow p'$. By the soundness of rule (r), there is $\models_{cdl} \varphi^\flat \rightarrow [p']\phi^\flat \leftrightarrow \varphi^\flat \rightarrow [p]\phi^\flat$. Since p' is sequential, we can analyze it based on the cases given above. Using inductive hypothesis, finally we can get $\vdash^+_{cdl} \varphi^\flat \rightarrow [p']\phi^\flat$. By rule ($r$) and other FOL rules there is $\vdash^+_{cdl} \varphi^\flat \rightarrow [p]\phi^\flat$.

For the case when op is $\langle p\rangle$, the proof of the cases $p = \mu$, $p = \alpha$, $p = P?\alpha$, $p = q \cup r$, $p = q;r$ are similar. The difference is that in the proof we need to use rules ($\phi\langle\rangle$), ($P?\langle\rangle$), ($\mu\langle\rangle$), ($\langle;\rangle$), ($\langle\cup\rangle$), ($\langle\bullet\rangle u$), ($\langle\rangle gen$) and ($\langle\bullet\rangle i$) (some of them are shown in Table A.11), which are the *dual rules* of rules ($\phi[\alpha]$), ($P?$), (μ), ($[;]$), ($[\cup]$), ($[\bullet]u$), ($[]gen$) and ($[\bullet]i$) respectively.

We now prove the case $p = q^\bullet$. If $\models_{cdl} \varphi^\flat \rightarrow \langle q^\bullet\rangle\phi^\flat$, since $\models_{cdl} \neg[q^\bullet]\neg\phi^\flat \leftrightarrow \langle q^\bullet\rangle\phi^\flat$, by Prop. A.2 there is $\models_{cdl} \neg[r^*]\neg\phi^\flat \leftrightarrow \neg[q^\bullet]\neg\phi^\flat \leftrightarrow \langle q^\bullet\rangle\phi^\flat$, where $\mathit{finite}(q^\bullet) = r^*$, $r = \mathit{finite}(q)$. From Lemma A.5, the formula $\langle q^\bullet\rangle\phi^\flat$ can be expressed by an equivalent AFOL formula $\phi^\flat_0 = \neg\forall\vec{x}(A_{r^*}(\vec{v},\vec{x}) \rightarrow \neg\phi^\flat[\vec{x}/\vec{v}])$. Recall in Lemma A.3, let $A_{r^*}(\vec{v},\vec{v}') = \exists n \geq 1$, then $\mathit{Iter}(n) \vee \mathit{Iter}(0) = \exists n.((n \geq 1 \wedge \mathit{Iter}(n)) \vee \mathit{Iter}(0))$. Here we remove the part '$\exists n$', and let $A_{r^*}(\vec{v},\vec{v}',n) = (n \geq 1 \wedge \mathit{Iter}(n)) \vee \mathit{Iter}(0)$. In the formula $\phi^\flat_0$, we use $A_{r^*}(\vec{v},\vec{v}',n)$ to replace A_{r^*}, and we obtain an AFOL formula with n as a free variable in it: $\phi^\flat_0(n) = \neg\forall\vec{x}(A_{r^*}(\vec{v},\vec{x},n) \rightarrow \neg\phi^\flat[\vec{x}/\!/\vec{v}])$. Because $\models_{cdl} \varphi^\flat \rightarrow \langle q^\bullet\rangle\phi^\flat$, so there is

$$\models_{cdl} \varphi^\flat \rightarrow \exists n.\phi^\flat_0(n). \tag{A.23}$$

This is because $\exists n.\phi^\flat_0(n)$ and $\phi^\flat_0$ are in fact logical equivalent: intuitively, we just 'move' the quantifier $\exists n$ from $A_{r^*}(\vec{v},\vec{v}')$ to the front of the formula $\phi^\flat_0$. Since n only appears in $A_{r^*}(\vec{v},\vec{v}')$, the meaning of the formula does not change before and after the movement. According to (A.13) in Lemma A.3, Lemma A.5 and the soundness of the FOL rules (see Table 5), it is not hard to prove that when $n \geq 0$, there is

$$\models_{cdl} \phi^\flat_0(n) \leftrightarrow \langle r^n\rangle\phi^\flat. \tag{A.24}$$

By rule $(\langle;\rangle)$ of Table A.11 we have for any $n > 0$, $\models_{cdl} \phi_0^\flat(n) \leftrightarrow \langle r^n\rangle\phi^\flat \leftrightarrow \langle r\rangle\langle r^{n-1}\rangle\phi^\flat \leftrightarrow \langle r\rangle\phi_0^\flat(n-1)$. Because $\models_{cdl} \langle r^*\rangle\phi^\flat \leftrightarrow \langle q^\bullet\rangle\phi^\flat$, for any $n > 0$ there is

$$\models_{cdl} \phi_0^\flat(n) \to \langle q\rangle\phi^\flat(n-1). \tag{A.25}$$

In (A.24) let $n = 0$, by rule $(\mu\langle\rangle)$ of Table A.11 we get

$$\models_{cdl} \phi_0^\flat(0) \to \phi^\flat. \tag{A.26}$$

By inductive hypothesis, (A.23), (A.25) and (A.26), we obtain $\vdash^+_{cdl} \varphi^\flat \to \exists n.\phi_0^\flat(n)$, $\vdash^+_{cdl} \phi_0^\flat(n) \to \langle q\rangle\phi^\flat(n-1)$ and $\vdash^+_{cdl} \phi_0^\flat(0) \to \phi^\flat$. Applying rule $(\langle\bullet\rangle i)$ and the FOL rules we get $\vdash^+_{cdl} \varphi^\flat \to [q^\bullet]\phi^\flat$. □

$\dfrac{P \to \langle\alpha\rangle A}{\langle P?\alpha\rangle A}\ (P?\langle\rangle)$		$\dfrac{tt}{\langle\mu\rangle\sim\xi}\ (\pi\mu\langle\rangle)$	$\dfrac{\phi}{\langle\mu\rangle\phi}\ (\mu\langle\rangle)$
where $A \in \{\sim\xi, \phi\}$			
$\dfrac{\langle p\rangle\sim\xi \vee \langle p\rangle\langle q\rangle\sim\xi}{\langle p;q\rangle\sim\xi}\ (\pi\langle;\rangle)$	$\dfrac{\langle p\rangle\sim\xi \vee \langle q\rangle\sim\xi}{\langle p\cup q\rangle\sim\xi}\ (\pi\langle\cup\rangle)$	$\dfrac{\langle p;p^\bullet\rangle\sim\xi}{\langle p^\bullet\rangle\sim\xi}\ (\pi\langle\bullet\rangle u)$	$\dfrac{\langle p^\bullet\rangle\langle p\rangle\sim\xi}{\langle p^\bullet\rangle\sim\xi}\ (\pi\langle\bullet\rangle i)$
$\dfrac{\langle p\rangle\langle q\rangle\phi}{\langle p;q\rangle\phi}\ (\langle;\rangle)$	$\dfrac{\langle p\rangle\phi \vee \langle q\rangle\phi}{\langle p\cup q\rangle\phi}\ (\langle\cup\rangle)$	$\dfrac{\phi \vee \langle p;p^\bullet\rangle\phi}{\langle p^\bullet\rangle\phi}\ (\langle\bullet\rangle u)$	

Table A.11: Other dual rules in CDL proof system

As for the condition (iii) of Theorem 6.2, when $op \in \{[p], \langle p\rangle\}$, the cases are in fact stated as rules $([]gen)$ and $(\langle\rangle gen)$. We give the proof of Theorem 6.2 for the case when $op \in \{\forall x, \exists x\}$ as follows.

Theorem 6.2 (iii). We only consider the case '$\forall x$'. The case '$\exists x$' can be similarly obtained by using the dual rules of the rules used in the proof below.

Actually, using the FOL rules in Table 5, we can construct the following deduction:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, \forall x\varphi \Rightarrow \varphi[x'/x] \to \phi[x'/x], \Delta}{\Gamma, \forall x\varphi, \varphi[x'/x] \Rightarrow \phi[x'/x], \Delta}\ (\to l)}{\Gamma, \forall x\varphi \Rightarrow \phi[x'/x], \Delta}\ (\forall l)}{\Gamma, \forall x\varphi \Rightarrow \forall x\phi, \Delta}\ (\forall r)}{\Gamma \Rightarrow \forall x\varphi \to \forall x\phi, \Delta}\ (\to r)$$

where x' is a new variable w.r.t. Γ and Δ. □

The proof of Theorem 6.2 (iv) follows the proof of Theorem 6.2 (ii), where the analysis of the most cases are similar.

Theorem 6.2 (iv). We proceed by induction on the structure of p. Firstly, consider the completeness of the formula $\varphi^\flat \to [p]\xi$.

1. The cases for $p = \mu$, $p = \alpha$, $p = P?\alpha$, $p = q \cup r$ and $p = \|(q_1, ..., q_n)$ are similar to the corresponding cases in the proof of Theorem 6.2 (ii) given above. The proof can be given in an inductive way by applying rules $(\pi\mu[])$, $(\pi[])$, $(P?[])$ and $(\pi[\cup])$, the rewrite rules in Table 6, other FOL rules and the soundness of these rules. We omit it here.

2. If $p = q; r$, according to the soundness of rule $(\pi[;])$ and the FOL rules we obtain $\models_{cdl} \varphi^\flat \to [q]\xi \wedge [q][r]\xi \leftrightarrow \varphi^\flat \to [q; r]\xi$. From Theorem 6.2 (i) there exists an AFOL formula φ_1 such that

$$\models_{cdl} \varphi_1 \leftrightarrow [r]\xi. \tag{A.27}$$

Because $\models_{cdl} \varphi^\flat \to [q]\xi$ and $\models_{cdl} \varphi^\flat \to [q]\varphi_1$, by inductive hypothesis we know

$$\vdash^+_{cdl} \varphi^\flat \to [q]\xi \text{ and } \vdash^+_{cdl} \varphi^\flat \to [q]\varphi_1 \tag{A.28}$$

holds. From (A.27), by inductive hypothesis we obtain $\vdash^+_{cdl} \varphi_1 \to [r]\xi$. Applying rule $([]gen)$ to this formula we get $\vdash^+_{cdl} [q]\varphi_1 \to [q][r]\xi$. Then based on the right formula of (A.28) we have

$$\vdash^+_{cdl} \varphi^\flat \to [q][r]\xi. \tag{A.29}$$

With (A.29) and the left formula of (A.28) we can get $\varphi^\flat \to ([q]\xi \wedge [q][r]\xi)$. The result is obtained by applying rule $(\pi[;])$.

3. If $p = q^\bullet$, based on the soundness of rule $(\pi[\bullet]i)$, there is $\models_{cdl} \varphi^\flat \to [q^\bullet][q]\xi$. From Theorem 6.2 (i) there exists an AFOL formula $\phi^\flat_0$ such that

$$\models_{cdl} \phi^\flat_0 \leftrightarrow [q]\xi. \tag{A.30}$$

So $\models_{cdl} \varphi^\flat \to [q^\bullet]\phi^\flat_0$ also holds. But this case has already been proved in Theorem 6.2 (ii), we have

$$\vdash^+_{cdl} \varphi^\flat \to [q^\bullet]\phi^\flat_0. \tag{A.31}$$

According to (A.30), by inductive hypothesis we can get $\vdash^+_{cdl} \phi^\flat_0 \to [q]\xi$. Applying rule$([]gen)$, there is

$$\vdash^+_{cdl} [q^\bullet]\phi^\flat_0 \to [q^\bullet][q]\xi. \tag{A.32}$$

From (A.31) and (A.32) the result is straightforward.

For the completeness of the formula $\varphi^\flat \to \langle p\rangle \sim \xi$, its proof is similar to that of $\varphi^\flat \to \langle p\rangle \sim \xi$ given above. Accordingly, in the proof of different cases we need to use the dual rules of rules $(\pi\mu[])$, $(\pi[])$, $(P?[])$, $(\pi[\cup])$, $(\pi[;])$, $([]gen)$, $(\pi[\bullet]i)$, $(\pi\mu\langle\rangle)$, $(\pi\langle\rangle)$, $(P?\langle\rangle)$, $(\pi\langle\cup\rangle)$, $(\pi\langle;\rangle)$, $(\langle\rangle gen)$ and $(\pi\langle\bullet\rangle i)$. Some of them are listed in Table A.11. □

B. A Part of Code for the Definition of the Sequent in Coq

```
(* ========== CDL proof system ============*)
(* Gamma, Delta *)
Definition Gamma := list CDL_exp.
Definition Delta := list CDL_exp.

(* target place *)
Inductive place :=
  exp : CDL_exp -> place |
  empty : place.

(* ========== construction of the proof system ============ *)
Reserved Notation "<| T , p1 ==> p2 , D // V , C , ntC |> " (at level 75).

(* definition of sequent *)
Inductive Seq : Gamma -> place -> place -> Delta -> (list Var) -> (list Var) -> (list Var) -> Prop :=

  (* ********* rules special for the sequent ********** *)
  r_placeR_rmv_int : forall (T : Gamma) (pls1 : place) (D : Delta) (V : list Var) (C : list Var) (ntC : list Var)
    (phi : CDL_exp),
    (
      Seq T pls1 empty (addNail D phi) V C ntC (* put phi at the nail of D *)
    )
    ->
```

```
      (
        Seq T pls1 (exp phi) D V C ntC
      )
    |

    r_placeL_rmv_int : forall (T : Gamma) (pls1 : place) (D : Delta) (V : list Var) (C : list Var) (ntC : list Var)
      (phi : CDL_exp),
      (
        Seq (addNail T phi) empty pls1 D V C ntC (* put phi at the nail of T *)
      )
      ->
      (
        Seq T (exp phi) pls1 D V C ntC
      )
    |
    ...

  (* ************ rules for combinational events and formulae [p] rel ************ *)
  (* Rule (\pi []) *)
    r_Pi_all_int1 : forall (T : Gamma) (D : Delta) (V : list Var) (C : list Var) (ntC : list Var)
      (x : Var) (e : e_exp) (A : Evt) (r : rel),
      let n := NewId V in
      (
        Seq ((x =' (e_2_E e) [(var n) subs-E &x]) :: (T [ (var n) subs-I &x ]))
          empty (exp ([[ @ A ]]' r)) (D [ (var n) subs-I &x ]) (var n :: V) C ntC
      )
      ->
      (
        Seq T empty (exp ([[ @ ((x :=' e) :: A) ]]' r)) D V C ntC
      )
    |

    r_Pi_all_int2 : forall (T : Gamma) (D : Delta) (V : list Var) (C : list Var) (ntC : list Var)
      (c : Var) (e : e_exp) (A : Evt) (r : rel),
      let n := NewId V in
      let m := S n in
      (
        Seq (
            (n(c) =' (var n) +' 1) ::
              (s(c) =' 1) ::
                ((T [ (var n) subs-I &n(c) ]) [ (var m) subs-I &s(c) ])
          )
            empty
              (exp ([[ @ A ]]' r))
                ((D [ (var n) subs-I &n(c) ]) [ (var m) subs-I &s(c) ])
                  (var m :: var n :: V)
                    C
                      (rmv ntC c) (* remove c from the set ntC *)
      )
      ->
      (
        Seq T empty (exp ([[ @ ((c ! e) :: A) ]]' r)) D V C ntC
      )
    |

    r_Pi_all_int_idle1 : forall (T : Gamma) (D : Delta) (V : list Var) (C : list Var)
      (r : rel),
      (
        Seq T empty (exp (hbar r)) D V C C (* when ntC = nil, reset it to C *)
      )
      ->
      (
        Seq T empty (exp ([[ @ idle ]]' r)) D V C nil
      )
    |

    r_Pi_all_int_idle2 : forall (T : Gamma) (D : Delta) (V : list Var) (C : list Var) (c : Var) (ntC' : list Var)
      (r : rel),
      (
        let n := NewId V in
        (
          Seq ((s(c) =' 0) :: (T [(var n) subs-I &s(c)]))
              empty
                (exp ([[ @ idle ]]' r))
                  (D [(var n) subs-I &s(c)])
                    ((var n) :: V)
                      C
```

```
                        ntC'
      )
    )
    ->
    (
      Seq T empty (exp ([[ @ idle ]]' r)) D V C (c :: ntC')
    )
  |

  (* Rule P? *)
    r_Test_all_rel_int : forall (T : Gamma) (D : Delta) (V : list Var) (C : list Var) (ntC : list Var)
    (P : P_exp) (A : Evt) (r : rel),
    (
      Seq T empty (exp ((Pexp_2_CDLexp P) ->' [[ @ A ]]' r )) D V C ntC
    )
    ->
    (
      Seq T empty (exp ([[ P ? A ]]' r)) D V C ntC
    )
  |

  (* Rule \pi[\cup] *)
   r_PiCho_all_int : forall (T: Gamma) (D : Delta) (V : list Var) (C : list Var) (ntC : list Var)
    (p : SEP_exp) (q : SEP_exp) (r : rel),
    (
      Seq T empty (exp ([[ p ]]' r /\' [[ q ]]' r)) D V C ntC
    )
    ->
    (
      Seq T empty (exp ([[ p U q ]]' r)) D V C ntC
    )
  |
   ...

  (* ********** rules of FOL *********** *)
  (* Rule (o) *)
   r_o_int : forall (T : Gamma) (D : Delta) (V : list Var) (C : list Var) (ntC : list Var),
    (
      let fof := (Seq_2_CDLexp T empty empty D) in
      (
        forall (st : Var -> nat) (fn : Var -> nat) (gn : Var -> nat) , (CDLexp_2_Prop fof st fn gn)
       (* the Prop formula corresponding to the sequent: 'Seq T empty empty D V' *)
      )
    )
    ->
    (
      Seq T empty empty D V C ntC
    )
  |

  (* Rule (/\ r) *)
  r_andR_int : forall (T : Gamma) (D : Delta) (V : list Var) (C : list Var) (ntC : list Var)
    (phi1 : CDL_exp) (phi2 : CDL_exp),
    (
      (
        Seq T empty (exp phi1) D V C ntC
      )
      /\
      (
        Seq T empty (exp phi2) D V C ntC
      )
    )
    ->
    (
      Seq T empty (exp (phi1 /\' phi2)) D V C ntC
    )
  |

  (* Rule (-> r) *)
  r_impR_int : forall (T : Gamma) (D : Delta) (V : list Var) (C : list Var) (ntC : list Var)
    (phi1 : CDL_exp) (phi2 : CDL_exp),
    (
      Seq T (exp phi1) (exp phi2) D V C ntC
    )
    ->
    (
      Seq T empty (exp (phi1 ->' phi2)) D V C ntC
```

```
      )
    |
    ...

  (* define the notation at the end *)
where "<| T , p1 ==> p2 , D // V , C , ntC |>" := (Seq T p1 p2 D V C ntC).
```