Front.Comput.Sci.
DOI

REVIEW ARTICLE

A Dynamic Logic for Verification of Synchronous Models based on Theorem Proving

Yuanrui ZHANG [1,2], **Frédéric MALLET** [3], **Zhiming LIU** (✉)[2,4]

1 School of Mathematics and Statistics, Southwest University, China
2 RISE, College of Computer & Information Science, Southwest University, China
3 University Cote d'Azur, CNRS, Inria, I3S, France
4 Center for Intelligent and Embedded Software, Northwest Polytechnical University, China

Abstract The synchronous paradigm has been very successful for the design of safety-critical reactive systems. There are many languages following the synchrony hypothesis to address rigorously systems with an inherently concurrent but fully determined behaviour. They come with a wide variety of verification tools including model-checkers SAT/SMT Solvers, term-rewriting techniques, type inference engines. In this paper, we propose a verification framework for synchronous models based on theorem proving. Specifically, we initially propose a novel dynamic logic, called synchronous dynamic logic (SDL). SDL extends the regular program model of first-order dynamic logic (FODL) with necessary primitives to capture the notion of synchrony and synchronous communication between parallel synchronous programs, and enriches FODL formulas with temporal dynamic logical formulas to specify safety properties — a type of properties mainly concerned in reactive systems. To capture precisely the synchronous communications, we define a constructive semantics for the program model of SDL. We build a sound proof system for SDL, which is also relatively complete under a certain restriction made to the SDL programs. Compared to previous verification approaches, SDL provides a divide and conquer way to analyze and verify synchronous models based on compositional reasoning of the syntactic structure of the programs of SDL. To show the potential of SDL to be used in practice, we apply SDL to specify and verify a toy example in the synchronous model SyncChart.

Received month dd, yyyy; accepted month dd, yyyy

E-mail: *zhangyrmath*@126.*com*, *zhimingliu*88@*swu.edu.cn*

Keywords dynamic logic, synchronous models, reactive systems, verification framework, theorem proving, safety properties

1 Introduction

Synchronous models are well-adapted for modelling and specifying reactive systems [1] — a type of computer systems which maintain an on-going interaction with their environment at a speed determined by this environment. The notion of synchronous model was firstly proposed in 1980's along with the invention and development of synchronous programming languages [2]. Among these synchronous programming languages, the most famous and representative ones are Signal [3], Lustre [4] and Esterel [5], which have been widely used in academic and industrial communities (for instance, cf. [6–9]).

One crucial modelling paradigm taken in synchronous models is *synchrony hypothesis* [2], which models reactive systems as sequences of *reactions*. Each reaction captures inputs, computes the reaction and produces some outputs simultaneously. This logical time of reaction is reduced to a discrete sequence of instants, at which, instantaneous events may or may not occur. There may still be some causal relationships between events at the same instant, we call them *logical orders*. In this time model, physical time is then treated as a particular case, where instants are tagged with a physical time tags forcing a worst-case time analysis to ensure the correct behaviour. While the abstraction leaves the non-

determinism outside the system, synchronous model forces a fully determined behavior by computing which set of events must tick at each instant. This proves to be very handy for testing [10, 11], decrease the size of models compared to an exhaustive exploration of all the interleavings.

Synchronous systems proved to be particularly convenient for systems where all the resources are known at compile time (like circuit design). It therefore reduces to finite state spaces and model-checking becomes a tool of choice [4, 12–16] to conduct exhaustive analysis. Model checking is decidable but in numerous cases still suffers from the well-known state-space explosion problem. Theorem-proving-based program verification [17], on the other hand, is based on the reasoning of syntactic structure of programs. As a complement technique to model checking it provides a "divide and conquer" way to support *modular verification*. Dynamic logic [18] is a mathematical logic that supports theorem-proving-based program verification. Unlike the three-triple forms $\{\phi\}p\{\psi\}$ of Hoare logic [19], it integrates a program model p directly into a logical formula so as to define a *dynamic logical formula* in a form like $\phi \to [p]\psi$ (meaning the same thing as the triple $\{\phi\}p\{\psi\}$). Such an extension on formulas allows to express the negations of program properties, e.g. $\neg[p]\psi$, which makes dynamic logic more expressive than Hoare logic. Dynamic logical formulas can well support compositional reasoning on programs. For example, by applying a compositional rule:

$$\frac{[p]\psi \wedge [q]\psi}{[p \cup q]\psi} \quad ,$$

the proof of the property $[p \cup q]\psi$ of a choice program $p \cup q$, can be discomposed into the proof of the property $[p]\psi$ of the subprogram p and the proof of the property $[q]\psi$ of the subprogram q.

Dynamic logic has proven to be a useful and successful verification framework for reasoning about programs and has been applied or extended to be applied in different types of programs and systems (e.g. [20–23]). However, traditional dynamic logics can only reason about sequential programs, and can only specify state properties, i.e., properties held after the terminations of a program. In reactive systems, however, since a system is usually non-terminating, people care more about temporal properties (especially safety properties [4]), i.e., properties concerning each reaction of a system.

In this paper, we propose a novel theory of dynamic logic, called *synchronous dynamic logic* (SDL), for reasoning about synchronous models of reactive systems. SDL extends *regular programs* [24] of first-order dynamic logic (FODL) [25] with the notion of reactions and parallel operator to model the execution mechanism and communication in synchronous models, and extends FODL formulas with *temporal dynamic logical formulas* from [23] to express safety properties in reactive systems. Based on the dynamic logic with modalities proposed and developed in [26], we build a sound proof system for SDL, which is also proved to be relatively complete [27] under a certain condition.

Unlike process algebras such as SCCS [28], which considers all events executed at an instant as disordered elements, one innovation of SDL is that in SDL we consider the logical before-and-after order between the events which are executed at one instant. This is one of the key characteristics of synchronous models. Models with similar characteristic were also proposed recently, like [11]. When defining the program model of SDL, the main challenge is to give a correct semantics for synchronous execution mechanism. We adopt an approach similar to that in the definition of constructive semantics of Esterel in [29], and prove that our proposed semantics is constructive in the same sense as in Esterel.

The proof system of SDL extends that of FODL with a set of rules for the new primitives introduced in SDL, among which the rules for temporal dynamic logical formulas are directly inherited from [23, 26]. However, as will be discussed in detail in Sect. 5.2, those rules are not valid for all programs of SDL so that the whole proof system is no longer a relatively complete one. The deep reason is that the test statements and parallel operator introduced in SDL can block parts of the program behaviours, which is one of the essential differences of our proposed formalism compared to the previous one [26]. We show that when confining our consideration to a certain type of programs, the proof system is relatively complete. Besides the rules for sequential programs, we also propose rewrite rules for the parallel operator of SDL, which can deduce parallel programs by rewriting them into sequential ones. For those parallel programs which contain star programs and cannot be directly transformed by rewriting procedure, we propose an algorithm, which turns out to be the standard Brzozowski's procedure of transforming a finite automaton into a regular expression.

To summarize, this paper mainly focus on the theory of SDL, the main contributions are as follows:

1. We propose a synchronous dynamic logic — SDL — and define its syntax and semantics (Sect. 3); We give a constructive semantics for the synchronous programs in SDL and make an analysis of its correctness (Sect. 3.2.2).

2. We build a proof system for SDL (Sect. 4), in which we propose a set of rewrite rules for parallel programs of SDL, and propose a Brzozowski's transformation procedure for complex parallel programs that cannot be directly transformed by simple rewrites (Sect. 4.3).
3. We analyze the completeness of the proof system of SDL (Sect. 5.2); We prove that it is sound and relatively complete under a certain type of programs of SDL (Appendix 9.1, 9.2).

To show the potential of SDL, we give a toy example (Sect. 6) of syncCharts [30] to explain how SDL models synchronous models, specifies and verifies their safety properties. We also display how to prove a simple SDL formula in the proof system of SDL.

Rather than synchronous programming languages, such as Esterel [5] and Quartz [31], which have been used in implementing real reactive systems, the program model of SDL is an abstract formalism which aims at describing reactive systems at a high level. It only contains necessary primitives for expressing basic behaviours of programs and capturing the synchronous execution mechanism of reactive systems, and ignore other data structures for actual implementation of real systems. SDL itself should NOT be taken as a practical verification methodology neither, but a theory which provides a framework showing how a general synchronous model can be reasoned about. SDL, together with its proof system, forms the theoretical foundation of, perhaps a more practical verification technique (with tools), which is equipped with a more concrete synchronous language and a richer proof system than those of SDL.

The rest of this paper is organized as follows. In Sect. 2, we give a brief introduction to FODL. We propose SDL and build a proof system for SDL in Sect. 3 and 4 respectively, and in Sect. 5, we analyze the soundness and completeness of SDL calculus. In Sect. 6, we show how SDL can be used in specifying and verifying synchronous models through an example. Sect. 7 introduces related work, while a conclusion is made in Sect. 8.

2 First-order Dynamic Logic

The syntax of FODL is as follows:

$$p ::= \psi? \mid x := e \mid p\,;p \mid p \cup p \mid p^*,$$
$$\phi ::= tt \mid a \mid [p]\phi \mid \neg\phi \mid \phi \wedge \phi \mid \forall x.\phi.$$

It consists of two parts, a program model called *regular programs* p and a set of logical formulas ϕ.

$\psi?$ is a *test*, meaning that at current state the proposition ψ is true. $x := e$ is an *assignment*, meaning that assigning the value of the expression e to the variable x. The syntax of ψ and e depends on the discussed domain. For example, e could be an arithmetic expression and ψ could be a quantifier-free first-order logical formula defined in Def. 3.1, 3.2. ; is the *sequence operator*, $p\,;q$ means that the program p is first executed, after it terminates, the program q is executed. $\cup$ is the *choice operator*, $p \cup q$ means that either the program p or q is executed. $*$ is the *star operator*, p^* means that p is nondeterministically executed for a finite number of times.

tt is Boolean true. (We also use ff to represent Boolean false.) a is an atomic formula whose definition depends on the discussed domain. For example, it could be the term θ defined in Def. 3.2. Formulas $[p]\phi$ are called *dynamic formulas*. $[p]\phi$ means that after all executions of p, formula ϕ holds.

The semantics of FODL is based on Kripke frames [24]. A Kripke frame is a pair (S, val) where S is a set of states, also called *worlds*, and val is an interpretation, also called a *valuation*, 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, the program p is executed and terminates at the state s'. For each state s, $s \in val(\phi)$ means that s satisfies the formula ϕ; $s \in val([p]\phi)$ means that for all pairs $(s, s') \in val(p)$, $s' \in val(\phi)$. For a formal definition of the semantics of FODL, one can refer to [24].

The proof system of FODL is sound and relatively complete. Except for the the assignment $x := e$ and test $\psi?$, the rules for other operators and logical connectives are also defined as a part of the proof system of SDL below in Table 1 and 3. Refer to [24] for more details.

3 Synchronous Dynamic Logic

In this section, we propose a novel dynamic logic, called synchronous dynamic logic (SDL), for specifying and reasoning about synchronous models of reactive systems. SDL extends FODL [25] in two aspects: (1) the program model of SDL, called *synchronous programs* (SPs), extends regular programs with the notions of reactions and signals in order to model the execution mechanism of synchronous models, and with the parallel operator to model communications in synchronous models; (2) SDL formulas extends FODL formulas with a type of formulas from [23], called *temporal dynamic logical formulas*, in order to capture safety properties in

reactive systems.

In Sect. 3.1, we firstly define the syntax of SDL, then in Sect. 3.2, we give its semantics.

3.1 Syntax of SDL

SDL consists of a program model defined in Def. 3.3 and a set of logical formulas defined in Def. 3.5.

Before defining the program model of SDL, we first introduce the concepts of terms and first-order logical formulas.

Definition 3.1 (Terms)**.** *The syntax of a term e is an arithmetical expression given as the following BNF form:*

$$e ::= x \mid c \mid f(e, e),$$

where $x \in Var$ is a variable, $c \in \mathbb{Z}$ is a constant, $f \in \{+, -, \cdot, /\}$ is a function.

The symbols $+, -, \cdot, /$ represent the usual binary functions in arithmetic theory: addition, substraction, multiplication and division respectively.

Definition 3.2 (First-order Logical Formulas)**.** *The syntax of an arithmetical first-order logical formula ψ is defined as follows:*

$$\psi ::= tt \mid \theta(e, e) \mid \neg\psi \mid \psi \wedge \psi \mid \forall x.\psi,$$

where $\theta \in \{<, \leq, =, >, \geq\}$ is a relation.

The symbols $<, \leq, =, >, \geq$ represent the usual binary relations in arithmetic theory, for example, $<$ is the "less-than" relation, $=$ is equality, and so on. As usual, the logical formulas with other connectives, such as $\vee, \exists, \rightarrow$, can be expressed by the formulas given above.

The formulas defined in Def. 3.2 contains the terms and relations interpreted as in Peano arithmetic theory, so in this paper, we also call them *arithmetic first-order logical (AFOL) formulas*.

The program model of SDL, called *synchronous program* (SP), extends the regular program of FODL with primitives to support the modelling paradigm of synchronous models. In order to capture the notion of reactions, we introduce the *macro event* in an SP to collect all events executed at the same instant. In order to describe the communications in synchronous models, we introduce the *signals* and *signal conditions* as in synchronous programming languages like Esterel [5] to send and receive messages between SPs, and we introduce a parallel operator to express that several SPs are executed concurrently.

Definition 3.3 (Synchronous Programs)**.** *The syntax of a synchronous program p is given as the following BNF form:*

$$p ::= \mathbf{1} \mid \mathbf{0} \mid \alpha \mid p\,;p \mid p \cup p \mid p^* \mid \cap(p, ..., p),$$

where α is defined as:

$$\alpha ::= \epsilon \mid evt\,.\,\alpha,$$
$$evt ::= \psi? \mid \varrho? \mid \varsigma!e \mid x := e,$$
$$\varrho ::= \hat{\varsigma}(x) \mid \bar{\varsigma}.$$

The set of all synchronous programs is denoted by **SP**. We call $\mathbf{1}, \mathbf{0}$ and α *atomic programs*, and call the programs of other forms *composite programs*. We denote the set of all atomic programs as $\mathbf{SP}^{at}$.

1 is a program called *nothing*. As implied by its name, the program neither does anything nor consumes time. The role it plays is similar to the statement "nothing" in Esterel [5].

0 is a program called *halting*. It causes a deadlock of the program. It never proceeds and the program halts forever. The program **0** is similar to the statement "halt" in Esterel.

The program α is called a *macro event*. It is the collection of all events executed at the current instant in an SP. A macro event consists of a sequence of events linked by dots . one by one and it always ends up with a special event ϵ called *skip*. The event ϵ skips the current instant and forces the program move to the next instant. ϵ is the only term that consumes time in SPs. It plays a similar role as the statement "skip" in Esterel. An event *evt*, sometimes also called a *micro event*, can be either a test $\psi?$, a *signal test* $\varrho?$, a *signal emission* $\varsigma!e$ or an assignment $x := e$. The tests $\psi?$ and assignments $x := e$ have the same meanings as in FODL. A signal test $\varrho?$ checks the signal condition ϱ at the current instant. ϱ has two forms. $\hat{\varsigma}(x)$ means that the signal ς is emitted at the current instant. x is a variable used for storing the value of ς. $\bar{\varsigma}$ means that the signal ς is absent at the current instant. A signal emission $\varsigma!e$ emits a signal ς with a value expressed as a term e at the current instant. We stipulate that a signal ς can emit with no values, and we call it a *pure signal*, denoted by ς. Sometimes we also simply write $\varsigma!e$ as ς when the value e can be neglected in the context.

The sequence programs $p\,;q$, choice programs $p \cup q$ and the star programs (or called *finite loop programs*) p^* have the same meanings as in FODL. $\cap$ is the parallel operator. $\cap(p_1, ..., p_n)$ means that the programs $p_1, ..., p_n$ are executed concurrently. When $n = 2$, we also write $\cap(p_1, p_2)$ as $p_1 \cap p_2$. We often call an SP without a parallel operator $\cap$ a *sequential program*, and call an SP that is not a sequential program a *parallel program*.

In a parallel program $\cap(p_1, ..., p_n)$, at each instant, events from different programs $p_1, ..., p_n$ are executed simultaneously based on the same environment, while events in each program p_i are executed simultaneously, but in a sequence order. SP model follows the style of SCCS [28] in its syntax, but differs in the treatments to the execution order of events at one instant. In SCCS, all events at one instant are considered as disordered elements that are commutative.

A closed SP is a program that does not interfere with its environment. In this paper, as the first step to propose a dynamic logic for synchronous models, we only focus on building a proof system for closed SPs. This makes it easier for us to define the semantics for SPs because there is no need to consider the semantics for signals. Similar approach was taken in [32] when defining a concurrent propositional dynamic logic.

Definition 3.4 (Closed SPs)**.** *Closed SPs are a subset of SPs, denoted as Cl*(**SP**)*, where a closed SP $q \in Cl(\mathbf{SP})$ is defined by the following grammar:*

$$q ::= \mathbf{1} \mid \mathbf{0} \mid \alpha' \mid q\,;q \mid q \cup q \mid q^* \mid \cap(p, ..., p),$$

where $p \in \mathbf{SP}$ is an SP, α' is defined as:

$$\alpha' ::= \epsilon \mid evt' \,.\, \alpha',$$
$$evt' ::= \psi? \mid x := e.$$

As indicated in Def. 3.4, a closed SP is an SP in which signals and signal tests can only appear in a parallel program.

We often call a macro event α' a *closed macro event*, call an event evt' a *closed event*. We call an SP which is not closed an *open SP*. We denote the set of all closed atomic programs as $Cl(\mathbf{SP}^{at})$. For convenience, in the rest of the paper, we also use p to represent a closed SP and use α, evt to represent a closed macro event and a closed event respectively.

SDL formulas extend FODL formulas with temporal dynamic logical formulas of the form $[p]\Box\phi$ from [23] to capture safety properties in reactive systems. The syntax of SDL formulas is given as follows.

Definition 3.5 (SDL Formulas)**.** *The syntax of an SDL formula ϕ is defined as follows:*

$$\phi ::= tt \mid \theta(e, e) \mid \neg\phi \mid \phi \wedge \phi \mid \forall x.\phi \mid [p]\phi \mid [p]\Box\phi,$$

where $\theta \in \{<, \leq, =, >, \geq\}$, $p \in Cl(\mathbf{SP})$.

ϕ is often called a *state formula*, since its semantics concerns a set of states. A property described by a state formula, hence, is called a *state property*.

The formula $[p]\Box\phi$ captures that the property ϕ holds at each reaction of the program p. Intuitively, it means that all execution traces of p satisfy the temporal formula $\Box\phi$, which means that all states of a trace satisfy ϕ. The dual formula of $[p]\Box\phi$ is written as $\langle p\rangle\Diamond\phi$, we have $\neg[p]\Box\phi = \langle p\rangle\Diamond\neg\phi$. Intuitively, $\langle p\rangle\Diamond\phi$ means that there exists a trace of p that satisfies the temporal formula $\Diamond\phi$, which means that there exists a state of a trace that satisfies ϕ.

We assign precedence to the operators in SDL: the unary operators $*$, $\neg$, $\forall x$ and $[p]$ bind tighter than binary ones. ; binds tighter than $\cup$. For example, the formula $\neg\psi \wedge [\alpha\,;p^* \cup q]\phi \wedge \psi'$ should be read as $(\neg\psi) \wedge ([(\alpha\,;(p^*)) \cup q]\phi) \wedge \psi'$.

Definition 3.6 (Bound Variables in SPs)**.** *In SDL, a variable x is called a bound variable if it is bound by a quantifier $\forall x$, an assignment $x := e$ or a signal test $\hat{\varsigma}(x)?$, otherwise it is called a free variable.*

We use $BV(p)$ and $FV(p)$ (resp. $BV(\phi)$ and $BV(\phi)$) to represent the sets of bound and free variables of an SP p (resp. a formula ϕ) respectively.

We introduce the notion of substitution in SDL. A substitution $\phi[e/x]$ replaces each free occurrence of the variable x in ϕ with the expression e. A substitution $\phi[e/x]$ is *admissible* if there is no variable y that occurs freely in e but is bound in $\phi[e/x]$. In this paper, we always guarantee admissible substitutions by bound variables renaming mentioned above.

In this paper, we always assume *bound variables renaming* (also known as $\alpha-$ conversion) for renaming bound variables when needed to guarantee admissible substitutions. For example, for a substitution $([z := x - 1\,.\,\epsilon]x > z)[x/z]$, it equals to $[z := y - 1\,.\,\epsilon]y > x$ by renaming x with a new variable y.

In SPs, we stipulate that signals are *the only way* for communication between programs. Therefore, any two programs running in parallel cannot communicate with each other through reading/writing a variable in *Var*. To meet this requirement, we put a restriction on the bound variables of SPs that are running in parallel.

Definition 3.7 (Restriction on Parallel SPs)**.** *In SPs, any parallel program $\cap(p_1, ..., p_n)$ satisfies that*

$$BV(p_i) \cap FV(p_j) = BV(p_j) \cap FV(p_i) = \emptyset$$

for any i, j, $1 \leq i < j \leq n$.

Like above, we assume bound variables renaming for the restriction in Def. 3.7. For example, given two programs $p = (x := 1\,;v := x + 2)$ and $q = (\epsilon\,;y := x + 1)$, we have $p \cap q = (z := 1\,;v := z + 2) \cap (\epsilon\,;y := x + 1)$ by renaming the bound variable x of p with a new variable z.

3.2 Semantics of SDL

Because of the introduction of temporal dynamic logical formulas $[p]\Box\phi$ in SDL, it is not enough to define the semantics of SPs as a pair of states like the semantics of the regular program of FODL [24], because except recording the states where a program starts and terminates, we also need to record all states during the execution of the program.

Before defining the semantics of SDL, we first introduce the notions of states and traces.

Definition 3.8 (States)**.** *A state $s : Var \to \mathbb{Z}$ is a mapping from the set of variables Var to the domain of integers $\mathbb{Z}$.*

We denote the set of all states as $\mathbf{S}$.

Definition 3.9 (Traces)**.** *A trace tr is a finite sequence of states:*

$$s_1 s_2 ... s_n, \text{ where } n \geq 1.$$

n is the length of the trace tr, denoted by $l(tr)$.

We use $tr(i)$ ($i \geq 1$) to denote the ith element of tr. We use tr^i ($i \geq 1$) to denote the prefix of tr starting from the ith element of tr, i.e., $tr^i =_{df} tr(i)tr(i+1)...tr(n)$. We use tr^b to denote the first element of tr, so $tr^b = tr(1)$; we use tr^e to denote the last element of tr, so $tr^e = tr(l(tr))$.

Definition 3.10 (Concatenation between Traces)**.** *Given two traces $tr_1 = s_1...s_n$ and $tr_2 = u_1u_2...u_m$ ($n \geq 1$, $m \geq 1$), the concatenation of tr_1 and tr_2, denoted by $tr_1 \circ tr_2$, is defined as:*

$$tr_1 \circ tr_2 =_{df} s_1...s_n u_2...u_m, \text{ provided that } s_n = u_1.$$

The concatenation operator $\circ$ can be lifted to an operator between sets of traces. Given two sets of traces T_1 and T_2, $T_1 \circ T_2$ is defined as:

$$T_1 \circ T_2 =_{df} \{tr_1 \circ tr_2 \mid tr_1 \in T_1, tr_2 \in T_2\}.$$

Note that if $tr_1^e \neq tr_2^b$, $tr_1 \circ tr_2$ is undefinable. Therefore, $\circ$ is a partial function.

The semantics of SDL formulas is defined as a Kripke frame $(\mathbf{S}, val)$ where the interpretation val maps each SP to a set of traces on $\mathbf{S}$ and each SDL formula to a set of states in $2^{\mathbf{S}}$. In the following subsections, we define the valuation $val(\phi)$ of an SDL formula ϕ step by step, by *simultaneous induction* in Def. 3.22, Def. 3.12 and Def. 3.16.

In Sect. 3.2.1, we first define the semantics of closed SPs in Def. 3.12, in which we leave the detailed definition of the semantics of parallel programs in Def. 3.16 of Sect. 3.2.2. With the semantics of closed SPs, we define the semantics of SDL formulas in Def. 3.22 of Sect. 3.2.3.

3.2.1 Valuations of Terms and Closed Programs

In this subsection, we define the valuations of terms and closed SPs in SDL.

Definition 3.11 (Valuation of Terms)**.** *The valuation of terms of SDL under a state s, denoted by val_s, is defined as follows:*

1. $val_s(x) =_{df} s(x)$;
2. $val_s(c) =_{df} c$;
3. $val_s(f(e_1, e_2)) =_{df} f(val_s(e_1), val_s(e_2))$, *where* $f \in \{+, -, \cdot, /\}$;

Note that we assume that $+, -, \cdot, /$ are interpreted as their normal meanings in arithmetic theory.

The semantics of closed SPs is defined as a set of traces in the following definition.

Definition 3.12 (Valuation of Closed SPs)**.** *The valuation of closed SPs is defined inductively based on the syntactic structure of SPs as follows:*

1. $val(\mathbf{1}) =_{df} \mathbf{S}$;
2. $val(\mathbf{0}) =_{df} \emptyset$;
3. $val(\alpha) =_{df} \{ss' \mid tr \in val_m(\alpha), tr^b = s, tr^e = s'\}$, *where $val_m(\alpha)$ is defined as:*
   - *(i)* $val_m(\epsilon) =_{df} \{ss \mid s \in \mathbf{S}\}$;
   - *(ii)* $val_m(\psi?\,.\,\alpha') =_{df} \{ss \mid s \in val(\psi)\} \circ val_m(\alpha')$, *where $val(\psi)$ is defined in Def. 3.22;*
   - *(iii)* $val_m(x := e\,.\,\alpha') =_{df} \{ss' \mid s' = s[x \mapsto val_s(e)]\} \circ val_m(\alpha')$, *where $s[y \mapsto v]$ is defined as*

$$s[y \mapsto v](z) =_{df} \left\{ \begin{array}{ll} v & \text{if } z = y \\ s(z) & \text{otherwise} \end{array} \right\};$$

4. $val(p\,;q) =_{df} val(p) \circ val(q)$;
5. $val(p \cup q) =_{df} val(p) \cup val(q)$;
6. $val(p^*) =_{df} \bigcup_{n=0}^{\infty} val^n(p)$, *where* $val^n(q) =_{df} \underbrace{val(q) \circ ... \circ val(q)}_{n}$, $val^0(q) =_{df} \mathbf{S}$ *for any* $q \in \mathbf{SP}$;
7. $val(\cap(p_1, ..., p_n)) =_{df} Par(\cap(p_1, ..., p_n))$, *where $Par(\cap(p_1, ..., p_n))$ is defined in Def. 3.16.*

The semantics of $\mathbf{1}$ is defined as the set of all traces of length 1. For any set A, we have $\mathbf{S} \circ A = A \circ \mathbf{S} = A$, which meets the intuition that $\mathbf{1}$ does nothing and consumes no time. The definitions of the semantics of the programs $\mathbf{0}$, $p;q$, $p \cup q$ and p^* are easy to understand.

The key point of the definition lies in the definitions of the semantics of the macro event α and the parallel program $\cap(p_1, ..., p_n)$. The semantics of $\cap(p_1, ..., p_n)$ is given later in Sect. 3.2.2.

The semantics of α reflects the time model of synchronous models. It consists of a set of pairs of states, we call them

macro steps. Each macro step starts at the beginning of the macro event α, and ends after the execution of the last event of α — the skip ϵ. In each macro step ss' of a macro event, the sequential executions of all events in the macro event form a trace tr with $tr^b = s$ and $tr^e = s'$. We call the execution of a micro event a *micro step*. The set of the execution traces of all micro events of α is defined by $val_m(\alpha)$. The definition of the semantics of ϵ means that from any state s, ϵ just skips the current instant and jumps to the same state without doing anything more. By simultaneous induction, we can legally put the definition of $val(\psi)$ in Def. 3.22 since ψ is a pure first-order logical formula which does not contain any SPs. The semantics of an assignment $x := e$ is defined just as in FODL.

According to Def. 3.12, it is easy to see that each trace represents one execution of an SP, and each transition between two states of a trace exactly captures the notion of one "reaction" in synchronous models. During a reaction, all events of a macro event are executed in sequence in different micro steps. Fig. 1 gives an illustration.

We can call the valuation given in Def. 3.12 the *macro-step valuation*.

3.2.2 Valuation of Parallel Programs

3.2.2.1 Definition of the Function *Par*

In this subsection, we mainly define the semantics of parallel programs, which is $Par(\cap(p_1, ..., p_n))$ in Def. 3.12. To this end, we need to define how the programs $p_1, ..., p_n$ communicate with each other at one instant. However, because of the choice program and the star program, an SP turns out to be non-deterministic, which makes the direct definition of the communication rather complicated. To solve this problem, we adopt the idea proposed in [32]. We firstly split each program p_i $(1 \leq i \leq n)$ into many deterministic programs $r_{i1}, r_{i2}, ...$, each of which captures one deterministic behaviour of p_i. Then we define the communication of the parallel program $\cap(r_{1k_1}, ..., r_{nk_n})$ (where $k_1, ..., k_n \in \mathbb{N}^+$) for each combination $r_{1k_1}, ..., r_{nk_n}$ of the deterministic programs of $p_1, ..., p_n$.

We call a deterministic program a *trec*, a name inherited from [32][1)].

Definition 3.13 (Trecs)**.** *A trec is an SP whose syntax is given as follows:*

$$trec ::= \mathbf{0} \mid \mathbf{1} \mid str \mid par,$$
$$par ::= \cap(trec, ..., trec) \mid par\,;trec \mid trec\,;par,$$
$$str ::= \alpha \mid \alpha\,;str.$$

[1)] where trec means a "tree computation".

We denote the set of all trecs as **Trec**.

A trec is an SP in which there is no choice and star programs. In a trec, a sequential program must be $\mathbf{0}$, $\mathbf{1}$ or in the form of a string str, from which the macro event at the current instant is always visible; a parallel program can be combined with a trec by the sequence operator ;.

Different from the trec defined in [32], where when a parallel program $p \cap q$ is combined with a program r by the sequence operator ;, r is always combined with p and q separately in their branches as: $(p\,;r) \cap (q\,;r)$. The trec defined in Def. 3.13, however, allows the form like $(p \cap q)\,;r$. Unlike the program model of concurrent propositional dynamic logic, in our proposed SP model, the behaviour of $(p \cap q)\,;r$ is not equivalent to that of $(p\,;r) \cap (q\,;r)$, because generally, a program r is not equivalent to a parallel program $r \cap r$. For example, consider a simple SP $r = (x := 1\,; x := x + 1)$, according to the syntax of SPs in Def. 3.3 and the restriction defined in Def. 3.7, we have $r \cap r = (y := 1\,; y := y+1) \cap (x := 1\,; x := x + 1)$, whose behaviour is not equivalent to that of r.

Intuitively, an SP has a set of trecs, each of which exactly represents one deterministic behaviour of the SP. The relation between SPs and their trecs is just like the relation between regular expressions and their languages. For example, an SP $\varsigma_1 . \epsilon\,;(\varsigma_2 . \epsilon \cup \varsigma_3 . \epsilon)$ has two trecs: $\varsigma_1 . \epsilon\,; \varsigma_2 . \epsilon$ and $\varsigma_1 . \epsilon\,; \varsigma_3 . \epsilon$. Below, we introduce the notion of the trecs of an SP to capture its deterministic behaviours.

Before introducing the trecs of an SP, we need to introduce an operator which links two trecs with the sequence operator ; in a way so that the resulted program is still a trec.

Definition 3.14 (Operator $\trianglelefteq$)**.** *Given two trecs p and q, the trec $p \trianglelefteq q$ is defined as:*

$$p \trianglelefteq q =_{df} \begin{cases} \mathbf{0} & \text{if } p = \mathbf{0} \text{ or } q = \mathbf{0} \\ q & \text{if } p = \mathbf{1} \\ p & \text{if } q = \mathbf{1} \\ \alpha\,;(p' \trianglelefteq q) & \text{if } p = \alpha\,; p' \\ p\,;q & \text{otherwise} \end{cases}$$

Intuitively, $\trianglelefteq$ links p and q in a way that does not affect the meaning of $p\,;q$. In the definition above, the first case assumes the equivalence of the behaviours between $\mathbf{0}\,;q$ (or $p\,;\mathbf{0}$) and $\mathbf{0}$. The second (resp. third) cases assumes the equivalence of the behaviours between $\mathbf{1}\,;q$ and q (resp. $p\,;\mathbf{1}$ and p). The fourth case assume the equivalence of the behaviours between $(\alpha\,;p')\,;q$ and $\alpha\,;(p'\,;q)$. All these assumptions actually hold for closed SPs according to the definition of the operator $\circ$ given above. Given p, q are trecs, it is not hard to see that the resulted program $p \trianglelefteq q$ is a trec.

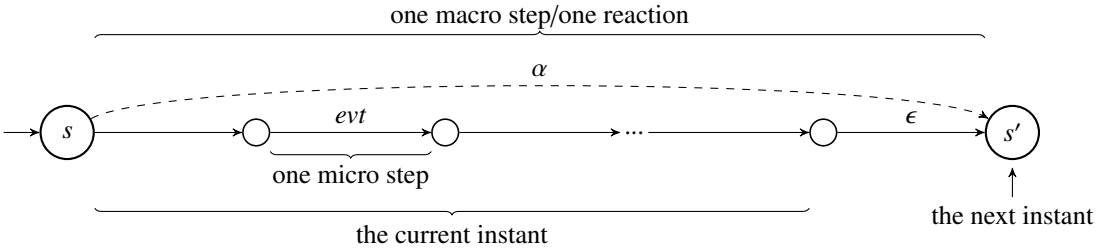

Fig. 1: Illustration of Semantics of α

Definition 3.15 (Trecs of SPs)**.** *The set of trecs of an SP is defined inductively as follows:*

1. $\tau(a) =_{df} \{a\}$*, where* $a \in \mathbf{SP}^{at}$ *is an atomic program;*
2. $\tau(p\,;q) =_{df} \{r_1 \trianglelefteq r_2 \mid r_1 \in \tau(p), r_2 \in \tau(q)\}$*;*
3. $\tau(p \cup q) =_{df} \tau(p) \cup \tau(q)$*;*
4. $\tau(p^*) =_{df} \bigcup_{n=0}^{\infty} \tau(p^n)$*, where* $p^0 =_{df} \mathbf{1}$*,* $p^n =_{df} \underbrace{p\,;p\,;...\,;p}_{n}$ *for* $n \geq 1$*;*
5. $\tau(\cap(p_1, ..., p_n)) =_{df} \{\cap(r_1, ..., r_n) \mid r_1 \in \tau(p_1), ..., r_n \in \tau(p_n)\}$.

With the notion of the trecs of an SP, as indicated at the beginning of Sect. 3.2.2, we give the definition of the function *Par* as follows.

Definition 3.16 (Function *Par*)**.** *The function* $Par(\cap(p_1, ..., p_n))$ *appeared in Def. 3.12 is defined as*

$$Par(\cap(p_1, ..., p_n)) =_{df} \bigcup_{r \in \tau(\cap(p_1,...,p_n))} val_t(r),$$

where $val_t(r)$ *is defined in Def. 3.21.*

Def. 3.16 says that the semantics of a parallel program $\cap(p_1, ..., p_n)$ is defined as the union of the semantics of all deterministic parallel programs of $\cap(p_1, ..., p_n)$.

Now it remains to give the detailed definition of $val_t(r)$ for a deterministic parallel program r of the form $\cap(p_1, ..., p_n)$, where $p_1, ..., p_n \in$ **Trec** . The central problem to define its semantics is to define how the programs $p_1, ..., p_n$ communicate with each other at one instant. Below we firstly consider the most interesting case when each program p_i $(1 \leq i \leq n)$ is of the form $\alpha_i\,;q_i$, where the macro event α_i to be executed at the current instant is visible. As will be shown later in Def. 3.21, other cases are easy to handle.

3.2.2.2 Definition of the Function *Mer*

In SPs, when n programs $p_1, ..., p_n$ are communicating at an instant, all events at the current instant are executed simultaneously. Signals emit their values by broadcasting. Once a signal is emitted, all programs observe its state immediately at the same instant. In such a scenario, it is natural to think that during one reaction, all programs $p_1, ..., p_n$ should hold a consistent view towards the state of a signal. Such a consistency is stated as the following law, which is called *logical coherence law* in Esterel [29].

> Logical coherence law: At an instant, a signal has a unique state, i.e., it is either emitted or absent, and has a unique value if emitted.

In Esterel, the programs satisfying this law are called *logically correct* [29]. Despite the simultaneous execution of all events at an instant, in SPs, the events of each macro event are also considered to be executed in a logical order that preserves data dependencies. In Esterel, the logically correct programs that follow the logical execution order in one reaction are called *constructive* [29]. In synchronous models, it is important to ensure the constructiveness when defining the semantics of the communication behaviour.

It is possible for an SP to be non-constructive. For example, let $p_1 = \bar{\varsigma_1}?\,.\,\varsigma_1!5\,.\,\epsilon$, $p_2 = \epsilon$, then the program $p_1 \cap p_2$ is logically incorrect because the signal ς_1 can neither be emitted nor absent at the current instant. If ς_1 is emitted, $\bar{\varsigma_1}$ cannot be matched, so by the logical order $\varsigma_1!5$ is never executed. If ς_1 is absent, then $\bar{\varsigma_1}$ is matched but then $\varsigma_1!5$ is executed.

In this paper, we propose a similar approach as proposed in [29] for defining a constructive semantics of the communication between SPs. The communication process, defined as the function *Mer* in Def. 3.17 below, ensures the logical coherence law while preserving the logical execution order in each macro event at the current instant. Our method is based on a recursive process of executing all events at the current instant step by step based on their micro steps. At each recursion step, we analyze all events of different macro events at the current micro step and execute them according to different cases and based on the current information obtained about signals. After the execution we update the information and use it for the next recursion step. This process is contin-

ued until all events at the current instant are executed.

In the definitions given below, we sometimes use a *pattern* of the form $p_1 \mid ... \mid p_n$ to represent a finite multi-set $\{p_1, ..., p_n\}$ of the programs $p_1, ..., p_n$, as it is easier to express certain changes made to the programs at certain positions separated by |.

Definition 3.17 (Function *Mer*). *Given a pattern of the form $A = (p_1 \mid ... \mid p_n)$ $(n \geq 2)$, where $p_i = \alpha_i ; q_i$ for any $1 \leq i \leq n$, the function $Mer(A)$ is defined as*

$$Mer(A) =_{df} rMer(A, \rho_0, R_0, Must_0, \Xi_0),$$

where $\rho_0 = \epsilon$, $R_0 = Must_0 = \Xi_0 = \emptyset$.

The recursive procedure $rMer(A, \rho, r, Must, \Xi)$, in which ρ is an atomic program, R is a pattern, Must is multi-set of signals, Ξ is a multi-set of signal-set pairs, is defined as follows:

1. *If $\alpha_i = \epsilon$ for some $1 \leq i \leq n$, then*

$$rMer((... \mid p_{i-1} \mid \alpha_i ; q_i \mid p_{i+1} \mid ...), \rho, R, Must, \Xi) =_{df} rMer((... \mid p_{i-1} \mid p_{i+1} \mid ...), \rho, (R \mid q_i), Must, \Xi);$$

2. *Else if $\alpha_i = a . \beta$ for some $1 \leq i \leq n$, where $a \in \{\psi?, x := e\}$, then*

$$rMer((p_1 \mid ... \mid \alpha_i ; q_i \mid ... \mid p_n), \rho, R, Must, \Xi) =_{df} rMer((p_1 \mid ... \mid \beta ; q_i \mid ... \mid p_n), \rho \triangleleft a, R, Must, \Xi);$$

3. *Else if $\alpha_i = \varsigma!e . \beta$ for some $1 \leq i \leq n$, then*

$$rMer((p_1 \mid ... \mid \alpha_i ; q_i \mid ... \mid p_n), \rho, R, Must, \Xi) =_{df} rMer((p_1 \mid ... \mid \beta ; q_i \mid ... \mid p_n), \rho, R, Must \uplus \{\varsigma!e\}, \Xi);$$

4. *Else if $\alpha_i = \varrho_i? . \beta_i$ for all $1 \leq i \leq n$, let $Can = getCan(A, Must)$, then*

   (i) if $\varrho_j = \hat{\varsigma}(x)$ and $Mat(\varrho_j, Must) = tt$ for some $1 \leq j \leq n$, then

$$rMer((p_1 \mid ... \mid \varrho_j . \beta_j ; q_j \mid ... \mid p_n), \rho, R, Must, \Xi) =_{df} rMer((p_1 \mid ... \mid r \mid ... \mid p_n), \rho, R, Must, \Xi \cup \{(\varsigma, Must_\varsigma)\}),$$

   where $r = (\beta_j ; q_j)[comb(\{e\}_{\varsigma!e \in Must})/x]$, $Must_\varsigma = \{\varsigma!e \mid \varsigma!e \in Must\}$;

   (ii) else if $\varrho_j = \bar{\varsigma}$ and $Mat(\varrho_j, Can) = tt$ for some $1 \leq j \leq n$, then

$$rMer((p_1 \mid ... \mid \varrho_j . \beta_j ; q_j \mid ... \mid p_n), \rho, R, Must, \Xi) =_{df} rMer((p_1 \mid ... \mid \beta_j ; q_j \mid ... \mid p_n), \rho, R, Must, \Xi),$$

   (iii) else if $Mat(\varrho_j, Must) = ff$ for all $1 \leq j \leq n$, then

$$rMer((p_1 \mid ... \mid \varrho_j . \beta_j ; q_j \mid ... \mid p_n), \rho, R, Must, \Xi) =_{df} rMer(\emptyset, \mathbf{0}, R, Must, \Xi),$$

   (iv) otherwise,

$$rMer((p_1 \mid ... \mid p_n), \rho, R, Must, \Xi) =_{df} (ff, \rho, R).$$

5. *Else if $A = \emptyset$, then*

$$rMer(A, \rho, R, Must, \Xi) =_{df} (b, \rho, R),$$

where

$$b = \begin{cases} tt, & \begin{array}{l} \text{if for any } (\varsigma, A) \in \Xi, \\ A = Must_\varsigma = \{\varsigma!e \mid \varsigma!e \in Must\} \end{array} \\ ff, & \text{otherwise} \end{cases}.$$

The function *Mer* calls a recursive function *rMer*, which executes the events at each micro step of different macro events according to different "If/Else-if" cases (the cases 1 - 5). *rMer* returns a triple (b, ρ, R), where b indicates whether the communication behaviour between the programs $p_1, ..., p_n$ is constructive or not. The parameters $\rho, R, Must, \Xi$ keep updated during the recursion process of *rMer*. ρ is the behaviour after the communication between the macro events $\alpha_1, ..., \alpha_n$. It collects all closed events executed at the current instant in a logical order. R represents the resulted n programs after the communication, corresponding to the original programs $p_1, ..., p_n$. *Must*, Ξ (and *Can*, computed from *Must* in the case 4,) maintain the information about signals that is necessary for analyzing the communication in order to obtain a constructive behaviour. *Must* and *Can* record the sets of signals that *must* emit and that *can* emit at the current instant respectively. Ξ records the set of observed signals for each signal test of the form $\hat{\varsigma}(x)$. Their usefulness will be illustrated below.

The recursion process of *rMer* is described as follows.

The case 1 is trivial. The skip ϵ has no effect on the communication between other programs so we simply remove the program and add the resulted part q_i after the execution to the set R.

In the case 2, a closed event a at the current micro step of a macro event α_i is executed. The operator $\triangleleft$, defined later in Def. 3.18, appends a to the tail of the sequence of events ρ. Note that the order to put the closed events of different macro events into ρ at each recursion step is irrelevant, because as indicated in Def. 3.7, all variables are local and do not interfere with each other. For example, in a program $(x > 1? . x := x + 1 . \epsilon) \cap (y := 1 . \epsilon)$, the logical execution orders between $x > 1?$ and $y := 1$, and between $x := x + 1$ and $y := 1$, are irrelevant. What really matters is the order between $x > 1?$ and $x := x + 1$, which is preserved in ρ.

In the case 3, we emit a signal at the current micro step of a macro event and put it in the set *Must*. Note that the

order to put the signals of different macro events into *Must* is irrelevant. This is because we do the signal-test matches (in the case 4) always after the emissions of all signals at the current micro step. This stipulation makes sure that we collect *as many as we can* the signals that *must* emit at the current instant before checking the signal tests. For example, let $p_1 = \varsigma_1!3 \,.\, \epsilon$, $p_2 = \varsigma_2!5 \,.\, \epsilon$, $p_3 = \bar{\varsigma_2}(x)? \,.\, \epsilon$, then in the program $\cap(p_1, p_2, p_3)$ the execution order of $\varsigma_1!3$ and $\varsigma_2!5$ is irrelevant, because $\bar{\varsigma_2}(x)$ will be matched after the executions of $\varsigma_1!3$ and $\varsigma_2!5$. If $\bar{\varsigma_2}(x)?$ does not wait till $\varsigma_2!5$ is executed, it would hold a wrong view towards the state of the signal ς_2 because $\bar{\varsigma_2}(x)$ is matched.

The case 4 checks the signal tests $\varrho_1, ..., \varrho_n$ after all signals at the current micro steps are emitted. Here, based on the idea originally from [29], we propose an approach, called *Must-Can approach* in this paper, to check the signal tests one by one according to two aspects of information currently obtained about signals: the set of signals (*Must*) that are certain to occur because they have already been executed in the case 3, and the set of signals (*Can*) that possibly occur because the current information about signals cannot decide their absences. The basic idea of our approach is illustrated as the following steps, corresponding to the different "If/Else-if" cases (i) - (iv) of the case 4.

(i) We firstly try to match all signal tests of the form $\hat{\varsigma}(x)$ with the current set *Must*. The result of a match is returned by the function *Mat* (defined later in Def. 3.19), which checks whether a multi-set of signals satisfy the condition of a signal test. If a signal test $\hat{\varsigma}(x)$ is successfully matched by *Must*, it means that this test must be matched at the current instant since the signal ς must be emitted at the current instant.
We substitute the variable x in the program $\beta_j \,;\, q_j$ with a value computed by a combinational function *comb*. When several signals with the same name are emitted at the same instant, their values are combined into a value by a function *comb*. For example, we can define a combinational function as $add(V) =_{df} \Sigma_{v \in V} v$. A similar approach was taken in [**?**].

(ii) If all signal tests of the form $\hat{\varsigma}(x)$ have been tried and their matches fail, we try to match the signal tests of the form $\bar{\varsigma}$ with the set *Can*, which is computed by the current set *Must* in the function *getCan* (defined later in Def. 3.20). The successful match of a signal test $\bar{\varsigma}$ means that this test must be matched at the current instant since the signal ς has no possibilities to be emitted at the current instant.

(iii) If all signal tests in the form of either $\hat{\varsigma}(x)$ or $\bar{\varsigma}$ fail in matching with the set *Must*, the communication is blocked because it means that the current emissions of signals have decided the mismatches of all signal tests at the current micro steps.

(iv) If none of the match conditions in the above 3 steps holds, for example, $Mat(\bar{\varsigma}, Can)$ always holds for a signal test $\bar{\varsigma}$, then we conclude that the communication behaviour between the programs $p_1, ..., p_n$ is not constructive and end the procedure *rMer* by returning b as false.

For example, let $p_1 = \bar{\varsigma_1}? \,.\, \varsigma_2!5 \,.\, \epsilon$, $p_2 = \bar{\varsigma_2}? \,.\, \varsigma_1!3 \,.\, \epsilon$, consider the communication of the program $p_1 \cap p_2$, in the case 4, we have $Must = \emptyset$ and $Can = \{\varsigma_2!5, \varsigma_1!3\}$. We cannot proceed because $Mat(\bar{\varsigma_1}, Can) = Mat(\bar{\varsigma_2}, Can) = ff$. In fact, this program is logical incorrect because it allows two possible states of signals: 1) ς_1 is emitted and ς_2 is absent, and 2) ς_1 is absent and ς_2 is emitted. Let $q_1 = \bar{\varsigma_2}? \,.\, \varsigma_4 \,.\, \epsilon$, $q_2 = \hat{\varsigma_1}? \,.\, \varsigma_3 \,.\, \varsigma_2 \,.\, \epsilon$ and $q_3 = \hat{\varsigma_3}? \,.\, \varsigma_1$, in the communication of the program $\cap(q_1, q_2, q_3)$, $Must = \emptyset$. It is easy to see that though $\bar{\varsigma_2}$ is possible to be matched and so ς_4 can be emitted, the tests $\hat{\varsigma_1}$ and $\hat{\varsigma_3}$ will never be matched. So $Can = \{\varsigma_4\}$ and $Mat(\bar{\varsigma_2}, Can) = tt$. Hence this program is constructive.

The case 5 ends the recursion procedure when all macro events $\alpha_1, ..., \alpha_n$ at the current instant are executed. At this end, we need to check whether all programs $p_1, ..., p_n$ hold a consistent view towards the value of each signal. Only when a signal test $\hat{\varsigma}(x)?$ observes the set $Must_\varsigma$ of all emissions of ς, we can guarantee that all programs agree with the value of ς, i.e., $comb(\{e\}_{\varsigma'!e \in Must_\varsigma})$. For example, let $p_1 = \varsigma_1!3 \,.\, \hat{\varsigma_2}? \,.\, \varsigma_1!5 \,.\, \epsilon$, $p_2 = \hat{\varsigma_1}(x)? \,.\, \varsigma_2 \,.\, \epsilon$, then the program $p_1 \cap p_2$ does not follow the logical coherence law if we choose *comb* as the function *add* (defined above in the case 4(i)), because the value received by the signal test $\hat{\varsigma_1}(x)?$ is 3, while the value of the signal ς_1 at the current instant is $add(3, 5)$, which is 8.

3.2.2.3 Definitions of $\triangleleft$, *Mat* and *getCan*

The operator $\triangleleft$ and the functions *Mat* and *getCan* used in the function *Mer* above are defined as follows.

Definition 3.18 (Operator $\triangleleft$)**.** *Given an event α and an event $a \in \{\psi?, x := e\}$, $\alpha \triangleleft a$ is defined as:*

$$\alpha \triangleleft a =_{df} \left\{ \begin{array}{ll} a \,.\, \epsilon & if\ \alpha = \epsilon \\ b \,.(\alpha' \triangleleft a) & if\ \alpha = b \,.\, \alpha' \end{array} \right\}.$$

The operator $\triangleleft$ appends an event a to the tail of a macro event α before ϵ.

Definition 3.19 (Function *Mat*)**.** *Given a signal test ϱ and a multi-set of signals Y, the function $Mat(\varrho, Y)$ is defined as*

$$Mat(\varrho, Y) =_{df} \left\{ \begin{array}{l} tt \ \textit{if} \ \varrho = \hat{\varsigma}(x) \ \textit{and} \ \varsigma = \varsigma' \ \textit{for some} \ \varsigma'!e \in Y \\ tt \ \textit{if} \ \varrho = \bar{\varsigma} \ \textit{and} \ \varsigma \neq \varsigma' \ \textit{for all} \ \varsigma'!e \in Y \\ ff \ \textit{otherwise} \end{array} \right\}.$$

The function *Mat* checks whether a multi-set of signals Y matches a signal test ϱ. It returns true if Y matches ϱ, and returns false otherwise.

Definition 3.20 (Function *getCan*)**.** *Given a pattern of the form $A = \alpha_1 | ... | \alpha_n$, where $\alpha_i = \varrho_i ? . \beta_i$ $(1 \leq ß \leq n)$, and a multi-set of signals M, then*

$$getCan(A, M) =_{df} rgetCan(A, M, Can_0),$$

where $Can_0 = \emptyset$. The function $rgetCan(A, M, Can)$ is recursively defined as follows:

1. *if $\alpha_i = \epsilon$ for some $1 \leq i \leq n$, then*

$$rgetCan((... | \alpha_{i-1} | \alpha_i | \alpha_{i+1} | ...), M, Can) =_{df} rgetCan((... | \alpha_{i-1} | \alpha_{i+1} | ...), M, Can);$$

2. *else if $\alpha_i = a . \beta$ for some $1 \leq i \leq n$, where $a \in \{\psi ?, x := e\}$, then*

$$rgetCan((\alpha_1 | ... | \alpha_i | ... | \alpha_n), M, Can) =_{df} rgetCan((\alpha_1 | ... | \beta | ... | \alpha_n), M, Can);$$

3. *else if $\alpha_i = \varsigma!e . \beta$ for some $1 \leq i \leq n$, then*

$$rgetCan((\alpha_1 | ... | \alpha_i | ... | \alpha_n), M, Can) =_{df} rgetCan((\alpha_1 | ... | \beta | ... | \alpha_n), M, Can \uplus \{\varsigma!e\});$$

4. *else if $\alpha_i = \hat{\varsigma}(x) ? . \beta$ and $Mat(\hat{\varsigma}(x), Can \cup M) = tt$ for some $1 \leq i \leq n$, then*

$$rgetCan((\alpha_1 | ... | \alpha_i | ... | \alpha_n), M, Can) =_{df} rgetCan((\alpha_1 | ... | \beta | ... | \alpha_n), M, Can);$$

5. *else if $\alpha_i = \bar{\varsigma} ? . \beta$ and $Mat(\bar{\varsigma}, M) = tt$ for some $1 \leq i \leq n$, then*

$$rgetCan((\alpha_1 | ... | \alpha_i | ... | \alpha_n), M, Can) =_{df} rgetCan((\alpha_1 | ... | \beta | ... | \alpha_n), M, Can);$$

6. *otherwise*

$$rgetCan((\alpha_1 | ... | \alpha_i | ... | \alpha_n), M, Can) =_{df} Can;$$

The function *getCan* returns a set *Can* of signals that are possibly emitted in the communication of n macro events $\alpha_1, ..., \alpha_n$ at the current instant. In *getCan*, whether a signal test is possible to be matched is judged according to the set of signals M that must emit and the current set *Can*. In the case 4, if a signal ς must or can be emitted in the current environment, then we can conclude that $\hat{\varsigma}(x)$ can be matched. In the case 5, a signal test $\bar{\varsigma}$ can possibly be matched only when we cannot decide whether the signal ς is emitted in the current environment. Other cases are easy to understand and we omit their explanations.

3.2.2.4 More Analysis on the Function *Mer*

As the definition of *Mer* in Def. 3.17 is quite complex, one might doubt that whether it is well defined. We have the following proposition.

Proposition 3.1. *Mer defined in Def. 3.17 is a function, i.e., given a pattern A, it returns a unique result.*

We omit its formal proof and only give a brief explanation as below. In fact, it is not hard to see that in all "If/Else-if" cases, the order of the macro events α_i $(1 \leq i \leq n)$ to be dealt with is irrelevant. The situation for the case 1 is trivial. We have already explained for the cases 2 and 3 above. Similar to the case 3, the order of checking ϱ_j in the case 4(i) has nothing to do with the result because the case 4(ii) can only be checked when all signal tests of the form $\hat{\varsigma}(x)$ have been dealt with in the case 4(i). Similar analysis can be given for the cases in the function *getCan* of Def. 3.20.

We show that the communication behaviour at each instant produced by *Mer* is constructive, in the sense that it is logically correct and follows the logical execution order in each reaction.

Proposition 3.2 (Constructiveness of *Mer*)**.** *At each instant, the behaviour ρ produced by Mer(A), when $b = tt$, is constructive, i.e., ρ satisfies that*

1. *it is logically correct, in the sense that*
   (a) *a signal of ρ is emitted iff it is in the final "Must" set, named $Must_f$, of Mer;*
   (b) *each signal test ϱ of ρ must coincide with $Must_f$, i.e.,*
   - *If $\varrho = \hat{\varsigma}(x)$, $\hat{\varsigma}(x)$ is matched iff ς is in $Must_f$ (with $\varsigma!e \in Must_f$ for some value e);*
   - *If $\varrho = \bar{\varsigma}$, $\bar{\varsigma}$ is matched iff ς is not in $Must_f$.*
2. *it preserves the logical execution order of the programs in A.*

We show the main idea of the proof as follows.

The proof of Prop. 3.2 (the main idea). It is obvious that the logical execution order of A is preserved in *Mer* at each instant, for each "If/Else-if" case is based on the form of the current micro event. Therefore, for a macro event $\alpha =$

$...a...b.....\epsilon$ where a micro event a is in front of another micro event b, a is always dealt with before b in *Mer*.

Now we prove the logical correctness of ρ. (a) is due to the case 3, which is the only place where the set *Must* can be modified. For (b), if ϱ is of the form $\hat{\varsigma}(x)$, according to the case 4(i), it is impossible that ς is in $Must_f$ but $\hat{\varsigma}(x)$ is not matched. Because whenever ς is added into *Must*, we can always match $\hat{\varsigma}(x)$ by the case 4(i). If ϱ is of the form $\bar{\varsigma}$, on one hand, if $\bar{\varsigma}$ is matched, according to the case 4(ii), ς is not in the current set *Can*. So ς is neither in the set *Can* later, because as the procedure of *Mer* continues, according to the definition of *getCan* in Def. 3.20, the set *Can* would become smaller since *Must* would become larger so that less tests of the form $\bar{\varsigma'}$ would be matched in the case 5 of Def. 3.20. Therefore ς cannot be in $Must_f$. On the other hand, if ς is not in $Must_f$, then there must exist a set *Can* s.t. ς is not in *Can*, otherwise, it is not hard to see that during the procedure of *Mer*, either we get that $b = ff$ for some test $\bar{\varsigma'}$ cannot be matched, or we get that ς is in some *Must*. Hence $\bar{\varsigma}$ is matched in *Mer*. □

3.2.2.5 Definition of the Function $val_t(r)$

With the function *Mer*, we define the valuation of the trecs $\cap(p_1, ..., p_n)$ as follows.

Definition 3.21 (Valuation of the Trecs $\cap(p_1, ..., p_n)$)**.** *The valuation of a trec $\cap(p_1, ..., p_n) \in$ **Trec**, denoted as $val_t(\cap(p_1, ..., p_n))$, is recursively defined as follows:*

1. *If $p_i = \mathbf{0}$ for some $1 \leq i \leq n$, then*

$$val_t(\cap(p_1, ..., p_n)) =_{df} val(\mathbf{0});$$

2. *Else if $p_i = \mathbf{1}$ for some $1 \leq i \leq n$, then*

$$val_t(\cap(p_1, ...p_{i-1}, p_i, p_{i+1}, ..., p_n)) =_{df} val_t(\cap(p_1, ..., p_{i-1}, p_{i+1}, ..., p_n));$$

3. *Else if $p_i = \cap(r_1, ..., r_n); q$ for some $1 \leq i \leq n$ and $val_t(\cap(r_1, ..., r_n)) = val(r)$, where r is the program computed in the procedure $val_t(\cap(r_1, ..., r_n))$, which is either $\mathbf{0}$ or in the form of $\alpha; \cap(r'_1, ..., r'_m)$, then*

$$val_t(\cap(p_1, ..., p_i, ..., p_n)) =_{df} val_t(\cap(p_1, ..., r \trianglelefteq q, ..., p_n));$$

4. *Else if $p_i = \alpha_i; q_i$ for all $1 \leq i \leq n$, then*

$$val_t(\cap(p_1, ..., p_n)) =_{df} val(\alpha; \cap(q'_1, ..., q'_n)) \text{ if } b = tt,$$

where $(b, \alpha, (q'_1 \mid ... \mid q'_n)) = Mer(\alpha_1; q_1 \mid ... \mid \alpha_n; q_n)$.

The cases 1 and 2 are easy to understand. In the case 3, when one of the trec p_i contains a parallel subprogram $\cap(r_1, ..., r_n)$, we first compute the valuation of this subprogram, which returns the valuation of a program r of the form $\mathbf{0}$ or $\alpha; \cap(r'_1, ..., r'_m)$. Then we replace p_i with the trec $r \trianglelefteq q$, the latter has a macro event α visible at the current instant.

In the case 4, we merge n programs $\alpha_1; q_1, ..., \alpha_n; q_n$ in the function *Mer* as described above, which returns the merged closed macro event α and the resulted n programs $q'_1, ..., q'_n$ after the communication at the current instant. Note that val_t is a partial function and it rejects the parallel programs that do not follow the consistency law 1 at the current instant. Our semantics makes sure that SPs must follow the consistency law 1, while preserving the data dependencies induced by the logical order between micro events. Our semantics is actually a *constructive semantics* following [29].

We call an SP p a *well-defined* SP if it has a semantics $val(p)$. Unless specially pointing out, all SPs discussed in the rest of the paper are well defined.

Note that the function val_t is well defined since a trec is always finite without the star program.

To see the rationality of our definition of the semantics of SPs in this section, we state the next proposition, which shows that the trecs of an SP exactly capture all behaviours of the SP.

Proposition 3.3. *For any SP p,*

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

The proof of Def. 3.3 is given in Appendix 9.1.

3.2.3 Valuation of Formulas

The semantics of SDL formulas is defined as a set of states in the following definition.

Definition 3.22 (Valuation of SDL Formulas)**.** *The valuation of SDL formulas is given inductively as follows:*

1. $val(tt) =_{df} \mathbf{S}$;
2. $val(\theta(e_1, e_2)) =_{df} \{s \mid s \in \mathbf{S}, \theta(val_s(e_1), val_s(e_2)) \text{ is true}\}$;
3. $val(\neg\phi) =_{df} \mathbf{S} - val(\phi)$;
4. $val(\phi \wedge \psi) =_{df} val(\phi) \cap val(\psi)$;
5. $val(\forall x.\phi) =_{df} \{s \mid \text{for all } n \in \mathbb{Z}, s \in val(\phi[n/x])\}$;
6. $val([p]\phi) =_{df} \{s \mid \text{for all } tr \in val(p) \text{ with } tr^b = s,\ tr^e \in val(\phi)\}$;
7. $val([p]\Box\phi) =_{df} \{s \mid \text{for all } tr \in val(p) \text{ with } tr^b = s,\ tr \in val_\pi(\Box\phi)\}$, *where the valuation $val_\pi(\Box\phi)$ of a trace formula $\Box\phi$ is defined as*

$$val_\pi(\Box\phi) =_{df} \{tr \mid tr(i) \in val(\phi) \text{ for all } i \geq 1\}.$$

The items 1-5 are normal definitions for first-order formulas. The formula $[p]\phi$ describes the partial correctness of a program, since all traces of p are finite, tr^e always exists. $[p]\Box\phi$ captures safety properties in synchronous models that hold at each instant. The semantics of a temporal formula $\Box\phi$ is given as val_π.

We introduce the satisfaction relation between states and SDL formulas.

Definition 3.23 (Satisfaction Relation $\models$)**.** *The satisfaction relation between a state $s \in \mathbf{S}$ and an SDL formula ϕ, denoted as $s \models \phi$, is defined s.t.*

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

We say ϕ is valid, denoted by $\models \phi$, if for all $s \in \mathbf{S}$, $s \models \phi$ holds.

4 Proof System of SDL

In this section, we propose a sound and relatively complete proof system for SDL to support verification of reactive systems based on theorem proving. We propose compositional rules for sequential SPs which transform a dynamic formula step by step into AFOL formulas according to the syntactic structure of programs. We propose rewrite rules for parallel SPs which transform a parallel SP into a sequential one so that the proof of a dynamic formula that contains parallel SPs can be realized by the proof of a dynamic formula that only contains sequential SPs. For those parallel SPs which contain star programs and cannot be directly transformed into sequential ones by simple rewrites, we propose an transformation procedure (Algo. 1) according to the Brzozowski's procedure for transforming an automaton into a regular expression. Since the communication of parallel SPs is deterministic, a parallel SP can be easily rewritten into a sequential one based on the process given in the function *Mer* (Def. 3.17) and the state space of the sequential SP keeps a linear size w.r.t that of the parallel SP.

In Sect. 4.1 we first introduce a logical form called *sequent* which allows us to express deductions in a convenient way. In Sect. 4.2 and Sect. 4.3, we propose rules for sequential and parallel SPs respectively. In Sect. 4.4, we introduce other rules for first-order logic (FOL).

4.1 Sequent Calculus

A sequent [33] is of the form

$$\Gamma \Rightarrow \Delta,$$

where Γ and Δ are finite multisets of formulas, called *contexts*. The sequent means the formula

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

i.e., if all formulas in Γ hold, then one of formulas in Δ holds.

When $\Gamma = \emptyset$, we denote the sequent as $\cdot \Rightarrow \Delta$, meaning the formula $tt \rightarrow \bigvee_{\phi\in\Delta} \phi$. When $\Delta = \emptyset$, we denote it as $\Gamma \Rightarrow \cdot$, meaning the formula $\bigwedge_{\phi\in\Gamma} \phi \rightarrow ff$.

A rule in sequent calculus is of the form

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

where $\Gamma_1 \Rightarrow \Delta_1$,...,$\Gamma_n \Rightarrow \Delta_n$ are called *premises*, while $\Gamma \Rightarrow \Delta$ is called a *conclusion*. The rule means that if $\Gamma_1 \Rightarrow \Delta_1$,...,$\Gamma_n \Rightarrow \Delta_n$ are valid (in the sense of Def. 3.23), then $\Gamma \Rightarrow \Delta_n$ is valid.

We simply write rules $\dfrac{\Gamma_1 \Rightarrow \psi_1, \Delta_1 \quad ... \quad \Gamma_n \Rightarrow \psi_n, \Delta_n}{\Gamma \Rightarrow \phi, \Delta}$ and $\dfrac{\Gamma', \psi \Rightarrow \Delta'}{\Gamma, \phi_1, ..., \phi_n \Rightarrow \Delta}$ as

$$\frac{\psi_1 \quad ... \quad \psi_n}{\phi} \text{ and } \frac{\phi_1 \quad ... \quad \phi_n}{\psi}$$

respectively if $\Gamma_1 = ... = \Gamma_n = \Gamma' = \Gamma$ and $\Delta_1 = ... = \Delta_n = \Delta' = \Delta$ hold. In fact, the rule $\dfrac{\psi_1 \quad ... \quad \psi_n}{\phi}$ just means that the formula $\bigwedge_{i=1}^n \psi_i \rightarrow \phi$ is valid in the sense of Def. 3.23 (cf. Prop. 9.1 in Appendix 9.1).

When both rules $\dfrac{\psi}{\phi}$ and $\dfrac{\phi}{\psi}$ exist, we simply use a single rule of the form

$$\frac{\psi}{\phi}$$

to represent them. It means that the formula $\psi \leftrightarrow \phi$ is valid.

4.2 Compositional Rules for Sequential SPs

As shown in Table 1, the rules for sequential SPs are divided into two types. The rules of type (a) are initially proposed in this paper for special primitives in SPs, whereas the rules of type (b) are inherited from FODL [24] and differential temporal dynamic logic (DTDL) [23].

Explanations of each rule in Table 1 are given as follows.

The rules of the type (a) are for atomic programs. Rule $(\alpha, \Box\phi)$ says that proving that ϕ is satisfied at each instant of the macro event α, is equal to prove that ϕ holds before and after the execution of α. This is because a macro event α only concerns two instants, i.e., the instants before and after the execution of α. The time does not proceed in a macro event. The rules $(\psi?)$ and $(x := e)$ deal with the micro events in a

macro event. In rule ($\psi?$), if the test $\psi?$ does not hold, the formula $[\psi?\,.\,\alpha]\phi$ is always true because there is no trace in the program $\psi?\,.\,\alpha$. Rule ($x := e$) is similar to the rule for assignment in FODL. Note that the assignment $x := e$ also affects on the micro steps after it (i.e. α) and this reflects the data dependencies between different micro events of a macro event. In rule (ϵ), since ϵ only skips the current instant, it does not affect a state property ϕ. Rule (**1**) says that **1** neither consumes time nor does anything, so only one instant is involved. Thus $[\mathbf{1}]\xi$ equals to that ϕ holds at the current instant. Rule (**0**) says that $[\mathbf{0}]\xi$ is always true because there is no trace in **0**.

Except for the rules ($[], gen$) and ($\langle\rangle, gen$), all rules of type (b) are for composite programs. The rules ($;, \Box\phi$) and ($*, \Box\phi$) are inherited from DTDL [24]. The rules ($;, \phi$) and ($;, \Box\phi$) are due to the fact that any trace of $p\,;q$ is formed by concatenating a trace of p and a trace q, while rule ($\cup$) is based on the fact that any trace of $p \cup q$ is either a trace of p or a trace of q. Note that the reverse of the rule ($;, \Box\phi$), i.e., $\dfrac{[p\,;q]\Box\phi}{[p]\Box\phi \wedge [p][q]\Box\phi}$, does not hold, which directly leads to that the whole proof system of SDL is not relatively complete. We will analyze this problem in detail in Sect. 5.2. Rule ($*, \Box\phi$) converts the proof of a temporal formula $\Box\phi$ to the proof of a state formula $[p]\Box\phi$. This is useful because then we can apply other rules such as (ind) and (con) which are only designed for a state formula ϕ. Rule ($*$) is based on the semantics of p^*, which means that either does not execute p (i.e. **1**), or executes p for 1 or more than 1 times (i.e. $p; p^*$). The rules ($[], gen$) and ($\langle\rangle, gen$) are for eliminating the dynamic parts $[p]$, $\langle p\rangle$ of a formula during deductions. They are used in deriving the rules ($[*]$) and ($\langle * \rangle$) below and are necessary for the relatively completeness of the whole proof system.

$$\frac{\psi \quad \forall(\psi \to [p]\psi) \quad \forall(\psi \to \phi)}{[p^*]\phi}\ ([*]) \tag{1}$$

$$\frac{\exists v \geq 0.\psi(v) \quad \forall((v > 0 \wedge \psi(v)) \to \langle p\rangle\psi(v-1)) \quad \forall(\psi(0) \to \phi)}{\langle p^*\rangle\phi}\ (\langle * \rangle) \tag{2}$$

The rules (ind) and (con) are mathematical inductions for proving properties of star programs p^*. Rule (ind) means that to prove that $\phi \to [p^*]\phi$ holds at a state, we need to prove that $\phi \to [p]\phi$ holds at any state. Rule (con) has a similar meaning as (ind). The main difference is that in (con) a variable v is introduced as an indication of the termination of p^*. The rules (ind) and (con) are mainly used in theories. In practical verification, the rules ($[*]$) and ($\langle * \rangle$) above are applied for eliminating the star operator $*$, where ψ is often known as a *loop invariant* of p^*. ($[*]$) and ($\langle * \rangle$) can be derived from (ind) and (con), refer to [24] for more details.

4.3 Rewrite Rules for Parallel SPs

Table 2 gives the rewrite rules for parallel SPs, which are divided into three types and one single rule. The rules of the types $(a), (b), (c)$ perform rewrites for one step, whereas rule ($\cap, seq$) rewrites a parallel program as a whole into a sequential one through a series of steps in the algorithm *Brz*.

Two rules of type (a) are structural rules for rewriting an SDL formula or an SP according to the rewrites of its parts. They rely on the definition of *program holes* given below.

Definition 4.1 (Program Holes)**.** *Given a formula ϕ, a program hole of ϕ, denoted as $\phi\{\_\}$, is defined in the following grammar:*

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

where a program hole $p\{\_\}$ of an SP p is defined as:

$$p\{\_\} ::= \_ \mid p\{\_\}\,; p \mid p\,; p\{\_\} \mid p\{\_\} \cup p \mid p \cup p\{\_\} \mid (p\{\_\})^* \mid \cap(p, ..., p\{\_\}, ..., p).$$

We call _ a place which can be filled by a program.

Given a program hole $\phi\{\_\}$ (resp. $p\{\_\}$) and an SP q, $\phi\{q\}$ (resp. $p\{q\}$) is the formula (resp. the program) obtained by filling the place _ of $\phi\{\_\}$ (resp. $p\{\_\}$) with q.

The rules ($r1$) and ($r2$) say that the rewrites between SPs preserves the semantics of SDL formulas and SPs. Rule ($r1$) means that if p can be rewritten by q, then we can replace p by q anywhere of ϕ without changing the semantics of ϕ. Rule ($r2$) has a similar meaning. Note that $p\{q\}$, $p\{r\}$, q, r can also be open SPs, and in this case we do not care about their semantics (because they do not have one).

The rewrites in the rules of type (a) come from the rules of the types (b) and (c). The rules of type (b) are for sequential programs. They are all based on the semantics of sequential SPs. For example, rule (**1**, ;) is based on the fact that $val(\mathbf{1}\,; p) = val(\mathbf{1}) \circ val(p) = val(p)$. The rules of type ($c$) are for parallel programs. They are based on the semantics of parallel programs given in Sect. 3.2.2. See Appendix 9.1 for the proofs of their soundness.

In a parallel program $\cap(p_1, ..., p_n)$, if the program p_i ($1 \leq i \leq n$) is in the form q^* or $q^*\,; r$, consecutively applying the rules of the types (a), (b) and (c) may generate an infinite

$$\frac{\phi \wedge [\alpha]\phi}{[\alpha]\Box\phi}\ (\alpha,\Box\phi) \qquad \frac{\psi \rightarrow [\alpha]\phi}{[\psi?\,.\,\alpha]\phi}\ (\psi?) \qquad \frac{([\alpha]\phi)[e/x]}{[x := e\,.\,\alpha]\phi}\ (x{:=}e) \qquad \frac{\phi}{[\epsilon]\phi}\ (\epsilon) \qquad \frac{\phi}{[\mathbf{1}]\xi}\ 1\ (\mathbf{1}) \qquad \frac{tt}{[\mathbf{0}]\xi}\ (\mathbf{0})$$

(a) rules special in SDL

$$\frac{[p][q]\phi}{[p\,;q]\phi}\ (;,\phi) \qquad \frac{[p]\Box\phi \wedge [p][q]\Box\phi}{[p\,;q]\Box\phi}\ (;,\Box\phi) \qquad \frac{[p]\xi \wedge [q]\xi}{[p \cup q]\xi}\ (\cup) \qquad \frac{[p^*][p]\Box\phi}{[p^*]\Box\phi}\ (*,\Box\phi) \qquad \frac{[\mathbf{1} \cup p\,;p^*]\xi}{[p^*]\xi}\ (*)$$

$$\frac{\forall(\phi \rightarrow [p]\phi)}{\phi \rightarrow [p^*]\phi}\ 2\ (ind) \qquad \frac{\forall((v > 0 \wedge \phi(v)) \rightarrow \langle p\rangle\phi(v-1))}{(\exists v \geq 0.\phi(v)) \rightarrow \langle p^*\rangle\phi(0)}\ 3\ (con) \qquad \frac{\forall(\phi \rightarrow \psi)}{[p]\phi \rightarrow [p]\psi}\ ([],gen) \qquad \frac{\forall(\phi \rightarrow \psi)}{\langle p\rangle\phi \rightarrow \langle p\rangle\psi}\ (\langle\rangle,gen)$$

(b) rules from other dynamic logics

[1] $\xi \in \{\phi, \Box\phi\}$
[2] $\forall(\phi) =_{df} \forall x_1.\forall x_2.....\forall x_n.\phi$, where $x_1, ..., x_n$ are the set of all free variables in ϕ
[3] the variable v does not appear in p

Table 1: Rules for Sequential SPs

$$\frac{\phi\{q\}}{\phi\{p\}}\ \text{if } p \rightsquigarrow q\ ^{1}\ (r1) \qquad p\{q\} \rightsquigarrow p\{r\}\ \text{if } q \rightsquigarrow r\ ^{2}\ (r2)$$

(a) structural rewrite rules

$\mathbf{1}\,;p \rightsquigarrow p$ $(\mathbf{1},;)$ — $\mathbf{1}^* \rightsquigarrow \mathbf{1}$ $(\mathbf{1},*)$ — $\mathbf{0}\,;p \rightsquigarrow \mathbf{0}$ $(\mathbf{0},;)$
$\mathbf{0}^* \rightsquigarrow \mathbf{1}$ $(\mathbf{0},*)$ — $(p\,;q)\,;r \rightsquigarrow p\,;(q\,;r)$ $(;,ass)$ — $p\,;(q \cup r) \rightsquigarrow (p\,;q) \cup (p\,;r)$ $(;,dis1)$
$(q \cup r)\,;p \rightsquigarrow (q\,;p) \cup (r\,;p)$ $(;,dis2)$ — $(p \cup q) \cup r \rightsquigarrow p \cup (q \cup r)$ $(\cup,ass)$ — $p^* \rightsquigarrow \mathbf{1} \cup p;p^*$ $(*,exp)$

(b) rewrite rules for sequential programs

$\cap(..., p, \mathbf{1}, q, ...) \rightsquigarrow \cap(..., p, q, ...)$ $(\cap,\mathbf{1})$ $\quad \cap(..., p, \mathbf{0}, q, ...) \rightsquigarrow \mathbf{0}$ $(\cap,\mathbf{0})$ $\quad \cap(..., p \cup q, ...) \rightsquigarrow \cap(..., p, ...) \cup \cap(..., q, ...)$ $(\cap,dis)$
$\cap(\alpha_1\,;q_1, ..., \alpha_n\,;q_n) \rightsquigarrow \alpha\,;\cap(q_1', ..., q_n')$ if $(b, \alpha, (q_1' \mid ... \mid q_n')) = Mer(\alpha_1\,;q_1 \mid ... \mid \alpha_n\,;q_n)$ and $b = tt$ $(\cap,mer)$

(c) rewrite rules for parallel programs

$\cap(p_1, ..., p_n) \rightsquigarrow Brz(\cap(p_1, ..., p_n))$ if $\cap(p_1, ..., p_n)$ is well defined $(\cap,seq)$

[1] p and q are closed programs; $\phi\{\_\}$ is a program hole of the formula ϕ
[2] $p\{\_\}$ is a program hole of the program p; the reduction $q \rightsquigarrow r$ is from the rules of the types (b) and (c)

Table 2: Rewrite Rules for Parallel SPs

proof tree. An example is shown in Fig. 2, where we merge several derivation steps into one by listing all the rules applied during these steps. In the proof tree of this example, we can see that the node (1) is the same as the root node so the proof procedure will never end.

To avoid this situation, we propose rule $(\cap, seq)$. Rule $(\cap, seq)$ reduces a parallel program into a sequential one by the procedure *Brz* (Algorithm 1), which follows the Brzozowski's method [34] that transforms an NFA (nondeterministic finite automaton), expressed as a set of equations, into a regular expression. This process relies on Arden's rule [35] to solve the equations of regular expressions. Before introducing the procedure *Brz*, we firstly explain why Arden's rule can apply to SPs.

In fact, it is easily seen that a sequential SP is a regular expression, whose semantics is exactly the set of sequences the regular expression represents. The sequence operator ; represents the *concatenation* between sequences of two sets. The choice operator $\cup$ represents the *union* of two sets of sequences. The loop operator $*$ represents the *star* operator that is applied to a set of sequences. An event α is a *word* of a regular expression, representing a set of sequences with the minimal length (here in SP the minimal length is 2). The empty program $\mathbf{1}$ is the *empty string* of a regular expression, representing a set of sequences with "zero length" , i.e., sequences that do not change other sequences when are concatenated to them (here in SP, the "zero length" is 1 due to the definition of the operator $\circ$). The halt program $\mathbf{0}$ is the *empty set* of a regular expression, representing an empty set of sequences.

With this fact, Arden's rule obviously holds for sequential SPs, as stated as the following proposition. In the below of this paper, we sometimes use $p \equiv q$ to mean that two programs p and q are semantically equivalent, i.e., $val(p) = val(q)$.

$$\dfrac{\dfrac{...}{\mathbf{1}\cap\mathbf{1}}\qquad \dfrac{...}{\mathbf{1}\cap(\hat{\varsigma}?\,.\,\epsilon)\,;(\hat{\varsigma}?\,.\,\epsilon)^*}\qquad \dfrac{...}{(\varsigma\,.\,\epsilon)\,;(\varsigma\,.\,\epsilon)^*\cap\mathbf{1}}\qquad \dfrac{\dfrac{(1):\quad(\varsigma\,.\,\epsilon)^*\cap(\hat{\varsigma}?\,.\,\epsilon)^*}{\epsilon\,;((\varsigma\,.\,\epsilon)^*\cap(\hat{\varsigma}?\,.\,\epsilon)^*)}\,{\scriptstyle(;,\phi),(\epsilon)}}{(\varsigma\,.\,\epsilon)\,;(\varsigma\,.\,\epsilon)^*\cap(\hat{\varsigma}?\,.\,\epsilon)\,;(\hat{\varsigma}?\,.\,\epsilon)^*}\,{\scriptstyle(\cap,mer)}}{\dfrac{(\mathbf{1}\cup(\varsigma\,.\,\epsilon)\,;(\varsigma\,.\,\epsilon)^*)\cap(\mathbf{1}\cup(\hat{\varsigma}?\,.\,\epsilon)\,;(\hat{\varsigma}?\,.\,\epsilon)^*)}{(\varsigma\,.\,\epsilon)^*\cap(\hat{\varsigma}?\,.\,\epsilon)^*}\,{\scriptstyle(*,exp),(r1)}}\,{\scriptstyle(\cap,dis),(r1),(\cup)}$$

Fig. 2: An Example of Inifinite Proof Trees

Algorithm 1 Procedure *Brz*

1: **procedure** BRZ($\cap(p_1, ..., p_n)$) /*$\cap(p_1, ..., p_n)$ is well defined*/

2: let $l_1 = \cap(p_1, ..., p_n)$, by consecutively applying the rules ($r2$), and the rules of the types (b) and (c) in Table 2, we can reduce l_1 as the following form:

$$l_1 \rightsquigarrow b_1 \cup \alpha_{11}\,; l_1 \cup ... \cup \alpha_{1n}\,; l_n, \qquad (3)$$

where $n \geq 0$ ($l_1 \rightsquigarrow b_1$ when $n = 0$), b_1 is a sequential program, $l_2, ..., l_n$ are parallel programs. From the reduction above we can build an equation, namely

$$l_1 \equiv b_1 \cup \alpha_{11}\,; l_1 \cup ... \cup \alpha_{1n}\,; l_n.$$

3: continuing this reduction procedure for the programs $l_2, ..., l_n$, we can then build n equations:

$$l_1 \equiv b_1 \cup \alpha_{11}\,; l_1 \cup ... \cup \alpha_{1n}\,; l_n, \quad (1)$$
$$...$$
$$l_n \equiv b_n \cup \alpha_{n1}\,; l_1 \cup ... \cup \alpha_{nn}\,; l_n, \quad (n)$$

where $b_1, ..., b_n$ are sequential programs, $l_1, ..., l_n$ are taken as n variables.

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

5: transform the equation (k) into the form $l_k \equiv p \cup q\,; l_k$

6: by Prop. 4.1, obtain $l_k \equiv q^*\,; p$ from $l_k \equiv p \cup q\,; l_k$

7: substitute l_k on the right of the other equations ($k-1$),..., (1) with $q^*\,; p$

8: **return** l_1

Proposition 4.1 (Arden's Rule in SPs)**.** *Given any sequential SPs p and q with $q \not\equiv \mathbf{1}$, $X \equiv q^*\,; p$ is the unique solution of the equation $X \equiv p \cup q\,; X$.*

Prop. 4.1 can be proved according to the semantics of SPs, we omit it here.

The procedure *Brz* transforms a parallel program into a sequential one. Algorithm 1 explains how it works. At line 2, it is easy to see that by using the rules ($r2$) and the rules of the types (b) and (c) in Table 2, a parallel program can always be reduced into a form as the righthand side of the reduction (3). We can actually prove it by induction on the syntactic structure of the parallel program. At line 3, we can always build a finite number of n equations because whichever rule we use for reducing a parallel program $\cap(p_1, ..., p_n)$, e.g. $\cap(p_1, ..., p_n) \rightsquigarrow q$, the reduced form q only contains the symbols in the original form $\cap(p_1, ..., p_n)$. Therefore, the total number of reduced expressions from a parallel program $\cap(p_1, ..., p_n)$ by consecutively applying those rules must be finite because $\cap(p_1, ..., p_n)$ only contains a finite number of symbols.

By taking all parallel programs $l_1, ..., l_n$ as variables, the process of solving the n equations is attributed to Brzozowski's standard process of solving n regular equations, as stated from line 4 - line 8. On each iteration, Arden's rule is applied to eliminate the kth variable l_k. Finally, all $n-1$ variables l_n,...,l_2 are eliminated and we return l_1 as the result.

4.4 Rules for FOL Formulas

Table 3 shows the rules for FOL formulas. Since they are all common in FOL, we omit the discussion of them here. For convenience of analysis in the rest of the paper, we also give the rules for $\vee, \rightarrow$ and $\exists$ in Table 4, although they can be derived by the rules in Table 3.

At the end of this section, we introduce the notion of *deductive relation* in SDL.

Definition 4.2 (Deductive Relation $\vdash$)**.** *Given a SDL formula ϕ and a multi-set of formulas Φ, we say ϕ is derived by Φ, denoted as $\Phi \vdash \phi$, iff the sequent $\Phi \Rightarrow \phi$ can be derived by the rules in Table 1, 2, 3.*

If Φ is an empty set, we simply denote $\Phi \vdash \phi$ as $\vdash \phi$.

5 Analysis of Soundness and Completeness of SDL Calculus

In this section, we analyze the soundness and completeness of SDL calculus proposed in previous sections in Sect. 5.1 and 5.2 respectively. We give a complete proof of the soundness of SDL, and the relative completeness of SDL under a certain condition in Appendix 9.1 and 9.2 respectively.

5.1 Soundness of SDL Calculus

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

Theorem 5.1 (Soundness of SDL)**.** *Given an SDL formula ϕ, if $\vdash \phi$, then $\models \phi$.*

To prove Theorem 5.1, it is equivalent to prove that each rule in Table 1, 2, 3 is sound.

$$\frac{}{\Gamma, \phi \Rightarrow \phi, \Delta}\ (ax) \qquad \frac{\Gamma \Rightarrow \phi, \Delta \quad \Gamma, \phi \Rightarrow \Delta}{\Gamma \Rightarrow \Delta}\ (cut) \qquad \frac{\Gamma, \neg\phi \Rightarrow \Delta}{\Gamma \Rightarrow \phi, \Delta}\ (\neg r) \qquad \frac{\Gamma \Rightarrow \neg\phi, \Delta}{\Gamma, \phi \Rightarrow \Delta}\ (\neg l)$$

$$\frac{\Gamma \Rightarrow \phi, \Delta \quad \Gamma \Rightarrow \psi, \Delta}{\Gamma \Rightarrow \phi \wedge \psi, \Delta}\ (\wedge r) \qquad \frac{\Gamma, \phi, \psi \Rightarrow \Delta}{\Gamma, \phi \wedge \psi \Rightarrow \Delta}\ (\wedge l) \qquad \frac{\Gamma \Rightarrow \phi[y/x], \Delta}{\Gamma \Rightarrow \forall x.\phi, \Delta}\ {}^{1}\ (\forall r) \qquad \frac{\Gamma, \phi[e/x] \Rightarrow \Delta}{\Gamma, \forall x.\phi \Rightarrow \Delta}\ (\forall l)$$

[1] y is a new variable with respect to Γ, ϕ and Δ

Table 3: Rules of FOL

$$\frac{\Gamma \Rightarrow \phi, \psi, \Delta}{\Gamma \Rightarrow \phi \vee \psi, \Delta}\ (\vee r) \qquad \frac{\Gamma, \phi \Rightarrow \Delta \quad \Gamma, \psi \Rightarrow \Delta}{\Gamma, \phi \vee \psi \Rightarrow \Delta}\ (\vee l) \qquad \frac{\Gamma, \phi \Rightarrow \psi\Delta}{\Gamma \Rightarrow \phi \rightarrow \psi, \Delta}\ (\rightarrow r) \qquad \frac{\Gamma \Rightarrow \phi, \Delta \quad \Gamma, \psi \Rightarrow \Delta}{\Gamma, \phi \rightarrow \psi \Rightarrow \Delta}\ (\rightarrow l)$$

$$\frac{\Gamma \Rightarrow \phi[e/x], \Delta}{\Gamma \Rightarrow \exists x.\phi, \Delta}\ (\exists r) \qquad \frac{\Gamma, \phi[y/x] \Rightarrow \Delta}{\Gamma, \exists x.\phi \Rightarrow \Delta}\ {}^{1}\ (\exists l)$$

[1] y is a new variable with respect to Γ, ϕ and Δ

Table 4: Other Rules of FOL

For the rules in Table 1, we only need to prove the soundness of the rules of type (a), since all rules of type (b) are inherited from FODL and DTDL. Based on the semantics of SDL formulas, the soundness of the rules of type (a) is given in Appendix 9.1. For the proofs of the soundness of the rules $(;, \Box\phi)$ and $(*, \Box\phi)$, one can refer to [23]. For the proofs of the soundness of other rules of type (b), one can refer to [24].

The soundness of a rewrite rule means that given two closed SPs p and q, if $p \rightsquigarrow q$, then $val(p) = val(q)$. In Appendix 9.1, we prove the soundness of all rewrite rules of Table. 2. We firstly prove the soundness of the rules of the types (b) and (c) (see Prop. 9.3 and 9.4) based on their semantics, then we prove the soundness of the rules $(r1)$ and $(r2)$ (see Prop. 9.8 and 9.6) by induction on the structures of the program holes and the level of a rewrite relation (as defined in Def. 9.1). The soundness of rule $(\cap, seq)$ is direct according to Arden's rule (see Prop. 9.7).

The soundness of the FOL rules in Table 3 and 4 are directly from FOL.

5.2 Relative Completeness of SDL under a Certain Condition

Since SDL includes AFOL in itself, due to Gödel's incompleteness theorem [36], SDL is not complete. Therefore, we mainly analyze the relative completeness [27] of SDL.

Given an SDL formula ϕ, we use $\vdash^{+}\phi$ to represent that ϕ is derivable in the proof system consisting of all rules of Table 1, 2, 3 and all tautologies in AFOL as axioms. Intuitively, $\vdash^{+}\phi$ means that ϕ can be transformed into a set of pure AFOL formulas in the proof system of SDL. The relative completeness of SDL is stated as follows:

$$\text{for any SDL formula } \phi, \text{ if } \models \phi, \text{ then } \vdash^{+}\phi.$$

However, SDL is not relatively complete. The reason is that there exist sequential programs of the form $p\,;q$ which cannot be derived by the rule $(;, \Box\phi)$. For example, let $p = (x := 1 \,.\, \epsilon \cup x := 2 \,.\, \epsilon)$, $q = (x \leq 1? \,.\, x := 0 \,.\, \epsilon)$, then the formula $[p\,;q]\Box x \leq 1$ is true, but it cannot be derived using the rule $(;, \Box\phi)$, since $[p]\Box x \leq 1$ does not hold.

General speaking, for any programs p, q, formula $[p\,;q]\Box\phi \rightarrow [p]\Box\phi \wedge [p][q]\Box\phi$ is not always valid. Essentially, this is because that some trace of p can be blocked by some test statement or the halt program $\mathbf{0}$ of q so that it is no longer the part of a trace of $p\,;q$. For example, in the example above, the traces of $x := 2 \,.\, \epsilon$ in p are blocked by the test $x \leq 1?$ of q, so that they are not the part of traces of $p\,;q$.

Below we discuss in which condition the proof system proposed in Sect. 4 is a relatively complete one. we make a restriction on the set of SPs to be discussed by introducing the notion of *non-blocking* SPs as follows. Whether there exists a relatively complete proof system for all SPs is an open problem.

Definition 5.1 (Non-blocking SPs)**.** *The set of non-blocking SPs, denoted by NB*(**SP**)*, is the maximum set of closed SPs which satisfies that for any sequential program $p\,;q$,*

$$val(p) = \{tr \mid \exists tr'.tr' \in val(q) \wedge tr \circ tr' \in val(p\,;q)\}.$$

The satisfaction condition in Def. 5.1 means that for each trace tr of p, there exists a trace tr' of q such that $tr \circ tr'$ is a trace of $p\,;q$, so each trace of p is non-blocked in $p\,;q$. With this property, we can prove that the reverse of the rule $(;, \Box\phi)$ is sound, which is stated as follows.

Proposition 5.1 (The Reverse of the Rule $(;, \Box\phi)$)**.** *In NB*(**SP**)*, the reverse of the rule* $(;, \Box\phi)$ *is sound, i.e., formula*

$$[p\,;q]\Box\phi \to [p]\Box\phi \wedge [p][q]\Box\phi$$

is valid for any non-block programs $p, q \in NB(\mathbf{SP})$.

We omit its proof, which is the same as the proof given in [23], based on the fact stated in Def. 5.1.

With Prop. 5.1, now we show that, when we only consider the set of non-blocking SPs, the proof system built in Table 1, 2, 3 is relatively complete.

Theorem 5.2 (Relative Completeness of SDL in $NB(\mathbf{SP})$)**.** *If we limit our discussion to the set of non-block SPs NB*(**SP**)*, then given an SDL formula* ϕ*,*

$$\text{if } \models \phi, \text{ then } \vdash^{+} \phi.$$

The main idea behind the proof of Theorem 5.2 follows the proof of FODL initially proposed in [25], where a theorem called "the main theorem" (Theorem 3.1 of [25]) was proposed as the skeleton of the whole proof. We give our proof by augmenting that main theorem with one more condition for the temporal dynamic formulas $[p]\Box\phi$, which are new in SDL. Our main theorem is stated as follows.

Theorem 5.3 (The "Main Theorem")**.** *Under the domain of NB*(**SP**)*, SDL is relatively complete if the following conditions hold:*

(i) SDL formulas are expressible in AFOL.

(ii) For any AFOL formulas $\phi^{\flat}$ *and* $\psi^{\flat}$*, if* $\models \phi^{\flat} \to op\,\psi^{\flat}$*, then* $\vdash^{+} \phi^{\flat} \to op\,\psi^{\flat}$.

(iii) For any SDL formulas ϕ *and* ψ*, if* $\vdash^{+} \phi \to \psi$*, then* $\vdash^{+} op\,\phi \to op\,\psi$.

(iv) For any AFOL formulas $\phi^{\flat}$ *and* $\psi^{\flat}$*, if* $\models \phi^{\flat} \to [p]\Box\psi^{\flat}$*, then* $\vdash^{+} \phi^{\flat} \to [p]\Box\psi^{\flat}$*; and if* $\models \phi^{\flat} \to \langle p\rangle\Diamond\psi^{\flat}$*, then* $\vdash^{+} \phi^{\flat} \to \langle p\rangle\Diamond\psi^{\flat}$.

In the above conditions, $op \in \{[p], \langle p\rangle, \forall x, \exists x\}$*. We use* $\phi^{\flat}$ *to stress that* ϕ *is an AFOL formula.*

The only differences between Theorem 5.2 and the main theorem in [25] are: 1) we replace all "FODL formulas" with "SDL formulas" in the context; 2) we add a new condition (the condition (iv)) to our theorem for temporal dynamic formula $[p]\Box\phi$ and its dual form $\langle p\rangle\Diamond\phi$. With Theorem 5.3, we can prove the relative completeness of SDL based on these 4 conditions (See Appendix 9.2). During the process of the proof the new-adding condition (iv) plays a central role when proving the relative completeness of SDL formulas of the forms $[p]\Box\phi$ and $\langle p\rangle\Diamond\phi$.

The rest remains to prove the 4 conditions of Theorem 5.3. For the condition (i), the expressibility of SDL formulas in AFOL is directly from that of FODL formulas because, intuitively, a sequential program in SDL can be seen as a regular program in FODL if we ignore the differences between macro steps and micro steps which only play their roles in parallel SPs. The proofs of the conditions (ii) and (iii) mainly follow the corresponding proofs of FODL in [25], where the main difference is that in the proof steps we also need to consider the cases when an SP is a special primitive (such as the macro event) and when an SP is a parallel program. The proof of the condition (iv) mainly follows the idea behind the proof of the condition (*ii*) but is adapted to fit the proofs of the temporal dynamic formulas $[p]\Box\phi$ and $\langle p\rangle\Diamond\phi$. Appendix 9.2 gives the proofs of these 4 conditions.

6 SDL in Specification and Verification of SyncCharts

In this section, we show the potential of SDL as a theoretical framework for specifying and verifying synchronous models through a toy example. Among many synchronous models, in this paper, we choose SyncChart [30] as an example. SyncChart has the same semantics as Esterel and has been embedded into the famous industrial tool SCADE. SyncChart captures the most essential features of synchronous models and is in the form of automata, which is easy to be transformed into SPs.

6.1 Modelling Basic SyncCharts

In this subsection, we use SPs to model basic syncCharts. As we will see, the process of modelling a basic syncChart as an SP is quite straightforward as SP models can rightly capture the features of synchronous models. In the following, we give two examples to show that SPs can express syncCharts. The first example is about a sequential system, while the second one is about a concurrent system.

Currently, SPs only support modelling the basic syncCharts, which do not include advanced synchronous features such

as preemption, hierarchy and so on (cf. [30]). However, in terms of expressiveness, considering the basic syncCharts is enough because those advanced features essentially do not inhence the expressive power of syncCharts. They are just for engineers to model in a more convenient and neater way.

For the first example, we consider a simple circuit as considered in [30], called a "frequency divider", which is modelled as an syncChart, named "FDIV2", shown in Fig. 3. The frequency dividor waits for a first occurrence of a signal T, and then emits a signal C at every other occurrence of T. Its syncChart FDIV2 consists of two states, with the *off* state as the initial state. Each transition in a syncChart represents one reaction of a reactive system, i.e., an instant at which all events occur simultaneously in a logical order. It exactly corresponds to a macro step in SPs. The label on a transition is called the "trigger and effect", which is in the form of *trigger*/*effect*, representing the actions of reading input signals and sending output signals respectively at each instant. For example, in FDIV2, the label "T/C" means that at an instant, if the signal T is triggered, the signal C is emitted. A label corresponds to a macro event in SPs.

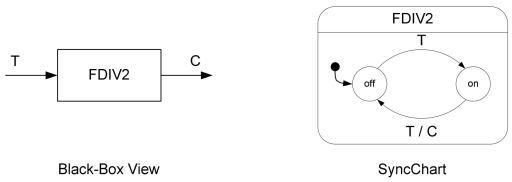

Fig. 3: A Frequency Divider

The behaviour of FDChart is that

(1) At the state *off*, the syncChart waits for a signal T and moves to the state *on*;
(2) at the state *on*, it waits for a signal T and emits a signal C at the same instant, and moves to the state *off*.

In SDL, let ς_t, ς_c represent the signals T and C respectively, we can model FDChart as an open SP

$$P_{FD} = ((\bar{\varsigma}_t?\,.\,\epsilon)^*\,;\hat{\varsigma}_t?\,.\,\epsilon\,;(\bar{\varsigma}_t?\,.\,\epsilon)^*\,;\hat{\varsigma}_t?\,.\,\varsigma_c\,.\,\epsilon)^*,$$

where the event $\hat{\varsigma}_t?\,.\,\epsilon$ models the transition from the state *off* to the state *on*, on which the signal T is triggered, while the event $\hat{\varsigma}_t?\,.\,\varsigma_c\,.\,\epsilon$ models the transition from the state *on* to the state *off*, on which both the signals T and C are triggered. The logical order between $\hat{\varsigma}_t$ and ς_c in the event $\hat{\varsigma}_t?\,.\,\varsigma_c\,.\,\epsilon$ indicates the "trigger - effect" relation between the two signals T and C. The program $(\bar{\varsigma}_t?\,.\,\epsilon)^*$ means to wait the signal ς_t without doing anything.

FDChart looks simpler than the program P_{FD} because it omits the behaviour of "waiting the signal" on its graph (, which should be a self-loop added on the state *off* or *on*). In SPs, we can actually define a syntactic sugar for this behaviour: for any signal ς and event α,

$$\hat{\varsigma}(x)??\alpha =_{df} (\bar{\varsigma}\,.\,\epsilon)^*\,;\hat{\varsigma}(x)?\,.\,\alpha, \tag{4}$$

which means that the program waits ς until it it emitted, and then proceeds as α (at the same instant). With this shorthand, P_{FD} can be rewritten as

$$P_{FD} = (\hat{\varsigma}_t??\epsilon\,;\hat{\varsigma}_t??(\varsigma_c\,.\,\epsilon))^*.$$

In the second example, we consider a simple circuit from [30], called a "binary counter", which is modelled as a syncChart in Fig. 4. The binary counter reads every signal T and counts the number of occurrences of T by outputting the signals $B0$ and $B1$ that represent the bit: the present of a signal represents 1, while the absent of a signal represents 0. The syncChart of the binary counter, called "Cnt2", is a parallel syncChart and is obtained by a parallel composition of two syncCharts that model circuits called "flip-flops" [30]. A dashed line is to separate the two syncCharts running in parallel. The execution mechanism of parallel syncCharts is the same as that of parallel SPs. At each reaction, transitions in different syncCharts can be triggered simultaneously in a logical order. Therefore, the behaviour of the parallel syncChart is deterministic.

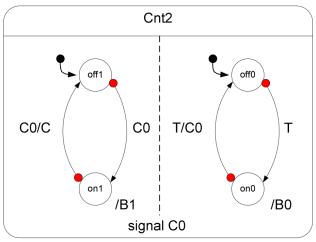

Fig. 4: A Binary Counter

In syncCharts, a state can be tagged with a *trigger*/*effect* label and special types of transitions were introduced to decide when the label on a state is triggered. This does not increase the expressiveness of syncCharts but can reduce the number of states and transitions a syncChart has. The special transitions (with a red circle at the tail of an arrow) appeared in Cnt2 are called "strong abortion transitions" [30]. When a strong abortion transition is triggered, the label of the state that the transition is entering is triggered simultaneously, while the label of the state the transition is exiting cannot be triggered.

The behaviour of Cnt2 is given as follows:

- The right syncChart:

(1) At the state *off0*, the syncChart waits for the signal T and moves to the state *on0*, at the same instant, the signal $B0$ is emitted.
(2) At the state *on0*, the syncChart waits for the signal T and if it is emitted, the signal $C0$ is emitted, and then the syncChart moves to the state *off0*.

- The left syncChart:
  (1) At the state *off1*, the syncChart waits for the signal $C0$ from the right syncChart and moves to the state *on1*, at the same instant, the signal $B1$ is emitted.
  (2) At the state *on1*, the syncChart waits for the signal $C0$ from the right syncChart and if it is emitted, the signal $C0$ is emitted, and then the syncChart moves to the state *off0*.

Let $\varsigma_t, \varsigma_{c0}, \varsigma_{b0}, \varsigma_{b1}, \varsigma_c$ represent the signals $T, C0, B0, B1$ and C respectively, Cnt2 can be modelled as an SP as follows:

$$\begin{aligned} P_{BC} &= P_l \cap P_r, \\ P_r &= (\hat{\varsigma_t}??(\varsigma_{b0} \,.\, \epsilon)\,;\hat{\varsigma_t}??(\varsigma_{c0} \,.\, \epsilon))^*, \\ P_l &= (\hat{\varsigma_{c0}}??(\varsigma_{b1} \,.\, \epsilon)\,;\hat{\varsigma_{c0}}??(\varsigma_c \,.\, \epsilon))^*. \end{aligned}$$

P_l, P_r model the left and right syncCharts respectively.

6.2 Specifying and Proving Properties in SyncCharts

We consider a simple safety property for the first example discussed above, which says

"whenever the signal ς_c is emitted, the signal ς_t is emitted".

Since P_{FD} is an open SP, we assume an environment E for P_{FD} which generally emits ς_t or does nothing at each instant. To collect the information about the signals ς_t and ς_c, we define two observers O_t and O_c, which listen to these two signals and record their states within the local variables x and y at each instant.

The property is thus specified in an SDL formula as follows:

$$\phi_{FD} = (x = 0 \wedge y = 0) \rightarrow [\cap(P_{FD}, E, O_t, O_c)]\Box(y = 1 \rightarrow x = 1),$$

where

$$\begin{aligned} E &= (\varsigma_t \,.\, \epsilon \cup \epsilon)^*, \\ O_t &= (\hat{\varsigma_t}? \,.\, x := 1 \,.\, \epsilon \cup \bar{\varsigma_t}? \,.\, x := 0 \,.\, \epsilon)^*, \\ O_c &= (\hat{\varsigma_c}? \,.\, y := 1 \,.\, \epsilon \cup \bar{\varsigma_c}? \,.\, y := 0 \,.\, \epsilon)^*. \end{aligned}$$

Fig. 5 shows the process of proving ϕ_{FD}. Starting at the root node $\cdot \Rightarrow \phi_{FD}$, the proof tree is constructed by applying the rules of the SDL calculus reversely step by step. By rule $(\cap, seq)$, the program $\cap(P_{FD}, E, O_t, O_c)$ can be rewritten into a sequential program P through a Brzozoski's procedure. To save spaces, we omit most of the branches of the proof tree by "...", and we merge several derivations into one by listing all the rules applied during these derivations on the right side of the derivation. ϕ_{FD} is proved to be true if the AFOL formula at each leaf node of the proof tree is valid (indicated by a $\surd$ at each leaf node).

7 Related Work

7.1 Verification Frameworks and Techniques for Synchronous Models

Previous verification approaches for synchronous models are mainly based on model checking. Different specification languages, such as synchronous observers [12, 13], LTL [14] and clock constraints [15], were applied to capture the safety properties. They were transformed into target models where reachability analysis was made. For synchronous programming languages, the process of reachability analysis is often embedded into their compilers, for instance, cf. [4, 16].

Despite for its decidability and efficiency on small size of state spaces, model checking suffers from the notorious state-explosion problem. Recent years automated/interactive theorem proving, as a complement technique to model checking, has been gradually applied for analysis and verification of synchronous models in different aspects. One hot research work is to use theorem provers like Coq [37] to mechanize and verify the compiling processes of synchronous programming languages, so that the equivalence between compiled code and source code can be guaranteed [38, 39]. SAT/SMT solving, as a fully automatic verification technique, was used for checking the time constraints in synchronous programming languages such as Lustre [40] and Signal [41]. In [42], a type theory was proposed to provide a compile-time framework for analyzing and checking time properties of Signal, through the inference of refinement types.

Rather than targeting on explicit synchronous languages, our proposed formalism focuses on a more general synchronous model SPs, extended from regular programs that are suitable for compositional reasoning. As indicated in Sect. 1, SPs capture the essential features of synchronous models and ignore those which do not support compositional reasoning. Different from synchronous languages that are totally deterministic, SPs have the extra expressive power to support describing programs at an abstract level with the non-deterministic operator $\cup$. Similar to the type-theory approach [42], instead of directly using SAT/SMT solving, SDL

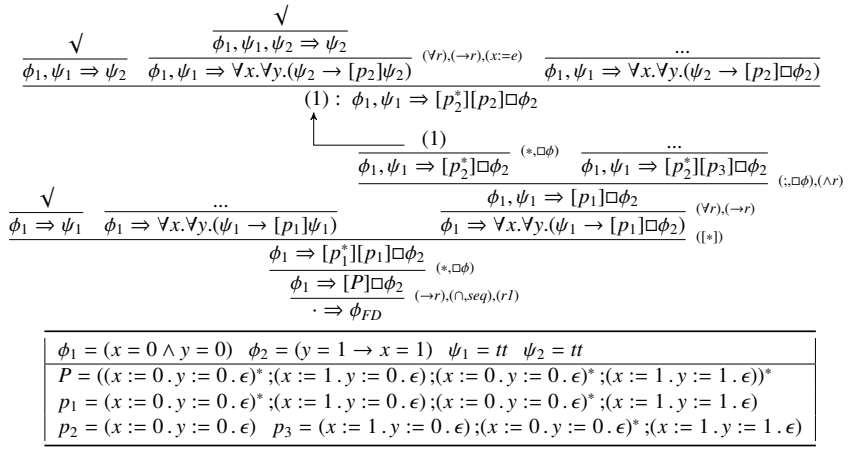

Fig. 5: The Derivation Procedure of ϕ_{FD}

provides compositional rules for decomposing SPs according to their syntactic structures, so as to divide a big verification problem into small SMT-solving problems in derivation processes.

[43] proposed an equation theory for pure Esterel. There, term rewrite rules were built for describing the constructive semantics of Esterel so that two different Esterel programs can be formally compared and their equivalences can be formally reasoned about. [44] proposed a so-called *synchronous effects logic* for verifying temporal properties of pure Esterel programs. A Hoare-style forward proving process was developed to compute the behaviours of Esterel programs as synchronous effects. Then a term rewriting system was proposed to verify the temporal properties, which are also expressed as synchronous effects, of Esterel programs.

Compared to [43, 44], the verification of SDL is not solely based on term rewriting, but also based on a Hoare-style program verification [17] process. Instead of verifying by checking the equivalences or refinement relations between two programs, we reason about the satisfaction relation between a program and a logic formula, in a form $[p]\phi$ or $[p]\Box\phi$. In SPs, the rewrite rules built in Table 2 for reducing parallel SPs play a similar role as the rewrite rules defined in [43, 44] for symbolically executing parallel Esterel programs. The synchronous effects used to capture the behaviours of Esterel programs in [44] was similar to the form of SPs.

In [45], a Hoare logic calculus was proposed for the synchronous programming language Quartz. In that work, the authors manage to prove a Quartz program in a simple form in which there is no parallel compositions and all events in a macro step are collected together in a single form called *synchronous tuple assignment* (STA). Such a simple form can be obtained either by manual encoding or by the compiler of Quartz in an automatic way. The STA proposed there actually corresponds to the macro event in SPs. Compared to [45], SP is not an synchronous language but a more general synchronous model. Except for state properties, SDL also supports verifying safety properties with the advantage brought by dynamic logic to support formulas of the form $[p]\Box\phi$.

7.2 Dynamic Logic

Dynamic logic was firstly proposed by V.R. Pratt [18] as a formalism for modelling and reasoning about program specifications. The syntax of SDL is largely based on and extends that of FODL [25] and that of its extension to concurrency [32]. Temporal dynamic logical formulas of the form $[p]\Box\phi$ were firstly proposed in Process logic [46]. [26] studied a first-order dynamic logic containing formulas of this form (there, it was written as $[|p|]\phi$) and initially proposed a relatively complete proof system for it. Inspired from [26], in [23], the author introduces the form $[p]\Box\phi$ into his DTDL and proposed the compositional rules for $[p]\Box\phi$. The semantics of SDL mainly follows the trace semantics of process logic. In SDL, we inherit formulas of the form $[p]\Box\phi$ from DDTL to express safety properties of synchronous models, and adopt the compositional rules from DTDL (i.e. the rules $(;, \Box\phi)$ and $(*, \Box\phi)$) in order to build a relatively complete proof system for SDL.

Many variations of dynamic logic have been proposed for modelling and verifying different types of programs and system models. For instance, Y.A. Feldman proposed a probabilistic dynamic logic PrDL for reasoning about probabilistic programs [20]; [21] proposed the Java Card Dynamic Logic for verifying Java programs; In [22] and [23], the Differential

Dynamic Logic (DDL) and DTDL were proposed respectively for specifying and verifying hybrid systems. DDL introduced differential equations in the regular programs of FODL to capture physical dynamics in hybrid systems. The time model of DDL is continuous. In DDL, a discrete event (e.g. $x := e$) does not consume time, while a continuous event (i.e. a differential equation) continuously evolves until some given conditions hold. Compared to DDL, the time model of SDL is discrete and is reflected by the macro events. SDL mainly focuses on capturing the features of synchronous models and preserving them in program models during theorem-proving processes.

An attempt to build a dynamic logic for synchronous models was made in [47], where a clock-based dynamic logic (CDL) was proposed to specify and verify specific clock specifications in synchronous models. SDL differs from CDL in the following two main points. Firstly, in CDL, all events occurring at an instant are disordered, while in SDL, all micro events of a macro event are executed in a logical order. So SDL is able to capture data dependencies, which are an important feature of synchronous models. Secondly, in CDL, we propose formulas of the form $[p]\xi$, where ξ is a clock constraint such as $c_1 \prec c_2$, to express the clock specifications of a program p. While in SDL, we introduce a more general form $[p]\Box\phi$, which can express not only clock constraints, but also other safety properties. Compared to CDL, SDL is a logic that is more general and more expressive.

8 Conclusion and Future Work

In this paper, we mainly propose a dynamic logic — SDL — for specifying and verifying synchronous models based a theorem proving technique. We define the syntax and semantics of SDL, build a constructive semantics for parallel SPs, and propose a sound proof system for SDL, which is also relatively complete under a certain condition. We show the potential of SDL to be used in specifying and verifying synchronous models through a toy example.

As for future work, we mainly focus on mechanizing SDL in the theorem prover Coq and apply SDL in specifying and verifying more interesting examples rather than the toy examples in this paper.

Acknowledgements This work is partially supported by the Youth Project of National Science Foundation of China (No. 62102329), the Project of National Science Foundation of Chongqing (No. cstc2021jcyj-bsh0204), the Capacity Development Grant of Southwest University (No. SWU116007) and the Key Projects of National Science Foundation of China (No. 61732019, No. 62032019).

References

1. Harel D, Pnueli A. On the development of reactive systems. In: Logics and Models of Concurrent Systems. 1985, 477 – 498
2. Benveniste A, Caspi P, Edwards S A, Halbwachs N, Le Guernic P, de Simone R. The synchronous languages 12 years later. Proceedings of the IEEE, 2003, 91(1): 64 – 83
3. Benveniste A, Guernic P L, Jacquemot C. Synchronous programming with events and relations: the signal language and its semantics. Science of Computer Programming, 1991, 16(2): 103 – 149
4. Halbwachs N, Caspi P, Raymond P, Pilaud D. The synchronous data flow programming language lustre. Proceedings of the IEEE, 1001, 79(9): 1305 – 1320
5. Berry G, Gonthier G. The Esterel synchronous programming language: design, semantics, implementation. Science of Computer Programming, 1992, 19(2): 87 – 152
6. Espiau B, Coste-Maniere E. A synchronous approach for control sequencing in robotics application. In: Proceedings of the IEEE International Workshop on Intelligent Motion Control, Vol. 2. 1990, 503 – 508
7. Berry G, Bouali A, Fornari X, Ledinot E, Nassor E, de Simone R. Esterel: a formal method applied to avionic software development. Science of Computer Programming, 2000, 36(1): 5 – 25
8. Gamatié A, Gautier T, Besnard L. Modeling of avionics applications and performance evaluation techniques using the synchronous language signal. Electronic Notes in Theoretical Computer Science, 2004, 88: 87 – 103
9. Qian J, Liu J, Chen X, Sun J. Modeling and verification of zone controller: The scade experience in china's railway systems. In: IEEE/ACM 1st International Workshop on Complex Faults and Failures in Large Software Systems (COUFLESS). 2015, 48 – 54
10. Lee E. The past, present and future of cyber-physical systems: A focus on models. Sensors, 2015, 15: 4837 – 4869
11. Lohstroh M, Lee E A. Deterministic actors. In: Forum on Specification and Design Languages (FDL). 2019, 1 – 8
12. Halbwachs N, Lagnier F, Raymond P. Synchronous observers and the verification of reactive systems. In: Proceedings of the Third International Conference on Methodology and Software Technology: Algebraic Methodology and Software Technology, AMAST '93. 1993, 83 – 96
13. Pilaud D, Halbwachs N. From a synchronous declarative language to a temporal logic dealing with multiform time. In: Proceedings of a Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems. 1988, 99 – 110
14. Jagadeesan L J, Puchol C, Von Olnhausen J E. Safety property verification of esterel programs and applications to telecommunications software. In: Computer Aided Verification. 1995, 127 – 140
15. André C, Mallet F. Specification and verification of time requirements with CCSL and Esterel. In: Proceedings of the 2009 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for

Embedded Systems. 2009, 167 – 176
16. Berry G, Cosserat L. The Esterel synchronous programming language and its mathematical semantics. In: Seminar on Concurrency. 1985, 389 – 448
17. Apt K R, de Boer F S, Olderog E R. Verification of Sequential and Concurrent Programs. Texts in Computer Science, Springer, 2009
18. Pratt V R. Semantical considerations on Floyd-Hoare logic. In: Annual IEEE Symposium on Foundations of Computer Science (FOCS). 1976, 109 – 121
19. Hoare C A R. An axiomatic basis for computer programming. Commun. ACM, 1969, 12(10): 576 – 580
20. Feldman Y A, Harel D. A probabilistic dynamic logic. Journal of Computer and System Sciences, 1984, 28(2): 193 – 215
21. Rustan K, Leino M. Verification of Object-Oriented Software. The KeY Approach - Foreword by K. Rustan M. Leino. Lecture Notes in Computer Science (LNCS), Springer, 2007
22. Platzer A. Differential dynamic logic for verifying parametric hybrid systems. In: International Conference on Theorem Proving with Analytic Tableaux and Related Methods (TABLEAUX). 2007, 216 – 232
23. Platzer A. A temporal dynamic logic for verifying hybrid system invariants. In: Logical Foundations of Computer Science (LFCS). 2007, 457 – 471
24. Harel D, Kozen D, Tiuryn J. Dynamic Logic. MIT Press, 2000
25. Harel D. First-Order Dynamic Logic. Vol. 68 of Lecture Notes in Computer Science (LNCS), Springer, 1979
26. Beckert B, Schlager S. A sequent calculus for first-order dynamic logic with trace modalities. In: Automated Reasoning. 2001, 626 – 641
27. Cook S A. Soundness and completeness of an axiom system for program verification. SIAM Journal on Computing, 1978, 7(1): 70 – 90
28. Milner R. Calculi for synchrony and asynchrony. Theoretical Computer Science, 1983, 25(3): 267 – 310
29. Berry G. The constructive semantics of pure Esterel. https://www-sop.inria.fr/members/Gerard.Berry/Papers/EsterelConstructiveBook.pdf, 1999
30. André C. Semantics of Synccharts. Tech. Rep. ISRN I3S/RR–2003–24–FR, I3S Laboratory, Sophia-Antipolis, France, 2003
31. Schneider K, Brandt J. Quartz: a synchronous language for model-based design of reactive embedded systems. Handbook of Hardware/-Software Codesign, 2017: 29 – 58
32. Peleg D. Communication in concurrent dynamic logic. Journal of Computer and System Sciences, 1987, 35(1): 23–58.
33. Gentzen G. Untersuchungen über das logische schließen. Ph.D. thesis, NA Göttingen, 1934
34. Brzozowski J A. Derivatives of regular expressions. Journal of the ACM, 1964, 11(4): 481 – 494
35. Arden D N. Delayed-logic and finite-state machines. In: SWCT (FOCS), IEEE Computer Society. 1961, 133 – 151
36. Gödel K. über formal unentscheidbare sätze der principia mathematica und verwandter systeme. Monatshefte für Mathematik und Physik, 1931, 38(1): 173 – 198
37. Bertot Y, Castéran P. Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions, Texts in Theoretical Computer Science. An EATCS Series, Springer, 2004
38. Bourke T, Brun L, Dagand P E, Leroy X, Pouzet M, Rieg L. A formally verified compiler for Lustre. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). 2017, 586 – 601
39. Berry G, Rieg L. Towards coq-verified esterel semantics and compiling. arXiv:1909.12582, 2019
40. Hagen G, Tinelli C. Scaling up the formal verification of Lustre programs with SMT-based techniques. In: Formal Methods in Computer Aided Design, FMCAD. 2008, 1 – 9
41. Ngo V C, Talpin J, Gautier T. Precise deadlock detection for polychronous data-flow specifications. In: Proceedings of the 2014 Electronic System Level Synthesis Conference (ESLsyn). 2014, 1 – 6
42. Talpin J P, Jouvelot P, Shukla S K. Towards refinement types for time-dependent data-flow networks. In: ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE. 2015, 36 – 41
43. Florence S P, You S H, Tov J A, Findler R B. A calculus for Esterel: If can, can. if no can, no can. In: Proceedings of the ACM on Programming Languages, POPL, Vol. 3. 2019, 61:1 – 61:29
44. Song Y, Chin W N. A synchronous effects logic for temporal verification of pure Esterel. In: Verification, Model Checking, and Abstract Interpretation, VMCAI. 2021, 417 – 440
45. Gesell M, Schneider K. A Hoare calculus for the verification of synchronous languages. In: Proceedings of the Sixth Workshop on Programming Languages Meets Program Verification, PLPV. 2012, 37 – 48
46. Harel D, Kozen D, Parikh R. Process logic: Expressiveness, decidability, completeness. Journal of Computer and System Sciences, 1982, 25(2): 144 – 170
47. Zhang Y, Wu H, Chen Y, Mallet F. A clock-based dynamic logic for the verification of CCSL specifications in synchronous systems. Science of Computer Programming, 2021, 203: 102591

9 Appendixes

9.1 Proofs for some Propositions and the Soundness of SDL

Proof of Prop. 3.3. We prove by induction on the syntactic structure of an SP p.

The base cases are trivial. We need to consider 3 basic cases: when p is $\mathbf{0}$, $\mathbf{1}$ or α. We only take α as an example, the cases for $\mathbf{0}$ and $\mathbf{1}$ are similar. In fact, we immediately obtain the result since $\tau(\alpha) = \{\alpha\}$.

If $p = p_1 ; p_2$, by induction hypothesis, we have $val(p) = val(p_1) \circ val(p_2) = \bigcup_{r\in\tau(p_1)} val(r) \circ \bigcup_{r\in\tau(p_2)} val(r) = \bigcup_{r_1\in\tau(p_1), r_2\in\tau(p_2)} val(r_1) \circ val(r_2)$. According to the definition of the operator $\trianglelefteq$ (Def. 3.14), it is easy to see that for any trecs q_1, q_2, $val(q_1 \trianglelefteq q_2) = val(q_1 ; q_2) = val(q_1) \circ val(q_2)$ holds. This is because $\trianglelefteq$ just concatenate two trecs with the operator ; in a special way which does not affect the semantics of the composed program, as analyzed in Sec-

tion 1. Therefore, we have $\bigcup_{r_1\in\tau(p_1),r_2\in\tau(p_2)} val(r_1)\circ val(r_2) = \bigcup_{r_1\in\tau(p_1),r_2\in\tau(p_2)} val(r_1 \trianglelefteq r_2) = \bigcup_{r\in\tau(p_1;p_2)} val(r)$.

If $p = p_1 \cup p_2$, by induction hypothesis, we immediately get $val(p) = val(p_1)\cup val(p_2) = \bigcup_{r\in\tau(p_1)} val(r) \cup \bigcup_{r\in\tau(p_2)} val(r) = \bigcup_{r\in\tau(p_1)\cup\tau(p_2)} val(r) = \bigcup_{r\in\tau(p_1\cup p_2)} val(r)$.

If $p = q^*$, by induction hypothesis, we have $val(q) = \bigcup_{r\in\tau(q)} val(r)$. So $val^n(q) = \underbrace{\bigcup_{r\in\tau(q)} val(r)\circ ...\circ \bigcup_{r\in\tau(q)} val(r)}_{n} = \bigcup_{r\in\tau(q)} val^n(r) = \bigcup_{r\in\tau(q)} val(\underbrace{r\trianglelefteq ...\trianglelefteq r}_{n}) = \bigcup_{r\in\tau(q^n)} val(r)$.

Hence, we have $val(p) = \bigcup_{n=0}^{\infty} val^n(q) = \bigcup_{n=0}^{\infty}\bigcup_{r\in\tau(q^n)} val(r) = \bigcup_{r\in\tau(q^*)} val(r)$.

If $p = \cap(q_1, ..., q_n)$, we immediately obtain the result by Def. 3.16. □

Proposition 9.1. *The following two propositions hold:*

1. $\dfrac{\Gamma\Rightarrow\psi_1,\Delta \quad ... \quad \Gamma\Rightarrow\psi_n,\Delta}{\Gamma\Rightarrow\phi,\Delta}$ $^{(seq1)}$ *is sound for all contexts* Γ *and* Δ *iff the formula* $\bigwedge_{i=1}^n\psi_i\to\phi$ *is valid.*
2. $\dfrac{\Gamma,\psi\Rightarrow\Delta}{\Gamma,\phi_1,...,\phi_n\Rightarrow\Delta}$ $^{(seq2)}$ *is sound for all contexts* Γ *and* Δ *iff the formula* $\bigwedge_{i=1}^n\phi_i\to\psi$ *is valid.*

Proof. We only prove the proposition *1*. The proposition *2* follows a similar idea.

1: For the direction $\to$, if rule (*seq1*) is sound, then we know that if $\bigwedge_{\varphi\in\Gamma}\varphi\to\psi_1\vee\bigvee_{\varphi\in\Delta}\varphi$,..., $\bigwedge_{\varphi\in\Gamma}\varphi\to\psi_n\vee\bigvee_{\varphi\in\Delta}\varphi$ are valid, then $\bigwedge_{\varphi\in\Gamma}\varphi\to\phi\vee\bigvee_{\varphi\in\Delta}\varphi$ is valid. Now we show that for any $s\in\mathbf{S}$, if $s\models\bigwedge_{i=1}^n\psi_i$ then $s\models\phi$. Let $\Gamma = \{\psi_1,...,\psi_n\}$, $\Delta = \{ff\}$, it is easy to see that since $(\psi_1\wedge...\wedge\psi_n)\to(\psi_i\vee ff)$ is valid for any $1\le i\le n$, from the soundness of rule (*seq1*), $(\psi_1\wedge...\wedge\psi_n)\to(\phi\vee ff)$ is valid, which means that for any $u\in\mathbf{S}$, if $u\models\psi_1\wedge...\wedge\psi_n$, then $u\models\phi$. Therefore we immediately have $s\models\phi$.

For the other direction $\leftarrow$, if the formula $\bigwedge_{i=1}^n\psi\to\phi$ is valid, we now show that if $\bigwedge_{\varphi\in\Gamma}\varphi\to\psi_1\vee\bigvee_{\varphi\in\Delta}\varphi$,..., $\bigwedge_{\varphi\in\Gamma}\varphi\to\psi_n\vee\bigvee_{\varphi\in\Delta}\varphi$ are valid, then $\bigwedge_{\varphi\in\Gamma}\varphi\to\phi\vee\bigvee_{\varphi\in\Delta}\varphi$ is valid. For any $s\in\mathbf{S}$, the situation when $s\not\models\bigwedge_{\varphi\in\Gamma}\varphi$ or when $s\models\bigvee_{\varphi\in\Delta}\varphi$ is trivial. So we assume that $s\models\bigwedge_{\varphi\in\Gamma}\varphi$ and $s\not\models\bigvee_{\varphi\in\Delta}\varphi$. Hence we have $s\models\psi_1$,..., $s\models\psi_n$. So $s\models\bigwedge_{i=1}^n\psi_i$. Since $\bigwedge_{i=1}^n\psi_i\to\phi$ is valid, $s\models\phi$ holds. So $s\models\bigwedge_{\varphi\in\Gamma}\varphi\to\phi\vee\bigvee_{\varphi\in\Delta}\varphi$. Since s is an arbitrary state, $s\models\bigwedge_{\varphi\in\Gamma}\varphi\to\phi\vee\bigvee_{\varphi\in\Delta}\varphi$ is valid.

2: The direction $\to$ follows a similar idea as the proof above by setting $\Gamma = \{tt\}$ and $\Delta = \{\psi\}$. The other direction $\leftarrow$ is also similar. □

Proposition 9.2 (Soundness of the Rules of Type (a))**.** *The rules* $(\alpha,\Box\phi),(\psi?),(x := e),(\epsilon),(\mathbf{1})$ *and* $(\mathbf{0})$ *of Table 1 are sound.*

Proof. By Prop. 9.1, for a rule $\dfrac{\psi}{\phi}$, it is sufficient to prove that $\phi\leftrightarrow\psi$ is valid.

In the following proofs for different rules, let s be any state of $\mathbf{S}$.

For rule $(\alpha,\Box\phi)$, $s\models[\alpha]\Box\phi$ iff $ss'\in val_\pi(\Box\phi)$ for all traces $ss'\in val(\alpha)$, iff $s\in val(\phi)$ and $s'\in val(\phi)$ for all traces $ss'\in val(\alpha)$, iff $s\models\phi$ and $s'\in val(\phi)$ for all traces $ss'\in val(\alpha)$, iff $s\models\phi$ and $s\models[\alpha]\phi$, iff $s\models\phi\wedge[\alpha]\phi$.

For rule $(\psi?)$, $s\models[\psi?\,.\,\alpha]\phi$ iff $s'\in val(\phi)$ for all traces $ss'\in val(\psi?\,.\,\alpha)$, iff $s'\in val(\phi)$ for all traces $ss'\in val_m(\psi?\,.\,\alpha)$, iff $s'\in val(\phi)$ for all traces $ss' = ss\circ ss'$ with $s\in val(\psi)$ and $ss'\in val(\alpha)$, iff if $s\in val(\psi)$, then $s'\in val(\phi)$ for all traces $ss' = ss\circ ss'$ with $ss'\in val(\alpha)$, iff if $s\in val(\psi)$, then $s'\in val(\phi)$ for all traces $ss'\in val(\alpha)$, iff $s\models\psi\to[\alpha]\phi$.

For rule $(x := e)$, $s\models[x := e\,.\,\alpha]\phi$ iff $s'\in val(\phi)$ for all traces $ss'\in val(x := e\,.\,\alpha)$, iff $s'\in val(\phi)$ for all traces $ss'\in val_m(x := e\,.\,\alpha)$, iff $s'\in val(\phi)$ for all traces $ss' = ss''\circ s''s'$ with $s'' = s[x\mapsto val_s(e)]$ and $s''s'\in val_m(\alpha)$, iff $s'\in val(\phi)$ for all traces $s''s'\in val(\alpha)$ with $s'' = s[x\mapsto val_s(e)]$, iff $s''\models[\alpha]\phi$ and $s'' = s[x\mapsto val_s(e)]$, iff $s\models([\alpha]\phi)[e/x]$.

For rule (ϵ), $s\models[\epsilon]\phi$ iff $s'\in val(\phi)$ for all traces $ss'\in val(\epsilon)$, iff $s\in val(\phi)$ for the trace $ss\in val_m(\epsilon) = \{tt\mid t\in\mathbf{S}\}$, iff $s\in val(\phi)$, iff $s\models\phi$.

For rule $(\mathbf{1})$, $s\models[\mathbf{1}]\phi$ (resp. $s\models[\mathbf{1}]\Box\phi$) iff $s'\in val(\phi)$ (resp. $ss'\in val_\pi(\Box\phi)$) for all traces $ss'\in val(\mathbf{1})$, iff $s\in val(\phi)$ for the trace (of length 1) $s\in val(\mathbf{1})$, iff $s\in val(\phi)$, iff $s\models\phi$.

For rule $(\mathbf{0})$, $s\models[\mathbf{0}]\phi$ (resp. $s\models[\mathbf{0}]\Box\phi$) iff $s'\in val(\phi)$ (resp. $ss'\in val_\pi(\Box\phi)$) for all traces $ss'\in val(\mathbf{0})$, iff tt since there is no traces in $val(\mathbf{0})$.

□

Proposition 9.3 (Soundness of the Rewrite Rules of Type (b))**.** *The rewrite rules* $(\mathbf{1},;)$, $(\mathbf{1},*)$, $(\mathbf{0},;)$, $(\mathbf{0},*)$, $(;,ass)$, $(;,dis1)$, $(;,dis2)$, $(\cup,ass)$ *and* $(*,exp)$ *of Table 2 are sound.*

Proof. We only show the soundness of the rules $(;,ass)$, $(;,dis1)$ and $(*,exp)$, the soundness of other rules is either trivial or can be proved in a similar way. We omit them here.

For rule $(;,ass)$, we have that for any programs p,q and r, $val((p;q);r) = val(p;q)\circ val(r) = (val(p)\circ val(q))\circ val(r) = val(p)\circ(val(q)\circ val(r)) = val(p)\circ val(q;r) = val(p;(q;r))$.

For rule $(;,dis1)$, we have that for any programs p,q and r, $val(p;(q\cup r)) = val(p)\circ(val(q\cup r)) = val(p)\circ(val(q)\cup val(r)) = (val(p)\circ val(q))\cup(val(p)\circ val(r)) = val(p;q)\cup val(p;r) = val(p;q\cup p;r)$.

For rule $(*,exp)$, we have that for any program p, $val(p^*) = \bigcup_{n=0}^{\infty} val^n(p) = val^0(p)\cup\bigcup_{n=1}^{\infty} val^n(p) = val^0(p)\cup(val(p)\circ(val^0(p)\cup val^1(p)\cup val^2(p)\cup ...)) = val^0(p)\cup(val(p)\circ\bigcup_{n=0}^{\infty} val^n(p)) = val(\mathbf{1})\cup val(p)\circ val(p^*) = val(\mathbf{1}\cup p;p^*)$. □

Proposition 9.4 (Soundness of the Rewrite Rules of Type (c))**.** *The rewrite rules* $(\cap,\mathbf{1})$, $(\cap,\mathbf{0})$, $(\cap,dis)$ *and* $(\cap,mer)$ *of Table 2 are sound.*

Proof. The soundness of the rules $(\cap,\mathbf{1})$, $(\cap,\mathbf{0})$ and $(\cap,mer)$ can be easily proved according to the cases 1, 2 and 4 of Def. 3.21 respectively. Here we only show how to prove the soundness of rule $(\cap,mer)$, the proofs for other two rules are similar.

By Def. 3.12 and 3.16 we know that $val(\cap(\alpha_1\,;q_1,...,\alpha_n\,;q_n)) = \bigcup_{r\in\tau(\cap(\alpha_1\,;q_1,...,\alpha_n\,;q_n))} val_t(r)$, where each trec r of $\cap(\alpha_1\,;q_1,...,\alpha_n\,;q_n)$ must be of the form $\cap(\alpha_1\,;r_1,...,\alpha_n\,;r_n)$ with $r_i \in \tau(q_i)$ for all $1 \le i \le n$. According to the case 4 of Def. 3.21, we have that for each r,

$$val_t(r) = val_t(\cap(\alpha_1\,;r_1,...,\alpha_n\,;r_n)) = val(\alpha')\circ val(\cap(r'_1,...,r'_n))$$

if $b = tt$, where $(b',\alpha',(r'_1\,|...|\,r'_n)) = Mer(\alpha_1\,;r_1\,|...|\,\alpha_n\,;r_n)$. From Def. 3.17 it is easy to see that actually the values of b' and α' are only related to α_1,...,α_n and have nothing to do with r_1,...,r_n, therefore, we have $b' = b$ and $\alpha' = \alpha$ (, where in the rule $(\cap, mer)$ we have $(b,\alpha,(q'_1\,|...|\,q'_n)) = Mer(\alpha_1\,;q_1\,|...|\,\alpha_n\,;q_n))$. So we have $\bigcup_{r\in\tau(\cap(\alpha_1\,;q_1,...,\alpha_n\,;q_n))} val_t(r) = \bigcup_{r\in\tau(\cap(\alpha_1\,;q_1,...,\alpha_n\,;q_n))} val(\alpha)\circ val(\cap(r'_1,...,r'_n)) = val(\alpha)\circ \bigcup_{r'_1\in\tau(q'_1),...,r'_n\in\tau(q'_n)} val(\cap(r'_1,...,r'_n)) = val(\alpha)\circ val(\cap(q'_1,...,q'_n))$.

It remains to show the soundness of rule $(\cap, dis)$. We have that $val(\cap(...,p\cup q,...)) = \bigcup_{r\in\tau(\cap(...,p\cup q,...))} val_t(r) = \bigcup_{r\in\tau(\cap(...,p,...))} val_t(r)\cup\bigcup_{r\in\tau(\cap(...,q,...))} val_t(r) = val(\cap(...,p,...))\cup val(\cap(...,q,...)) = val(\cap(...,p,...)\cup\cap(...,q,...))$.

□

We introduce the level of a rewrite relation between two programs as the following definition.

Definition 9.1 (Level of a Rewrite Relation)**.** *Given two SPs p, q and a rewrite relation $p \rightsquigarrow q$ between them, we define the level of the relation, denoted as $lev(p \rightsquigarrow q)$, as follows:*

1. *$lev(p \rightsquigarrow q) =_{df} 1$ if $p \rightsquigarrow q$ is from one of the rules of the types (b) and (c) in Table 2.*
2. *$lev(p \rightsquigarrow q) =_{df} 1 + lev(p' \rightsquigarrow q')$ if there exist SPs p', q' and a program hole $r\{\_\}$ such that $p = r\{p'\}$, $q = r\{q'\}$ and $p' \rightsquigarrow q'$. In other words, $p \rightsquigarrow q$ is from rule ($r2$).*

Lemma 9.1. *Given a parallel program $\cap(...,p\{\_\},...)$ with a program hole $p\{\_\}$, if $p\{q\} \rightsquigarrow p\{r\}$ with $q \rightsquigarrow r$ and $lev(q \rightsquigarrow r) = 1$ for some programs q and r, then*

$$val(\cap(...,p\{q\},...)) = val(\cap(...,p\{r\},...)). \tag{5}$$

Proof. We prove by analyzing two different cases of $lev(q \rightsquigarrow r) = 1$:

1. If $q \rightsquigarrow r$ is from one of the rules of type (b) and rule $(\cap, dis)$ (of type (c)), then $\tau(p\{q\}) = \tau(p\{r\})$, so the equation (5) holds obviously.
2. If $q \rightsquigarrow r$ is from other rules of type (c) except rule $(\cap, dis)$, then the equation (5) holds.

The case 1 can be proved by induction on the syntactic structure of $p_i\{\_\}$ for different rules. Here we only take rule $(;, dis1)$ as an example, other cases are similar.

Let $q = u_1\,;(u_2 \cup u_3)$ and $r = u_1\,;u_2 \cup u_1\,;u_3$. If $p_i\{\_\} = \_$, we need to prove that $\tau(q) = \tau(r)$, which holds because $\tau(u_1\,;(u_2 \cup u_3)) = \{t_1 \trianglelefteq t_2 \mid t_1 \in \tau(u_1), t_2 \in \tau(u_2 \cup u_3)\} = \{t_1 \trianglelefteq t_2 \mid t_1 \in \tau(u_1), t_2 \in \tau(u_2)\} \cup \{t_1 \trianglelefteq t_2 \mid t_1 \in \tau(u_1), t_2 \in \tau(u_3)\} = \tau(u_1\,;u_2) \cup \tau(u_1\,;u_3) = \tau(u_1\,;u_2 \cup u_1\,;u_3)$.

If $p\{\_\} = p_1\{\_\}\,;p_2$, by induction hypothesis we have $\tau(p_1\{q\}) = \tau(p_1\{r\})$, so $\tau(p\{q\}) = \{t_1 \trianglelefteq t_2 \mid t_1 \in \tau(p_1\{q\}), t_2 \in \tau(p_2)\} = \{t_1 \trianglelefteq t_2 \mid t_1 \in \tau(p_1\{r\}), t_2 \in \tau(p_2)\} = \tau(p\{r\})$.

The case for $p\{\_\} = p_1\,;p_2\{\_\}$ is similar.

If $p\{\_\} = p_1\{\_\} \cup p_2$, by induction hypothesis we have $\tau(p_1\{q\}) = \tau(p_1\{r\})$, so $\tau(p\{q\}) = \{t \mid t \in \tau(p_1\{q\})\} \cup \{t \mid t \in \tau(p_2)\} = \{t \mid t \in \tau(p_1\{r\})\} \cup \{t \mid t \in \tau(p_2)\} = \tau(p\{r\})$.

The case for $p\{\_\} = p_1 \cup p_2\{\_\}$ is similar.

If $p\{\_\} = (p_1\{\_\})^*$, by induction hypothesis we have $\tau(p_1\{q\}) = \tau(p_1\{r\})$, so $\tau(p\{q\}) = \bigcup_{n=0}^{\infty}\tau(p_1^n\{q\}) = \bigcup_{n=0}^{\infty}\{t_1 \trianglelefteq ... \trianglelefteq t_n \mid t_i \in \tau(p_1\{q\}) \text{ for } 1 \le i \le n\} = \bigcup_{n=0}^{\infty}\{t_1 \trianglelefteq ... \trianglelefteq t_n \mid t_i \in \tau(p_1\{r\}) \text{ for } 1 \le i \le n\} = \bigcup_{n=0}^{\infty}\tau(p_1^n\{r\}) = \tau(p\{r\})$.

If $p\{\_\} = \cap(p_1,...,p_i\{\_\},...,p_n)$, by induction hypothesis we have $\tau(p_i\{q\}) = \tau(p_i\{r\})$, so $\tau(p\{q\}) = \{\cap(t_1,...,t_n) \mid t_1 \in \tau(p_1),...,t_i \in \tau(p_i\{q\}),...,t_n \in \tau(p_n)\} = \{\cap(t_1,...,t_n) \mid t_1 \in \tau(p_1),...,t_i \in \tau(p_i\{r\}),...,t_n \in \tau(p_n)\} = \tau(p\{r\})$.

To prove the case 2, we analyze the rules $(\cap, \mathbf{1})$, $(\cap, \mathbf{0})$, $(\cap, mer)$ and rule $(\cap, seq)$ separately.

The cases for the rules $(\cap, \mathbf{1})$, $(\cap, \mathbf{0})$ and $(\cap, mer)$ can be proved according to the cases 1, 2 and 4 of Def. 3.21 respectively. Here we only consider the proof for rule $(\cap, mer)$ as an example, other cases for the rules $(\cap, \mathbf{1})$ and $(\cap, \mathbf{0})$ are similar.

If $q \rightsquigarrow r$ is from rule $(\cap, mer)$, let $q = \cap(\alpha_1\,;q_1,...,\alpha_n\,;q_n)$ and $r = \alpha\,;\cap(q'_1,...,q'_n)$ where $(b,\alpha,(q'_1\,|...|\,q'_n)) = Mer(\alpha_1\,;q_1\,|...|\,\alpha_n\,;q_n)$ and $b = tt$. In the sets $\tau(p\{q\})$ and $\tau(p\{r\})$, it is easy to see that each trec r either has no holes and $r \in \tau(p\{q\}) \cap \tau(p\{r\})$, or has a hole in the same position of the hole $p\{\_\}$ and there exist two trecs $r' = \cap(\alpha_1\,;u_1,...,\alpha_n\,;u_n) \in \tau(q)$ and $r'' = \alpha\,;\cap(u'_1,...,u'_n) \in \tau(r)$ such that $r\{r'\} \in \tau(p\{q\})$, $r\{r''\} \in \tau(p\{r\})$ and $(b,\alpha,(u'_1\,|...|\,u'_n)) = Mer(\alpha_1\,;u_1\,|...|\,\alpha_n\,;u_n)$ since the return values b and α only depend on the events α_1,...,α_n. Let $r_1 = \cap(...,r\{r'\},...)$ and $r_2 = \cap(...,r\{r''\},...)$ be two trecs of $\cap(...,p\{q\},...)$ and $\cap(...,p\{r\},...)$ respectively, it remains to show that $val_t(r_1) = val_t(r_2)$. However, this is trivial according to the case 4 of Def. 3.21. Therefore, we have the equation (5) holds for rule $(\cap, mer)$.

□

We show that Lemma 9.1 can be extended to the case when $lev(q \rightsquigarrow r)$ is an arbitrary number.

Lemma 9.2. *The equation (5) of Lemma. 9.1 holds for any $lev(q \rightsquigarrow r)$.*

Proof. We prove by induction on $lev(q \rightsquigarrow r)$, where Lemma. 9.1 has shown the base case. Now for any $n > 1$, we assume that the equation (5) holds for $lev(q \rightsquigarrow r) < n$, next we prove that the equation (5) holds for $lev(q \rightsquigarrow r) = n$.

Since $lev(q \rightsquigarrow r) > 1$, we know that there exist a program u and two programs u_1, r_1 such that $q = u\{q_1\}$, $r = u\{r_1\}$ and $q_1 \rightsquigarrow r_1$. Let $p\{q_1\}_1 = p\{u\{q_1\}\}$ and $p\{r_1\}_1 = p\{u\{r_1\}\}$ (where we use $p\{\_\}_1$ to distinguish itself from $p\{\_\}$), since $lev(q_1 \rightsquigarrow r_1) < n$, by induction hypothesis, we immediately have $val(\cap(...,p\{q_1\}_1,...)) = val(\cap(...,p\{r_1\}_1,...))$. So $val(\cap(...,p\{q\},...)) = val(\cap(...,p\{r\},...))$ holds.

□

Proposition 9.5 (Soundness of Rule ($r2$) on level 1)**.** *Given a closed SP p and a program hole $p\{\_\}$ of p, for any SPs q and r that satisfy $q \rightsquigarrow r$ and $lev(q \rightsquigarrow r) = 1$, we have*

$$val(p\{q\}) = val(p\{r\}). \tag{6}$$

Proof. We proceed the proof by induction on the syntactic structure of p.

If $p\{\_\} = \_$, i.e., $p\{\_\}$ is just a hole, then we need to prove $val(q) = val(r)$ for the closed programs q and r, which are the direct results of Prop. 9.3, Prop. 9.4 and Lemma 9.1 for all the cases when $lev(q \rightsquigarrow r) = 1$.

If $p\{\_\} = p_1\{\_\}; p_2$, by induction hypothesis we have that $val(p_1\{q\}) = val(p_1\{r\})$, hence $val(p\{q\}) = val(p_1\{q\}) \circ val(p_2) = val(p_1\{r\}) \circ val(p_2) = val(p\{r\})$.

The case for $p\{\_\} = p_1 ; p_2\{\_\}$ is similar.

If $p\{\_\} = p_1\{\_\} \cup p_2$, by induction hypothesis we have that $val(p_1\{q\}) = val(p_1\{r\})$, hence $val(p\{q\}) = val(p_1\{q\}) \cup val(p_2) = val(p_1\{r\}) \cup val(p_2) = val(p\{r\})$.

The case for $p\{\_\} = p_1 \cup p_2\{\_\}$ is similar.

If $p\{\_\} = (p_1\{\_\})^*$, by induction hypothesis we have that $val(p_1\{q\}) = val(p_1\{r\})$, so $val(p\{q\}) = \bigcup_{i=0}^{\infty} val^n(p_1\{q\}) = \bigcup_{i=0}^{\infty} val^n(p_1\{r\}) = val(p\{r\})$.

If $p\{\_\} = \cap(p_1, ..., p_i\{\_\}, ..., p_n)$, we directly obtain the result by Lemma 9.1. □

With Prop. 9.5, now we consider the soundness of rule ($r2$) on an arbitrary level.

Proposition 9.6 (Soundness of Rule ($r2$))**.** *The equation (6) of Prop. 9.5 holds for any $lev(q \rightsquigarrow r)$.*

Proof. We prove by induction on $lev(q \rightsquigarrow r)$. By Prop. 9.5 we obtain the base case. Now for any $n > 1$, we assume that the equation (6) holds for $lev(q \rightsquigarrow r) < n$, next we show that the equation (6) holds for $lev(q \rightsquigarrow r) = n$.

We prove the proposition by induction on the structure of $p\{\_\}$.

If $p\{\_\} = \_$, i.e., $p\{\_\}$ is just a hole, then we need to prove $val(q) = val(r)$ for the closed programs q and r. Since $lev(q \rightsquigarrow r) > 1$, there exist a program u and two programs q_1, r_1 such that $q = u\{q_1\}$, $r = u\{r_1\}$ and $q_1 \rightsquigarrow r_1$. Because $lev(q_1 \rightsquigarrow r_1) < n$, by induction hypothesis we immediately obtain that $val(q) = val(r)$.

The proofs for the inductive cases of $p\{\_\} = p_1\{\_\}; p_2$, $p\{\_\} = p_1 ; p_2\{\_\}$, $p\{\_\} = p_1\{\_\} \cup p_2$, $p\{\_\} = p_1 \cup p_2\{\_\}$ and $p\{\_\} = (p_1\{\_\})^*$ are similar to those in Prop. 9.5 and we omit them here.

The case when $p\{\_\} = \cap(p_1, ..., p_i\{\_\}, ..., p_n)$ is straightforward by Lemma 9.2.

□

Proposition 9.7 (Soundness of Rule ($\cap, seq$))**.** *Given a well-defined parallel program $\cap(p_1, ..., p_n)$, $val(\cap(p_1, ..., p_n)) = val(Brz(\cap(p_1, ..., p_n)))$.*

Proof. The soundness of rule ($\cap, seq$) is directly from Arden's rule (Prop. 4.1) and the soundness of rule ($r2$) and other rewrite rules of the types (b) and (c). □

Proposition 9.8 (Soundness of Rule ($r1$))**.** *The rewrite rule ($r1$) is sound.*

Proof. Since the programs p and q in rule ($r1$) are closed programs, it is enough to prove that if $val(p) = val(q)$, then $val(\phi\{p\}) = val(\phi\{q\})$.

We proceed by induction on the syntactic structure of $\phi\{\_\}$, where the base cases are $\phi\{\_\} = [r\{\_\}]\psi$ and $\phi\{\_\} = [r\{\_\}]\Box\phi$.

For the base cases, we consider $\phi\{\_\} = [r\{\_\}]\psi$ for example, the case of $\phi\{\_\} = [r\{\_\}]\Box\psi$ is similar. If $\phi\{\_\} = [r\{\_\}]\psi$, by the soundness of rule ($r2$) (Prop. 9.6) and the rules of the types (b) and (c) (Prop. 9.3, 9.4 and 9.7), we have $val(r\{p\}) = val(r\{q\})$. By Def. 3.22, we immediately obtain the result.

The cases for $\phi\{\_\} = \neg\psi\{\_\}$, $\phi\{\_\} = \phi_1\{\_\} \wedge \phi_2$, $\phi\{\_\} = \phi_1 \wedge \phi_2\{\_\}$ and $\forall x.\psi\{\_\}$ are similar, we only take $\phi\{\_\} = \neg\psi\{\_\}$ as an example. If $\phi\{\_\} = \neg\psi\{\_\}$, by induction hypothesis we have $val(\psi\{p\}) = val(\psi\{q\})$, then the result is straightforward by Def. 3.22. □

9.2 Proofs for the Relative Completeness of SDL

In the proofs below, we often denote an AFOL formula ϕ as $\phi^\flat$ to distinguish it from an SDL formula.

The proof of Theorem 5.3 mainly follows that of the "main theorem" in [25], but is augmented to fit the condition (iv).

The proof of Theorem 5.3. For an SDL formula ϕ, we can convert ϕ into a semantical equivalent 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 SDL formula, or its negation. By the FOL rules, it is sufficient to prove that for each clause C_i, $\models C_i$ implies $\vdash^+ C_i$. 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 dynamic formulas, 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, thus we immediately obtain $\vdash^+ C_i$.

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

$$C_i = \phi_1 \vee op\,\phi_2, C_i = \phi_1 \vee [p]\Box\phi_2, C_i = \phi_1 \vee \langle p\rangle\Diamond\phi_2$$

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

If $C_i = \phi_1 \vee op\,\phi_2$, which is equivalent to $\neg\phi_1 \rightarrow op\,\phi_2$, by the condition (i), there exist two AFOL formulas $\psi_1^\flat$ and $\psi_2^\flat$ such that $\models \psi_1^\flat \leftrightarrow \neg\phi_1$ and $\models \psi_2^\flat \leftrightarrow \phi_2$. Then we have $\models \psi_1^\flat \rightarrow op\,\psi_2^\flat$ holds. By the condition (ii) we have

$$\vdash^+ \psi_1^\flat \rightarrow op\,\psi_2^\flat. \tag{7}$$

Since in $\psi_1^\flat \leftrightarrow \neg\phi_1$ and $\psi_2^\flat \leftrightarrow \phi_2$ the sum is strictly less than n, by inductive hypothesis we can get that

$$\vdash^+ \neg\phi_1 \rightarrow \psi_1^\flat \tag{8}$$

and $\vdash^+ \psi_2^\flat \to \phi_2$ hold. By the condition (iii) we know that

$$\vdash^+ op\, \psi_2^\flat \to op\, \phi_2 \tag{9}$$

holds. Based on (7), (8), (9) and the FOL rules in Table 3 and 4 we can conclude that $\vdash^+ \neg\phi_1 \to op\, \phi_2$.

If $C_i = \phi_1 \vee [p]\Box\phi_2$, which is equivalent to $\neg\phi_1 \to [p]\Box\phi_2$, now we prove that $\vdash^+ \neg\phi_1 \to [p]\Box\phi_2$. By the condition (i) there exists an AFOL formula $\psi_1^\flat$ such that $\models \psi_1^\flat \leftrightarrow \neg\phi_1$. Note that in $\psi_1^\flat \leftrightarrow \neg\phi_1$ the sum is strictly less than n, by inductive hypothesis we have $\vdash^+ \neg\phi_1 \to \psi_1^\flat$. On the other hand, by the condition (iv), $\models \neg\phi_1 \to [p]\Box\phi_2$ and $\models \psi_1^\flat \leftrightarrow \neg\phi_1$, we have that $\vdash^+ \psi_1^\flat \to [p]\Box\phi_2$ holds. Thus we have $\vdash^+ \neg\phi_1 \to [p]\Box\phi_2$.

Similar for the case $C_i = \phi_1 \vee \langle p\rangle\Diamond\phi_2$.

□

The proof of Theorem 5.3(i). It is known that an FODL formula of the form $[p]\phi^\flat$ can be expressed as an AFOL formula (refer to page 326 of [24]). In SDL, it is easy to see that closed sequential SPs have the same expressiveness as the regular programs in FODL. Because a closed sequential SP can be taken as a regular program by simply ignoring the differences between macro and micro events in this SP which only play their roles in parallel SPs. For example, an SP $p = (x := 1\,.\,x >= 1?\,.\,\epsilon\,; y := x + 2)$ can be taken as a regular program $p' = (x := 1\,; x >= 1?\,; tt?\,; y := x + 2)$ by replacing all operators . with the operator ; and the event ϵ with $tt?$. p and p' have the same semantics.

Therefore,

any SDL formula of the form $[p]\phi^\flat$, where p is a sequential program, can be expressed as an AFOL formula. (10)

Based on this fact, we now show that any SDL formula is expressible in AFOL. Actually, we only need to consider the case when all programs of an SDL formula are sequential, because a parallel SP is semantically equivalent to a sequential one according to the procedure *Brz*.

Given an SDL formula ϕ, in which all programs are sequential, we prove by induction on the number of the appearances of $[p]$ in ϕ. The base case is trivial. For the inductive cases, the only non-trivial cases are $\phi = [p]\psi$ and $\phi = [p]\Box\psi$, where by inductive hypothesis there exists an AFOL formula such that $\models \psi \leftrightarrow \psi_1^\flat$. From the fact (10) above, we already prove the case $[p]\psi_1^\flat$. It remains to show that $[p]\Box\psi_1^\flat$ is expressible according to different cases of p. Below we prove $[p]\Box\psi_1^\flat$ by induction on the structure of p.

- The base cases are $p = \mathbf{1}$, $p = \mathbf{0}$ and $p = \alpha$. We only take $p = \alpha$ for example, other cases are similar. By the soundness of rule $(\alpha, \Box\phi)$, we obtain that $\models [\alpha]\Box\psi_1^\flat \leftrightarrow (\psi_1^\flat \wedge [\alpha]\psi_1^\flat)$, where by (10) $[\alpha]\psi_1^\flat$ is expressible in AFOL. So $\psi_1^\flat \wedge [\alpha]\psi_1^\flat$ is also expressible in AFOL.
- If $p = p_1\,; p_2$, by the soundness of rule $(;, \Box\phi)$ and Prop. 5.1, we have that $\models [p_1\,; p_2]\Box\psi_1^\flat \leftrightarrow ([p_1]\Box\psi_1^\flat \wedge [p_1][p_2]\Box\psi_1^\flat)$. By inductive hypothesis, we have $[p_1]\Box\psi_1^\flat$ and $[p_2]\Box\psi_1^\flat$ are expressible in AFOL. So by (10), $[p_1][p_2]\Box\psi_1^\flat$ is also expressible in AFOL. So $[p_1]\Box\psi_1^\flat \wedge [p_1][p_2]\Box\psi_1^\flat$ is expressible in AFOL.
- If $p = p_1 \cup p_2$, by the soundness of rule $(\cup)$, we have that $\models [p_1 \cup p_2]\Box\psi_1^\flat \leftrightarrow ([p_1]\Box\psi_1^\flat \wedge [p_2]\Box\psi_2^\flat)$. By inductive hypothesis, $[p_1]\Box\psi_1^\flat$ and $[p_2]\Box\psi_1^\flat$ are expressible in AFOL, so is $[p_1]\Box\psi_1^\flat \wedge [p_2]\Box\psi_2^\flat$.
- If $p = q^*$, by the soundness of rule $(*, \Box\phi)$, we have that $\models [q^*]\Box\psi_1^\flat \leftrightarrow [q^*][q]\Box\psi_1^\flat$. By inductive hypothesis, $[q]\Box\psi^\flat$ is expressible in AFOL. By (10) we then get that $[q^*][q]\Box\psi^\flat$ is expressible in AFOL.

□

The proof of Theorem 5.3(ii). We prove by induction on the syntactic structure of p. In $\phi^\flat \to op\,\psi^\flat$, when op is $\forall x$ or $\exists x$, the proof is trivial, because $\phi^\flat \to op\,\phi^\flat$ itself is an AFOL formula so there must be $\vdash^+ \phi^\flat \to op\,\psi^\flat$.

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

- The base cases are $p = \mathbf{1}$, $p = \mathbf{0}$ and $p = \alpha$, we only take $p = \alpha$ for example, other cases are similar. We prove by induction on the number k of micro events in α. If $k = 1$, then $\alpha = \epsilon$. From $\models \phi^\flat \to [\epsilon]\psi^\flat$, by the soundness of rule (ϵ), we have $\models \phi^\flat \to \psi^\flat$. Since $\phi^\flat \to \psi^\flat$ is an AFOL formula, $\vdash^+ \phi^\flat \to \psi^\flat$ holds. By rule (ϵ) and the FOL rules in Table 3 and 4, we obtain that $\vdash^+ \phi^\flat \to [\epsilon]\psi^\flat$. Suppose $k > 1$, α is either of the form $\psi_0?\,.\beta$ or $x := e\,.\beta$. We only consider the case $\alpha = (x := e\,.\beta)$, the other case is similar. From $\models \phi^\flat \to [x := e\,.\beta]\psi^\flat$, by the soundness of rule $(x := e)$ we have that $\models \phi^\flat \to ([\beta]\psi^\flat)[e/x]$. By inductive hypothesis, we know that $\vdash^+ \phi^\flat \to ([\beta]\psi^\flat)[e/x]$, therefore by rule $(x := e)$ and the FOL rules we can derive $\vdash^+ \phi^\flat \to [x := e\,.\beta]\psi^\flat$.
- If $p = p_1\,; p_2$, from $\models \phi^\flat \to [p_1; p_2]\psi^\flat$, by the soundness of rule $(;, \phi)$, we obtain $\models \phi^\flat \to [p_1][p_2]\psi^\flat$. By the condition (i), there exists an AFOL formula $\psi_1^\flat$ such that $\models [p_2]\psi^\flat \leftrightarrow \psi_1^\flat$. Hence $\models \psi_1^\flat \to [p_2]\psi^\flat$ and $\models \phi^\flat \to [p_1]\psi_1^\flat$. By inductive hypothesis, we have $\vdash^+ \psi_1^\flat \to [p_2]\psi^\flat$ and $\vdash^+ \phi^\flat \to [p_1]\psi_1^\flat$. Applying rule $([], gen)$ on $\vdash^+ \psi_1^\flat \to [p_2]\psi^\flat$, we obtain $\vdash^+ [p_1]\psi_1^\flat \to [p_1][p_2]\psi^\flat$. From $\vdash^+ \phi^\flat \to [p_1]\psi_1^\flat$ and $\vdash^+ [p_1]\psi_1^\flat \to [p_1][p_2]\psi^\flat$, by applying rule $(;, \phi)$ and the FOL rules in Table 3 and 4, we can derive $\vdash^+ \phi^\flat \to [p_1\,; p_2]\psi^\flat$.
- If $p = p_1 \cup p_2$, from $\models \phi^\flat \to [p_1 \cup p_2]\psi^\flat$, by the soundness of rule $(\cup)$, there is $\models \phi^\flat \to ([p_1]\psi^\flat \wedge [p_2]\psi^\flat)$, which is equivalent to $\models \phi^\flat \to [p_1]\psi^\flat$ and $\models \phi^\flat \to [p_2]\psi^\flat$. By inductive hypothesis we have that $\vdash^+ \phi^\flat \to [p_1]\psi^\flat$ and $\vdash^+ \phi^\flat \to [p_2]\psi^\flat$. By rule $(\cup)$ and the FOL rules in Table 3 and 4, we have $\vdash^+ \phi^\flat \to [p_1 \cup p_2]\psi^\flat$.
- If $p = q^*$, by the condition (i), there exists an AFOL formula $\phi_1^\flat$ such that $\models [q^*]\psi^\flat \leftrightarrow \phi_1^\flat$. From $\models \phi^\flat \to [q^*]\psi^\flat$, we also have $\models \phi^\flat \to \phi_1^\flat$. From $\models [q^*]\psi^\flat \leftrightarrow \phi_1^\flat$,

by the soundness of the rules $(*)$, $(\cup)$, $(\mathbf{1})$ and $(;,\phi)$, it is not hard to see that $\models \phi_1^\flat \leftrightarrow [q^*]\psi^\flat \leftrightarrow [\mathbf{1} \cup q\,;q^*]\psi^\flat \leftrightarrow [\mathbf{1}]\psi^\flat \wedge [q\,;q^*]\psi^\flat \leftrightarrow \psi^\flat \wedge [q][q^*]\psi^\flat \leftrightarrow \psi^\flat \wedge [q]\phi_1^\flat$. From these logical equivalences we can see that $\models \phi_1^\flat \rightarrow [q]\phi_1^\flat$ and $\models \phi_1^\flat \rightarrow \psi^\flat$. By inductive hypothesis, from $\models \phi^\flat \rightarrow \phi_1^\flat$, $\models \phi_1^\flat \rightarrow [q]\phi_1^\flat$ and $\models \phi_1^\flat \rightarrow \psi^\flat$, we get that $\vdash^+\phi^\flat \rightarrow \phi_1^\flat$, $\vdash^+\phi_1^\flat \rightarrow [q]\phi_1^\flat$ and $\vdash^+\phi_1^\flat \rightarrow \psi^\flat$. By rule $([*])$ and the FOL rules, finally there is $\vdash^+\phi^\flat \rightarrow [q^*]\psi^\flat$.

- If $p = \cap(q_1, ..., q_n)$, by applying rule $(\cap, seq)$, we can transform p into a sequential program p', i.e., $p \rightsquigarrow p'$. By the soundness of rule $(r1)$, from $\models \phi^\flat \rightarrow [p]\psi^\flat$, we can get that $\models \phi^\flat \rightarrow [p']\psi^\flat$. Since p' is sequential, we can analyze it based on the cases given above. Using inductive hypothesis, we can prove that $\vdash^+\phi^\flat \rightarrow [p']\psi^\flat$. By rule $(r1)$ we can obtain $\vdash^+\phi^\flat \rightarrow [p]\psi^\flat$.

For the case when op is $\langle p\rangle$, the proofs for the cases $p = \mathbf{1}$, $p = \mathbf{0}$, $p = \alpha$, $p = p_1\,;p_2$, $p = p_1 \cup p_2$ and $p = \cap(q_1, ..., q_n)$ are similar to the proofs above, since $\langle p\rangle\psi^\flat$ equals to $\neg[p]\neg\psi^\flat$ and the rules $(\epsilon), (x := e), (;,\phi), (\cup), (*), (\cup), (\mathbf{1})$ used in the proofs above are bidirectional. (For rule $([], gen)$, we have the rule $(\langle\rangle, gen)$.) In the following we only prove the case $p = q^*$.

- If $p = q^*$, by the condition (i) and the way of expressing regular programs in AFOL in [25], we know that for any n, $\langle q^n\rangle\psi^\flat$ can be expressed as an AFOL formula $\phi_1^\flat(n)$, i.e., $\models \langle q^n\rangle\psi^\flat \leftrightarrow \phi_1^\flat(n)$. From the semantics of $\langle q^*\rangle\psi^\flat$, it is easy to see that $\models \langle q^*\rangle\psi^\flat \leftrightarrow \exists n \geq 0.\phi_1^\flat(n)$. From $\models \phi^\flat \rightarrow \langle q^*\rangle\psi^\flat$, there is $\models \phi^\flat \rightarrow \exists n \geq 0.\phi_1^\flat(n)$. On the other hand, when $n > 0$, by the soundness of rule $(;,\phi)$, we have $\models \phi_1^\flat(n) \leftrightarrow \langle q^n\rangle\psi^\flat \leftrightarrow \langle q\,;q^{n-1}\rangle\psi^\flat \leftrightarrow \langle q\rangle\langle q^{n-1}\rangle\psi^\flat \leftrightarrow \langle q\rangle\phi_1^\flat(n-1)$. From these logical equivalences we can get that $\models \phi_1^\flat(n) \rightarrow \langle q\rangle\phi_1^\flat(n-1)$. When $n = 0$, from $\models \langle q^n\rangle\psi^\flat \leftrightarrow \phi_1^\flat(n)$, by the soundness of rule (ϵ), we have $\models \langle q^0\rangle\psi^\flat \leftrightarrow \langle\mathbf{1}\rangle\psi^\flat \leftrightarrow \psi^\flat \leftrightarrow \phi_1^\flat(0)$. So $\models \phi_1^\flat(0) \rightarrow \psi^\flat$. By inductive hypothesis, from $\models \phi^\flat \rightarrow \exists n \geq 0.\phi_1^\flat(n)$, $\models \phi_1^\flat(n) \rightarrow \langle q\rangle\phi_1^\flat(n-1)$ and $\models \phi_1^\flat(0) \rightarrow \psi^\flat$, we have that $\vdash^+\phi^\flat \rightarrow \exists n \geq 0.\phi_1^\flat(n)$, $\vdash^+\phi_1^\flat(n) \rightarrow \langle q\rangle\phi_1^\flat(n-1)$ and $\vdash^+\phi_1^\flat(0) \rightarrow \psi^\flat$. By rule $(\langle*\rangle)$ and the FOL rules in Table 3 and 4, we obtain $\vdash^+\phi^\flat \rightarrow \langle q^*\rangle\psi^\flat$.

□

The proof of Theorem 5.3 (iii). When $op \in \{[p], \langle p\rangle\}$, the condition (iii) is in fact stated as the rules $([], gen)$ and $(\langle\rangle, gen)$. So we only need to prove when $op \in \{\forall x, \exists x\}$. Below we only consider the case when op is $\forall x$. The case when op is $\exists x$ can be similarly obtained by using the dual rules of the rules used in the proof below.

Actually, using the FOL rules in Table 3 and 4, we can construct the following deductions:

$$\dfrac{\dfrac{\dfrac{\dfrac{\cdot \Rightarrow \phi[x'/x] \rightarrow \psi[x'/x]}{\phi[x'/x] \Rightarrow \psi[x'/x]}\,(\rightarrow l)}{\forall x.\phi \Rightarrow \psi[x'/x]}\,(\forall l)}{\forall x.\phi \Rightarrow \forall x.\psi}\,(\forall r)}{\cdot \Rightarrow \forall x.\phi \rightarrow \forall x.\psi}\,(\rightarrow r)$$

where x' is a new variable w.r.t. ϕ and ψ. □

The proof of Theorem 5.3 (iv). As the proof of Theorem 5.3 (ii), we proceed by induction on the syntactic structure of p. Below we only consider the case $\phi^\flat \rightarrow [p]\Box\psi^\flat$. The proof of the case $\phi^\flat \rightarrow \langle p\rangle\Diamond\psi^\flat$ is similar by the relation between $\langle p\rangle\Diamond\psi^\flat$ and its dual form $[p]\Box\psi^\flat$.

The cases for $p = \mathbf{1}$, $p = \mathbf{0}$, $p = p_1 \cup p_2$ and $p = \cap(q_1, ..., q_n)$ are similar to the corresponding cases in the proof of Theorem 5.3 (ii) above. We omit them here.

- For the base case, we only consider $p = \alpha$. From $\models \phi^\flat \rightarrow [\alpha]\Box\psi^\flat$, by the soundness of rule $(\alpha, \Box\phi)$, there is $\models \phi^\flat \rightarrow (\psi^\flat \wedge [\alpha]\psi^\flat)$, which is equivalent to $\models \phi^\flat \rightarrow \psi^\flat$ and $\models \phi^\flat \rightarrow [\alpha]\psi^\flat$. Obviously there is $\vdash^+\phi^\flat \rightarrow \psi^\flat$. By the condition (ii), we have $\vdash^+\phi^\flat \rightarrow [\alpha]\psi^\flat$. Therefore, by the FOL rules in Table 3 and 4 we have $\vdash^+\phi^\flat \rightarrow (\psi^\flat \wedge [\alpha]\psi^\flat)$. By rule $(\alpha, \Box\phi)$ and the rules in FOL, we obtain that $\vdash^+\phi^\flat \rightarrow [\alpha]\Box\psi^\flat$.
- If $p = p_1\,;p_2$, from $\models \phi^\flat \rightarrow [p_1\,;p_2]\Box\psi^\flat$, by the soundness of rule $(;, \Box\phi)$ and Prop. 5.1, there is $\models \phi^\flat \rightarrow ([p_1]\Box\psi^\flat \wedge [p_1][p_2]\Box\psi^\flat)$, which is equivalent to $\models \phi^\flat \rightarrow [p_1]\Box\psi^\flat$ and $\models \phi^\flat \rightarrow [p_1][p_2]\Box\psi^\flat$. From $\models \phi^\flat \rightarrow [p_1]\Box\psi^\flat$, by inductive hypothesis we can have $\vdash^+\phi^\flat \rightarrow [p_1]\Box\psi^\flat$. According to the condition (i), there is an AFOL formula $\psi_1^\flat$ such that $\models [p_2]\Box\psi^\flat \leftrightarrow \psi_1^\flat$. So $\models \phi^\flat \rightarrow [p_1]\psi_1^\flat$ and $\models \psi_1^\flat \rightarrow [p_2]\Box\psi^\flat$. By inductive hypothesis, there are $\vdash^+\phi^\flat \rightarrow [p_1]\psi_1^\flat$ and $\vdash^+\psi_1^\flat \rightarrow [p_2]\Box\psi^\flat$. From $\vdash^+\psi_1^\flat \rightarrow [p_2]\Box\psi^\flat$, by applying rule $([], gen)$, we get that $\vdash^+[p_1]\psi^\flat \rightarrow [p_1][p_2]\Box\psi^\flat$. By $\vdash^+\phi^\flat \rightarrow [p_1]\psi_1^\flat$ and $\vdash^+[p_1]\psi^\flat \rightarrow [p_1][p_2]\Box\psi^\flat$, we conclude that $\vdash^+\phi^\flat \rightarrow [p_1][p_2]\Box\psi^\flat$. From $\vdash^+\phi^\flat \rightarrow [p_1]\Box\psi^\flat$ and $\vdash^+\phi^\flat \rightarrow [p_1][p_2]\Box\psi^\flat$, by applying rule $(;, \Box\phi)$ and other FOL rules, we can derive $\vdash^+\phi^\flat \rightarrow [p_1\,;p_2]\Box\psi^\flat$.
- If $p = q^*$, from $\models \phi^\flat \rightarrow [q^*]\Box\psi^\flat$, by the soundness of rule $(*, \Box\phi)$, we have that $\models \phi^\flat \rightarrow [q^*][q]\Box\psi^\flat$. According to the condition (i), there exists an AFOL formula $\psi_1^\flat$ such that $\models [q]\Box\psi^\flat \leftrightarrow \psi_1^\flat$. Hence there are $\models \phi^\flat \rightarrow [q^*]\psi_1^\flat$ and $\models \psi_1^\flat \rightarrow [q]\Box\psi^\flat$. By the condition (ii), from $\models \phi^\flat \rightarrow [q^*]\psi_1^\flat$ we have $\vdash^+\phi^\flat \rightarrow [q^*]\psi_1^\flat$. From $\models \psi_1^\flat \rightarrow [q]\Box\psi^\flat$, by inductive hypothesis, we have $\vdash^+\psi_1^\flat \rightarrow [q]\Box\psi^\flat$. Applying rule $([], gen)$, there is $\vdash^+[q^*]\psi_1^\flat \rightarrow [q^*][q]\Box\psi^\flat$. From $\vdash^+\phi^\flat \rightarrow [q^*]\psi_1^\flat$ and $\vdash^+[q^*]\psi_1^\flat \rightarrow [q^*][q]\Box\psi^\flat$, we conclude that $\vdash^+\phi^\flat \rightarrow [q^*][q]\Box\psi^\flat$. Applying rule $(*, \Box\phi)$ and other rules in FOL, it is easy to see that $\vdash^+\phi^\flat \rightarrow [q^*]\Box\psi^\flat$.

□

Yuanrui Zhang is a researcher at RISE lab, Southwest University. He received his Phd at East China Normal University, China. He received his BS degree in pure and applied mathematics, and his MS degree in computer science. His current research interests are verification of real-time systems, theorem proving theory, automatic reasoning and their applications; formal modelling and verification of cyber-physical systems.

Frédéric Mallet is a full Professor in the Informatics Department , University of Nice Sophia Antipolis, France. He is also a member of the KAIROS team-project, a joint team between the I3S laboratory (UMR CNRS) and the INRIA research center Sophia-Antipolis Méditerranée. His current research interests focus on modelling, simulation and verification of real-time and embedded systems, model-driven engineering, parallel and distributed computing, computer architecture, modelling and verification of cyber-physical systems.

Zhiming Liu is currently a professor at Northwest Polytechnical University, China. He is also a part-time professor at Southwest University, China (the Recruitment Program of Global Experts). He received his PhD degree from University of Warwick in 1991. His principal research interest focuses on formal methods. He received a second prize of Natural Science Category of the Macau SAR Natural Science Award in 2012. He has published more than 150 journal and conference papers.