ELSEVIER Annals of Pure and Applied Logic 96 (1999) 167–186

ANNALS OF PURE AND APPLIED LOGIC

Computation paths logic: An expressive, yet elementary, process logic

David Harel *,[1], Eli Singerman [2]

Department of Applied Mathematics and Computer Science, The Weizmann Institute of Science, Rehovot, Israel

Received 30 March 1997; received in revised form 16 July 1997; accepted 7 November 1997

Communicated by S.N. Artemov

Abstract

A new process logic is defined, called computation paths logic (CPL), which treats formulas and programs essentially alike. CPL is a pathwise extension of PDL, following the basic process logic of Harel, Kozen and Parikh, and is close in spirit to the logic R of Harel and Peleg. It enjoys most of the advantages of previous process logics, yet is decidable in elementary time. We also offer extensions for modeling asynchronous/synchronous concurrency and infinite computations. All extensions are also shown to be decidable in elementary time. © 1999 Elsevier Science B.V. All rights reserved.

AMS classification: 03; 68

Keywords: Process logic; Dynamic logic; Temporal logic; Computation paths

1. Introduction

Two major approaches to modal logics of programs are dynamic logic [16] and temporal logic [15]. Propositional dynamic logic, PDL [5], is a natural 'dynamic' extension of the propositional calculus, in which programs are intermixed with propositions in a modal-like fashion. Formulas of PDL can express many input/output properties of programs in a natural way. Moreover, validity/satisfiability in PDL is decidable in exponential time, and the logic has a simple complete axiomatization [11]. PDL is thus a suitable system for reasoning about the input/output behavior of sequential programs on the propositional level. However, PDL is unsuited for dealing with the continuous, or progressive behavior of programs, i.e., the situations occuring *during* computations. The

* Corresponding author.
[1] This author holds the William Sussman chair in mathematics.
[2] Current address: Computer Science Laboratory, SRI International, Menlo Park, California.

need for reasoning about continuous behavior arises naturally in the study of reactive and concurrent programs.

The main approach proposed in response to this need is temporal logic, TL [15], in which assertions can be made naturally about the progressive behavior of programs. In particular, TL can easily express freedom from deadlock, liveness, and mutual exclusion. The basic versions of TL, however, are not compositional, in the sense that their treatment of a well-structured program does not derive directly from their treatment of its components. Indeed, TL usually does not name programs at all, but refers to instructions and labels in a fixed program. Although TL can discuss the synthesis of complex programs from simpler ones to some extent using *at* predicates, this method is rather cumbersome.

This dichotomy between the dynamic and temporal logic approaches has prompted researchers to try to combine the best of the two in what is generally called *process logic*. Accordingly, a system called PL was proposed in [6]. It borrows the program constructs and modal operators $[\ \ ]$ and $\langle\ \ \rangle$ from DL, and the temporal connectives **suf** (similar to **until**) and **f** (standing for **first**) from TL, and combines them into a single system. The expressive power of PL is greater than that of PDL and of TL, and its validity/satisfiability problem was shown in [6] to be decidable, though it is not known to be elementary.[3]

There are some inconvenient features of PL, including the asymmetry of its central path operator, **suf**, and the fact that its formula connectives are somewhat weaker than its program operators. A proposal that overcomes these problems is the regular process logic, RPL, of [7]. In RPL, the operators **suf** and **f** are replaced by **chop** and **slice**, corresponding essentially to Kleene's regular operations of concatenation and star. In this way, the regular operations on programs, $\alpha \cup \beta$, $\alpha\beta$, α^*, have natural counterparts on formulas: $X \vee Y$, X **chop** Y and **slice** X. It is shown in [7] that RPL is even more expressive than PL, and that its validity problem is also decidable but nonelementary.

Using the fact that in RPL both program and path operators are those of regular expressions, and that programs and formulas are interpreted over paths, a *uniform* process logic R was defined in [7]. In *R*, formulas are constructed inductively from atomic propositions and binary atomic programs, using a single set of regular operators. It was shown in [7] that *R* is more expressive than RPL with binary atomic programs, and is decidable (though, again, nonelementary).

In the interest of obtaining a useful process logic decidable in elementary time, an automata-oriented logic, YAPL, was defined in [18]. In YAPL, formulas are constructed using finite automata for both temporal (path) connectives and for constructing compound programs from basic (atomic) ones. There is a clear distinction between state and path formulas in YAPL, atomic programs are binary and atomic formulas are restricted to being state formulas. YAPL is indeed shown in [18] to be decidable in

[3] Some versions of PL have been shown to be nonelementary [10], but it is still not known whether PL itself is elementary.

elementary time (even over infinite paths). YAPL formulas, however, can be somewhat less intuitive and not that easy to comprehend.

In the present paper, we try to combine some of the advantages of previous methods by introducing a new process logic that is compositional, uniform in its treatment of programs and formulas, expressive enough to capture the interesting path properties mentioned in the literature in a natural way, explicit in its treatment of concurrency, and elementary decidable.

We term our basic formalism *computation paths logic* (CPL). A single set of regular operators acts on both transition formulas (programs) and state formulas. For example, $a^* \cdot P \cdot b$ is a CPL formula. (Here a and b are atomic programs and P is an atomic state formula.) Intuitively, this formula means: "perform action a some nondeterministic number of times, check for property P and then do action b". An important operator in CPL is '$\cap$' — pathwise intersection. Thus, $f \cap g$ is true on paths that satisfy both f and g. Using this operator, it is possible to express a large variety of properties of computation paths. For example, $\alpha \cap (skip^* \cdot P \cdot skip^*)$, where α is a program and P is a proposition, is true on α-paths that contain some P-state. Note that $a \cap b$, for atomic programs a and b is true only for paths which are both a-paths and b-paths, and is not expressible by PDL programs or formulas.

Unlike PL and its descendants, RPL and R, we have decided not to include the modal operators [ ] and $\langle\ \rangle$ in CPL. The reason is as follows. Consider a PL/RPL/R formula of the form $[\alpha]\varphi$, where α is a program and φ is a path property. While one might expect this formula to be true on all α-paths that satisfy φ, in PL it is defined to be true on all paths p which, when extended by an α-path r, result in a path $p \cdot r$ satisfying φ. This, however, corresponds to the above intuition only when p is a path of length 0, i.e., a state. This broader (and somewhat complicated) definition in PL is an unavoidable outcome of the wish of the authors of [6] to have only path formulas, but at the same time use $\langle\ \rangle$ and [ ] as in PDL. (For example, they wanted $\langle\alpha\beta\rangle\varphi$ to be equivalent to $\langle\alpha\rangle\langle\beta\rangle\varphi$.)

To make our logic elementary decidable, we use a special form of negation. Specifically, negation in CPL is not taken relative to the set of all paths (as is done, e.g., in PL/RPL/R). In fact, a negated formula is a *state* property, made true in any state that is not the initial state of a path that satisfies the argument formula. For example, $\neg(a \cdot P)$ asserts "it is not possible to carry out a computation of $a \cdot P$ from the present state". While this form of negation is weaker than negation relative to all paths, most interesting path properties are still expressible.

In Section 3, we show that CPL is elementary decidable, by reducing its satisfiability problem to that of APDL, the version of PDL in which programs are represented by finite automata rather than regular expressions [9]. The reduction is rather involved, and combines ideas from both [14] and [17].

In Section 4, we propose an extension of CPL for modeling concurrent processes, called ICPL. It uses '$\|$' to denote interleaving. Interleaving is one of the simplest ways to model concurrent computation (see, e.g., [12]), capturing processes that run concurrently in such a fashion that their atomic steps can be arbitrarily interleaved but with no communication taking place. This might be termed *asynchronous* concurrency. Even though the interleaving operator itself is very intuitive, combining it with other operators (especially '$\cap$') turns out to be rather technically involved. Nevertheless, we are able to extend our previous reduction, and show that ICPL is also decidable in elementary time.

To model *synchronous* concurrency, we introduce a further extension in Section 5, called SICPL (ICPL with synchronization). In SICPL, interleaving can be synchronized with respect to subsets of atomic programs. For such a subset *syn*, and formulas f and g, the interleaving of f and g synchronized on *syn* is expressed by $f \mid syn \mid g$ (the notation is apt, since '$\|$' denotes the special case where $syn = \emptyset$). For example, the formula $(a \cup b) \cdot c \mid a,b \mid (a \cup c) \cdot P \cdot (b \cup c)$ is true only in paths of the form:

$$\xrightarrow{a}\xrightarrow{c}\overset{P}{}\xrightarrow{c} \qquad \xrightarrow{a}\overset{P}{}\xrightarrow{c}\xrightarrow{c} \qquad \xrightarrow{c}\overset{P}{}\xrightarrow{b}\xrightarrow{c}$$

We prove that SICPL is also elementary, by extending the reduction to APDL mentioned above.

A further extension of CPL for expressing properties of infinite computations, ωCPL, is defined in Section 6. In ωCPL, we add the operator $^{\omega}$, which has the following intuitive meaning. Given a formula f, f^{ω} is true in the first state of a ω-path obtained by fusing infinitely many f-paths. With this interpretation, $^{\omega}$ plays a role similar to that of *repeat* in RPDL [8], and indeed, we prove that ωCPL is elementary by a reduction to an automata version of RPDL.

2. Definitions and basic observations

Definition 1. A *path* over a set S is simply a nonempty sequence of elements of S. We use s_i for elements of S, so that (s_1, s_2, s_3) is an example of a path over S. In the sequel, all paths are assumed to be over the same set S. For a path p, denote by $first(p)$ and $last(p)$ the first and the last elements of p. The *length* of a path p, denoted $|p|$, is the number of elements in p minus 1. The set of all paths of length 0 is denoted $(S) = \{(s) \mid s \in S\}$. For paths p and q with last(p)=first(q), the *fusion* of p and q, denoted $p \cdot q$, is the path obtained by writing the elements of p and then the elements of q, omitting one occurrence of $last(p)$. For example, $(s_1,s_2,s_3)\cdot(s_3,s_4) = (s_1,s_2,s_3,s_4)$. For sets of paths $\mathscr{P}$ and $\mathscr{Q}$, the operations $\mathscr{P}\cdot\mathscr{Q}$, $\mathscr{P}^*$, and $first(\mathscr{P})$ are defined in the natural way.

We now define *computation paths logic*, CPL for short. It has two sorts, a set ASF of atomic state formulas (propositions), and a set ATF of atomic transition formulas (programs). The set of formulas is defined as the least set containing ASF and ATF, and such that if f and g are formulas, then so are $(\neg f)$, (f^*), $(f \cdot g)$, $(f \cup g)$ and $(f \cap g)$. (We often omit the parentheses where there is no confusion.)

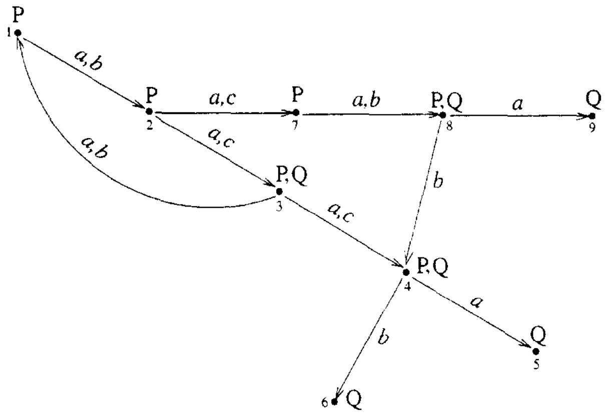


Fig. 1. A CPL model.

CPL formulas are interpreted over models $M = (S_M, \rho_M)$, where S_M is the set of *states*, $\rho_M(P) \subseteq (S_M)$ for each $P \in$ ASF, and $\rho_M(a) \subseteq S_M \times S_M$ for each $a \in$ ATF. In addition, ρ_M is is extended to all formulas as follows:

- $\rho_M(f \cdot g) = \rho_M(f) \cdot \rho_M(g)$.
- $\rho_M(f \cup g) = \rho_M(f) \cup \rho_M(g)$.
- $\rho_M(f \cap g) = \rho_M(f) \cap \rho_M(g)$.
- $\rho_M(f^*) = \rho_M(f)^*$.
- $\rho_M(\neg f) = (S_M) \setminus first(\rho_M(f))$.

(We often leave out the M subscript of S and ρ.) A path p in a model M *satisfies* a CPL formula f, written $M,\ p \models f$, when $p \in \rho_M(f)$. A formula f is *satisfiable* iff $M,\ p \models f$ for some path p in some model M. A state s in a model M *satisfies* a CPL formula f, written $M,\ s \models f$ iff there exist a path satisfying f whose first state is s.

Example. Consider the CPL formula $\varphi : (P \cdot a)^* \cdot Q \cap (b \cup c)^* \cdot \neg(b \cdot P) \cdot a$, where $P,\ Q \in$ ASF and $a,\ b \in$ ATF. In the model illustrated in Fig.1, paths that satisfy φ are (among others): (1, 2, 3, 4, 5), (1, 2, 3, 4) and (1, 2, 3, 1). On the other hand, a path that does *not* satisfy φ is (1, 2, 7, 8, 9) (this is because (8) $\not\models \neg(b \cdot P)$). For CPL formulas f and g, it is sometimes convenient to use the following abbreviations: $f?$ instead of $\neg\neg f$, this test is a state formula with $\rho_M(f?) = first(\rho_M(f))$, $f \vee g$ instead of $f? \cup g?$, and $f \wedge g$ instead of $f? \cap g?$. Regarding transitions, it useful to adopt the following abbreviations: *skip* instead of $\cup_{a \in \mathrm{ATF}} a$, *path* instead of *skip**, and *true* instead of *path*?. Note that *path* holds in every path in which consecutive states are connected by some atomic transition. Moreover, it follows from the semantics of CPL that for every $f \in$ CPL and every path p in any model M, if $p \in \rho_M(f)$ then $p \models$ *path*. So that *path* plays the role of 'true' for paths that correspond to formulas. The formula *true* is a 'state version' of *path* and is true in every path of length 0, i.e., in every state in every model.

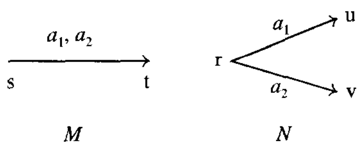
Let us demonstrate how to express some useful path properties in CPL.

- The existence of some segment of the path satisfying f is expressed by $someseg(f) = path \cdot f \cdot path$.
- The existence of some prefix of the path satisfying f is expressed by $somepre(f) = f \cdot path$.
- The existence of some suffix of the path satisfying f is expressed by $somesuf(f) = path \cdot f$.
- The existence of some state in the path satisfying f is expressed by $somestate(f) = someseg(f?)$.
- An operator similar to $\bigcirc$ of TL is $next(f) = skip \cdot f$.
- An operator similar to $\mathcal{U}$ of TL is $f\ until\ g = (f \cdot skip)^* \cdot g$.

CPL can clearly be viewed as a pathwise extension of PDL. To embed PDL in CPL, we associate with every PDL formula/program x, a corresponding CPL formula, denoted x'. Defining x' for boolean combinations and regular operations is easy. The $\langle\ \rangle$ of PDL is dealt as follows: $(\langle\beta\rangle f)' = (\beta' \cdot f')?$. (As in R [7], this last equality nicely brings out the uniform treatment of formulae and programs in CPL.)

Note that every PDL model $M = (S,\ \tau,\ R)$ is a priori also a CPL model; simply take $\rho(P) = (\tau(P))$ and $\rho(a) = \tau(a)$, for every $P \in$ ASF and $a \in$ ATF, respectively. Under this model correspondence it is easy show that PDLs expressive power is no more than that of CPL.

Actually, CPL is *more* expressive than PDL. For such a comparison to be fair, however, we ought to consider only state formulas of CPL (i.e., formulas satisfiable only in paths of length 0). As a trivial example, take the CPL formula $f = (a_1 \cap a_2)?$, where $a_1,\ a_2 \in$ ATF. We claim that f has no equivalent in PDL. To prove this, consider the following two models:



Obviously, $M, (s) \models_{\text{CPL}} f$ and $N, (r) \not\models_{\text{CPL}} f$. However, it can be verified that for every PDL formula g: $M, s \models_{\text{PDL}} g$ iff $N, r \models_{\text{PDL}} g$.

Considering other process logics, CPL can be thought of as a restricted version of the logic R of [7]. The main differences between CPL and R are the following. CPL does not use the modal operators $[\ \ ]$ and $\langle\ \ \rangle$, in CPL we consider only (finite) paths that arise from computations and negation in CPL is not relative to all paths, but rather is a state property.

In spite of these differences, it is not difficult to show that CPL can be embedded in R, and then use the fact that R is decidable [7] to conclude that CPL is decidable. Since R is nonelementary [14], this yields a nonelementary decision procedure for CPL. We will show in the next section, however, that CPL is, in fact, elementary.

3. CPL is elementary decidable

In this section we show that satisfiability of CPL formulas is decidable in elementary time. This will be done in two steps. In the first, we carry out a reduction from the satisfiability problem of CPL to the satisfiability problem of CPL over one-action-per-transition models. These *one-action-per-transition* (*oapt*, for short) models are defined below. (These models were used in [14] for the logic R.) In the second step we carry out a reduction from the satisfiability problem of CPL over *oapt*-models to the satisfiability problem of APDL.

3.1. Reduction to oapt-satisfiability

Definition 2. A model M is called an *oapt-model* relative to the set $ATF = \{a_1, \dots, a_n\}$ if for every $1 \leqslant i \neq j \leqslant n$, $\rho(a_i) \cap \rho(a_j) = \emptyset$. A CPL formula f is *oapt-satisfiable* iff there exist some oapt-model which satisfies f.

Lemma 3. *For every* CPL *formula f over $\{a_1 \dots a_n\}$, there exists a* CPL *formula f' (over a new* ATF) *such that f is satisfiable iff f' is* oapt-*satisfiable.*

Proof. Let f be a formula over $\{a_1, \dots, a_n\}$. We define a set ATF$'$ of $2^n - 1$ new symbols (to be used as the atomic transition formulas of f'), each of the form $a_{c_1 \dots c_n}$ where $c_k \in \{k, \bar{k}\}$,

$$\text{ATF}' = \{a_{c_1 \dots c_n} \mid \forall\ 1 \leqslant k \leqslant n\ ,\ c_k \in \{k, \bar{k}\}\} \setminus \{a_{\bar{1}\bar{2} \dots \bar{n}}\}.$$

Let f' be the formula obtained from f by replacing every appearance of a_k, for $1 \leqslant k \leqslant n$, with

$$\beta_k = \bigcup_{\{c_1 \dots c_n \mid c_k = k\}} a_{c_1 \dots c_n}.$$

The following claim completes the proof of the lemma.

Claim. *f is satisfiable $\iff$ f' is* oapt-*satisfiable.*

Proof. (*Claim*) ($\Longleftarrow$): assume $M', p \models f'$. We build a model M for f similar to M' without $\rho_{M'}(a_{c_1 \dots c_n})$, but with $\rho_M(a_k) = \rho_{M'}(\beta_k)$. Since this is the only difference between f and f', it is straightforward to check that $M, p \models f$.

($\Longrightarrow$): assume $M, p \models f$. We build a model M' for f' similar to M without $\rho_M(a_k)$, but with

$$\rho_{M'}(a_{c_1 \dots c_n}) = \bigcap_{\{k \mid 1 \leqslant k \leqslant n,\ c_k = k\}} \rho_M(a_k) \cap \bigcap_{\{k \mid 1 \leqslant k \leqslant n,\ c_k = \bar{k}\}} \overline{\rho_M(a_k)},$$

where $\overline{\rho_M(a_k)}$ denotes $S \times S - \rho_M(a_k)$. To complete the proof it suffices to show that for every $1 \leqslant k \leqslant n$, $\rho_{M'}(\beta_k) = \rho_M(a_k)$ (again, since this is the only difference between f and f'):

$$\begin{aligned}
\rho_{M'}(\beta_k) &= \bigcup_{\{c_1 \ldots c_n | c_k = k\}} \rho_{M'}(a_{c_1 \ldots c_n}) \\
&= \bigcup_{\{c_1 \ldots c_n | c_k = k\}} \left(\bigcap_{\{l | 1 \leqslant l \leqslant n,\ c_l = l\}} \rho_M(a_l) \cap \bigcap_{\{l | 1 \leqslant l \leqslant n,\ c_l = \bar{l}\}} \overline{\rho_M(a_l)} \right) \\
&= \bigcup_{c_1 = 1, \bar{1}} \cdots \bigcup_{c_{k-1} = k-1, \overline{k-1}} \ \bigcup_{c_{k+1} = k+1, \overline{k+1}} \cdots \bigcup_{c_n = n, \bar{n}} \\
&\quad \times \left(\bigcap_{\{l | 1 \leqslant l \leqslant n,\ c_l = l\}} \rho_M(a_k) \cap \bigcap_{\{l | 1 \leqslant l \leqslant n,\ c_l = \bar{l}\}} \overline{\rho_M(a_k)} \right) \\
&= \rho_M(a_k). \qquad \square
\end{aligned}$$

3.2. Reduction to APDL

As preparation for the reduction to APDL, let us start the discussion in the framework of PDL. Recall that a PDL model is also a CPL model; note, however, that while CPL formulas are interpreted over *paths*, PDL formulas are interpreted over *states*. To overcome this dichotomy we shall relate paths to PDL programs in the following way:

Definition 4. For a PDL program α and a path $p = (p_1, \ldots, p_k)$ in a model (S, τ, R), $p \in \alpha$ is defined by induction on the structure of α:

If $\alpha \in \text{ATF}$ then $p \in \alpha$ iff ($k = 1$ and $(p_0, p_1) \in R(\alpha)$)
$p \in \alpha \cup \beta$ iff $p \in \alpha$ or $p \in \beta$
$p \in \alpha; \beta$ iff there are paths $q \in \alpha$ and $r \in \beta$ with $p = q \cdot r$
$p \in \alpha^*$ iff $p \in \alpha^i$ for some $i \geqslant 1$ or $p = (p_0)$
$p \in \varphi?$ iff $p = (p_0)$ and $(p_0, p_0) \in R(\varphi?)$

Via this association we can view PDL programs as being carried out along paths rather than as binary relations. For the reduction, however, it is more convenient to use the automata version of PDL, namely APDL [9]. The reason for this is that '$\cap$' can be handled more economically by automata than by regular expressions. (This also applies to other operators used in the extensions of CPL we define later on.)

Remark. Using PDL with intersection (IPDL) does not help, since the intersection in CPL is pathwise and different from that of IPDL. As a matter of fact, the 2EXPTIME upper bound for deciding IPDL, was also obtained via a reduction to APDL [1], where one exponent is due a cross product automaton used to handle the intersection operator.

APDL formulas are, in general, more succinct than their equivalent PDL formulas. Nevertheless, satisfiability for APDL can be decided in EXPTIME [9]. This is also the case for deciding satisfiability over oapt-models. For if $M, s \models \varphi$, then M can be transformed into an oapt-model of φ. One way of doing this is to unwind the model into a tree, which, in particular, is an oapt-model. The resulting tree model might be infinite, but the point here is that it is possible, and therefore the complexity of deciding APDL over oapt models is exactly as for unrestricted models. Simply, check satisfiability without any restriction and you have the answer!

We shall use this to get an elementary decision procedure for CPL by carrying out a reduction from CPL into APDL. Relating paths in a model to APDL programs is done as in Definition 4, i.e., if α is an automaton (APDL program) then $p \in \alpha$ iff $p \in r(\alpha)$, where $r(\alpha)$ is a regular expression denoting the language of α.

Lemma 5. *For every* CPL *formula f there exists an* APDL *program* (NFA) A_f, *such that for every path p in every oapt-model, $p \in \rho(f)$ iff $p \in A_f$.*

Proof. The *APDL* automaton (program) A_f corresponding to the CPL formula f is built by induction on the structure of f.

- $P \in$ASF: let $A_P = \langle \{k_0, k_1\}, k_0, \delta_{P?}, \{k_1\} \rangle$, where $\delta_{P?}(k_0, P?) = \{k_1\}$. Thus, the language accepted by A_P is simply $\{P?\}$.
- $a \in$ATF: similarly, let A_a be a two state NFA accepting the language $\{a\}$.
- $\neg g$: let $A_{\neg g}$ be a two state NFA accepting the language $\{([A_g]false)?\}$.
- $g \cdot h$: let $A_{g \cdot h}$ be an NFA with $L(A_{g \cdot h}) = L(A_g); L(A_h)$.
- g^*: let A_{g*} be an NFA with $L(A_{g^*}) = L(A_g)^*$.
- $g \cup h$: let $A_{g \cup h}$ be an NFA with $L(A_{g \cup h}) = L(A_g) \cup L(A_h)$.
- $g \cap h$: this case should be dealt with more carefully since the '$\cap$' in CPL is intersection in the *path* sense rather than in the *language* sense. We use the fact that we are dealing with oapt-models and build $A_{g \cap h}$ that simulates both A_g and A_h synchronizing on ATF-letters. To be specific let

$$A_{g \cap h} = \langle K_g \times K_h,\ (s_g, s_h),\ \delta_{g \cap h},\ F_g \times F_h \rangle,$$

where

(i) $\delta_{g \cap h}((k, k'), \varphi?) = (\delta_g(k, \varphi?) \times \{k'\}) \cup (\{k\} \times \delta_h(k', \varphi?))$, for $\varphi \in Prop_{?}$

(ii) $\delta_{g \cap h}((k, k'), a) = \delta_g(k, a) \times \delta_h(k', a)$, for $a \in$ATF. □

We can now prove the main theorem.

Theorem 6. *If we fix* ATF *to be a subset of* $\{a_1, \ldots, a_n\}$, *then satisfiability of* CPL *formulas can be decided in* 2EXPTIME.

Proof. Let f be a CPL formula over ATF $\subseteq \{a_1,\dots,a_n\}$. Use Lemma 3 to construct f' with new atomic transition formulas ATF$'$, such that f is satisfiable iff f' is oapt-satisfiable. Note that since the set $\{a_1,\dots,a_n\}$ is fixed, $|f'| = c_1\cdot|f|$, for some constant c_1. By Lemma 5, there exists an APDL program $A_{f'}$ (in the form of an NFA over the alphabet ATF$'\cup Prop_A$) such that for every path p in every oapt-model $M : p\in\rho(f)$ iff $p\in A_{f'}$. In other words, $p\in\rho(f)$ iff $first(p)\models\langle A_{f'}\rangle true$. It is known [9], that satisfiability of APDL formulas can be decided in deterministic exponential time. One can easily prove by induction on the structure of f' that $|A_{f'}|\leqslant 2^{c_2\cdot|f'|}$, for some constant c_2 (actually, the exponent is needed only for the '$\cap$' case). So that the overall time complexity of deciding satisfiability of the original CPL formula f is bounded by $2^{2^{c_3\cdot|f|}}$, for some constant c_3. $\square$

Remark. Deciding satisfiability of CPL formulas without the restriction to a fixed finite subset of atomic transition formulas costs one more exponent (in this case, $|f'|$ is exponential in $|f|$). This will also apply to all the extensions of CPL in the sequel.

4. CPL with interleaving

The motivation for adding the interleaving operator to CPL is twofold. Our primary motivation is that the interleaving operator can be interpreted as the simplest case of composition used in algebraic approaches to modeling concurrent computation (see, e.g., [12]). Interleaving represents the case where processes run concurrently in such a fashion that their atomic steps can be arbitrarily interleaved but where no communication between them takes place. This form of concurrency, modeled by interleaving, might also be described as *asynchronous*. Second, as discussed in the sequel, using interleaving we gain succinctness.

Let us now define ICPL (CPL with interleaving). The syntax of ICPL extends that of CPL as follows: if f and g are formulas, then so is $(f \parallel g)$. Turning to the semantics, the basic difficulty is that our ρ, which associates paths with formulas, is not informative enough to capture interleaving. For example, we would like the formula $(a\cdot P)\parallel b$ to be satisfied by paths of the following forms:

$$\xrightarrow{a}{}^{P}\xrightarrow{b} \qquad\qquad \xrightarrow{a}\xrightarrow{b}{}^{P} \qquad\qquad \xrightarrow{b}\xrightarrow{a}{}^{P}$$

However, paths of the second form would not appear if we used $\rho(a\cdot P)$ and $\rho(b)$, since $\rho(a\cdot P)$ contains only 'a'-paths with P at the last state. To solve this problem we shall use a more detailed version of ρ. The idea is that now $\rho_M(f)$ will contain, in addition to paths in M that are associated with f, some 'evidence' of this association. We will associate with each formula (via this extended ρ) a set of *computation paths* (defined below) rather than a set of (ordinary) paths. A computation path in a model M consists of two objects: a *computation*, which is a sequence of transitions accompanied by a sequence of properties (state formulas); and an ordinary path over M, i.e., a sequence of states of M. To get a feeling for this, the figure below illustrates a computation path:

$$\begin{tikzcd}
P & \neg(b \cdot Q) & R \\
s \arrow[r, "a"] & t \arrow[r, "{a,c}"] & r
\end{tikzcd}$$

Here, the *path* is (s,t,r), i.e., the sequence of states, and the *computation* is $\langle(a,\{a,c\}), (P,\neg(b\cdot Q),R)\rangle$.

To make the discussion above more formal, we need some preparations. We first define the notion of a *computation*, and some operations on computations and on computation sets. Then, we combine computations and paths to define *computation paths*, and define several operations on them. Finally, we present the semantics of ICPL.

Definition 7. The set of *state formulas* SF is the minimal set of $ICPL$ formulas that contains ASF, contains all formulas of the form $\neg f$, and is closed under $\cdot$ and $\cap$. For state formulas f and g of the form $f = f^1 \cdot f^2 \ldots \cdot f^k, g = g^1 \cdot g^2 \ldots \cdot g^l$, where $k, l \geqslant 1$, let

$$f \dot{\cap} g = \begin{cases} (f^1 \cap g^1) \cdot \ldots \cdot (f^k \cap g^l), & k = l, \\ (f^1 \cap g^1) \cdot \ldots \cdot (f^l \cap g^l) \cdot f^{l+1} \cdot \ldots \cdot f^k, & k > l, \\ (f^1 \cap g^1) \cdot \ldots \cdot (f^k \cap g^k) \cdot g^{k+1} \cdot \ldots \cdot g^l, & k < l. \end{cases}$$

Definition 8. A *computation* is a pair $c = \langle Tran_c, Val_c\rangle$, where $Tran_c$ is a path over the set $2^{ATF} - \emptyset$ and Val_c is a path of length $|Tran_c| + 1$ over the set SF. The *length of* c, denoted $|c|$, is $|Val_c| - 1$.

We now define several operations on computations. For this we use the two computations:

$$c = \langle \overbrace{(t_1,\ldots,t_k)}^{Tran_c}, \overbrace{(f_0,\ldots,f_k)}^{Val_c} \rangle \quad \text{and} \quad d = \langle \overbrace{(r_1,\ldots,r_l)}^{Tran_d}, \overbrace{(g_0,\ldots,g_l)}^{Val_d} \rangle$$

– $c \cdot d \overset{\text{def}}{=} \langle (Tran_c);(Tran_d),\ (Val_c)\hat{\ }(Val_d)\rangle$, where
$(t_1,\ldots,t_k)\ ;\ (r_1,\ldots,r_l) = (t_1,\ldots,t_k,r_1,\ldots,r_l)$ and
$(f_0,\ldots,f_k)\hat{\ }(g_0,\ldots,g_l) = (f_0,\ldots,f_k \cdot g_0,\ldots,g_l)$.

– If c and d are of the same length (i.e., $k = l$),
then $c \cap d \overset{\text{def}}{=} \langle (t_1 \cup r_1,\ldots,t_k \cup r_k),\ (f_0 \dot{\cap} g_0,\ldots,f_k \dot{\cap} g_k)\rangle$.

The next operation we want to define is $c \parallel d$. In general, $c \parallel d$ is a set of computations. A computation in $c \parallel d$ is obtained by sequentially executing portions from c or from d. Let us make this notion more precise. First, denote by $I_c \subseteq \{0,\ldots,k\}$ the set of indices s.t. $i \in I_c$ iff f_i is of the form $f_i^1 \cdot f_i^2 \cdots f_i^{last(f_i)}$ (and $last(f_i) \geqslant 2$). Next, define a *formula portion* of c to be any element of the set

$$\left(\bigcup_{i \in \{0,\dots,k\} - I_c} f_i \right) \cup \left(\bigcup_{i \in I_c} \bigcup_{m=1}^{f_i^{l(i)}} f_i^m \right).$$

Finally, a *portion* of c is a formula portion or a *transition portion* of c, where a transition portion of c is an element of $\{r_i\}_{i=1}^{l}$. Portions of d are defined in a similar way.

Constructing a computation $e \in c \parallel d$ is carried out as follows. Initialize $Tran_e$ and Val_e with ( ), and set pointers to the leftmost formula portions of c and d. While there remain portions of c and d, that have not been dealt with, nondeterministically add to e the next portion of c or that of d, and advance the corresponding pointer to the next portion, where the successor of a transition is a formula and the successor of a formula portion is either the next portion of the same formula or the next transition, if the current portion is last in the formula. When one of c or d has been consumed, simply add to e the remaining portions of the other. The construction above can be put on a more formal basis by the following algorithm.

```
(i) (a) let $next(c)$ and $next(d)$ be the leftmost formula portions of $c$ and $d$, respectively.
    (b) $Tran_e := Val_e := (\ )$.
    (c) Last:=*false*. {this variable indicates whether the recent portion added to $e$ is a transition}
(ii) (a) if $next(c) = \emptyset$ and $next(d) = \emptyset$ then halt.
     (b) go to step (3) or step (4) nondeterministically.
(iii) if $next(c) = \emptyset$ then go to step (2).
      if $next(c)$ is a formula portion
        then begin
          (a) if Last then $Val_e := Val_e\ ;\ (next(c))$ else $Val_e := Val_e\ \hat{}\ (next(c))$.
          (b) if $next(c) = f_i^x$ for some $i \in I_c$, $x < last(f_i)$ then $next(c) := f_i^{x+1}$
          else let $next(c)$ be the next transition portion of $c$, i.e., $next(c) := r_{i+1}$.
          (If there is no transition $r_{i+1}$, that is $i = l$, then $next(c) := \emptyset$.)
          (c) Last:=*false*, and go to step (2).
            end
        else begin {$next(c)$ is a transition portion}
          (a') $Tran_e := Tran_e\ ;\ (next(c))$.
          (b') if Last then $Val_e := Val_e\ \hat{}\ (true)$.
          (c') let $next(c)$ be the next formula portion of $c$ (for example, if the
          current value of $next(c)$ is $r_i$, then the next value of $next(c)$ will be
          $f_i^1$ in case $i \in I_c$, and $f_i$ otherwise).
          (d') Last:=*true*, and go to step (2).
            end
(iv) analogous to step (3) with $next(d)$ instead of $next(c)$, $I_d$ replacing $I_c$, etc.
```

We are now ready to define the notion of *computation path.*

Definition 9. A *computation path* in a model M is a pair $p = (Stat_p,\ c_p)$, where $Stat_p$ is a nonempty path over S_M (i.e., an ordinary path in the model M) and c_p is a computation with $|Val_{c_p}| = |Stat_p|$.

For a computation path $p = (Stat_p,\ c_p)$, we denote $Tran_{c_p}$ and Val_{c_p} by $Tran_p$ and Val_p, respectively. We intend to use a computation path p as follows: $Stat_p$ will be the states along p, $Tran_p$ will be the sequence of transitions along p, and Val_p will be the sequence of state formulas satisfied in states along p. For example, a computation path p with $Stat_p = (s,\ t,\ r)$, $Tran_p = (\{a,b\},\ a)$ and $Val_p = (True,\ P\cdot Q,\ \neg(a \cup c))$ is illustrated by:

$$\begin{tikzcd} s \arrow[r, "{a,b}"] & \overset{P\cdot Q}{t} \arrow[r, "a"] & \overset{\neg(aUc)}{r} \end{tikzcd}$$

We have defined $\cdot$ both on computations and on paths, and we now use these together to define $p\cdot q$, for computation paths p and q (and then, extend it to sets of computation paths in the usual way): $p \cdot q \overset{\text{def}}{=} (Stat_p \cdot Stat_q,\ c_p \cdot c_q)$.

Definition 10. Let CP be a set of computation paths in a model M. A path $p = (s_0, \ldots, s_k)$ in M is CP consistent with a computation $c = \langle (t_1, \ldots, t_l),\ (f_0, \ldots, f_l)\rangle$, if the following conditions are satisfied:
(i) $|p| = |c|$ (i.e., $k = l$),
(ii) for every $0 \leqslant i \leqslant k-1$, there exist $q \in CP$ s.t. $Stat_q = (s_i, s_{i+1})$ and $Tran_q = (t_i)$,
(iii) for every $0 \leqslant i \leqslant k$, there exist $q \in CP$ s.t. $Stat_q = (s_i)$ and $Val_q = (f_i)$.
We can now define the semantics of ICPL. Formulas are interpreted over the same models as in CPL, that is, models of the form $M = (S_M, \rho^0_M)$, where S_M is the set of *states*, $\rho^0_M(P) \subseteq (S)$, for every element $P \in$ ASF, and $\rho^0_M(a) \subseteq S \times S$, for every element $a \in$ ATF.

Next, ρ^0_M is extended by induction to a function ρ_M, which assigns a set $\rho_M(f)$ of computation paths to every ICPL formula f. The set of all computation paths assigned to formulas in this way (i.e., those that are in $\rho_M(f)$ for some f) is denoted $CP(M)$. All the inductive cases in the definition of ρ_M are straightforward, except for the following two:

$$\rho_M(f \cap g) = \{r \mid \exists p \in \rho_M(f),\ q \in \rho_M(g) \text{ s.t. } Stat_r = Stat_p = Stat_q \\ \text{and } c_r = c_p \cap c_q\}$$

$$\rho_M(f \parallel g) = \{r \mid Stat_r \text{ is } CP(M) \text{ consistent with } c_r \text{ and } c_r \in (c_p \parallel c_q), \\ \text{for some } p \in \rho_M(f),\ q \in \rho_M(g)\}.$$

Now that we have this extended version of ρ we can extract from it the relevant information as follows.

Definition 11. An ICPL formula f is satisfied in a path p of a model M, written $M,\ p \models f$, iff $p = Stat_q$ for some computation path $q \in \rho_M(f)$. f is *satisfiable* iff $M,\ p \models f$ for some path p of some model M.

How does ICPL relate to CPL? Recall that ICPL is intended to be CPL extended with the'$\|$'-operator. While syntactically it is clear that CPL $\subset$ ICPL, semantically this may seem less obvious due to the differences in the definitions. We therefore proceed by showing that under the canonical correspondence between CPL models and ICPL models, that is, $\rho_{\mathrm{CPL}} = \rho^0_{\mathrm{ICPL}}$, this is indeed the case.

Proposition 12. *For every* CPL *formula f and every* (*ordinary*) *path p in any model M, $M, p \models_{\mathrm{CPL}} f$ iff $M, p \models_{\mathrm{ICPL}} f$.*

Sketch of Proof. Let M be fixed. We show, by induction on the structure of f, that $\rho(f) = Stat(\rho_{\mathrm{I}}(f))$, where $\rho = \rho_{\mathrm{CPL}}$, $\rho_{\mathrm{I}} = \rho_{\mathrm{ICPL}}$, and $Stat(\rho_{\mathrm{I}}(f)) = \{Stat(u) \mid u \in \rho_{\mathrm{I}}(f)\}$.

The base case of the induction follows from the model correspondence. The inductive clauses follow readily from the induction hypothesis. For example, the $f = g \cdot h$ case is proved as follows. Assume $p \in \rho(g \cdot h)$. Then, $p = q \cdot r$ for some $q \in \rho(g)$, $r \in \rho(h)$. By the i.h., $q \in Stat(\rho_{\mathrm{I}}(g))$ and $r \in Stat(\rho_{\mathrm{I}}(h))$. Thus, there exist computation paths $u \in \rho_{\mathrm{I}}(g)$, $v \in \rho_{\mathrm{I}}(h)$, such that $Stat(u) = q$ and $Stat(v) = r$. From this we get $q \cdot r \in Stat(u \cdot v)$, and since $u \cdot v \in \rho_{\mathrm{I}}(g \cdot h)$ we have $p \in \rho_{\mathrm{I}}(g \cdot h)$. The other direction is proved similarly. $\square$

In what sense is ICPL 'better' than CPL? Well, using the well known fact that regular sets are closed under interleaving it is not difficult to prove that ICPL and CPL have the same expressive power. Nevertheless, ICPL has two important advantages over CPL. The first is *clarity* in modeling asynchronous concurrent computations. For example, consider the following two computations: (i) Execute a, observe P and then perform b. (ii) Observe Q and then execute b followed by a.

In ICPL, we can use the formula $a \cdot P \cdot b \,\|\, Q \cdot b \cdot a$ to model computations that arise from running these two in parallel, while in CPL one must use a cumbersome formula of the form

$$(a \cdot P \cdot b \cdot Q \cdot b \cdot a) \cup (a \cdot P \cdot Q \cdot b \cdot b \cdot a) \cup (a \cdot Q \cdot P \cdot b \cdot b \cdot a)$$
$$\cup (a \cdot Q \cdot P \cdot b \cdot b \cdot a) \cup (Q \cdot a \cdot P \cdot b \cdot b \cdot a) \cup \cdots \cup (Q \cdot b \cdot a \cdot a \cdot P \cdot b).$$

Other CPL formulas exist for this, yet it is very unlikely that such a formula will be as transparent as the ICPL formula given above.

The second (and related) advantage of ICPL over CPL is *succinctness*. It is known that the use of the interleaving operator can shorten a regular expression by an exponential amount [4, 13]. It is true that interleaving in ICPL is (in general) not interleaving in the language sense. However, ICPL formulas that use only ATF and the operators '$\cdot$', '$*$', '$\cup$' and '$\|$' correspond essentially to regular expressions (extended with interleaving operator) over the alphabet ATF.

Let us now turn to the problem of deciding the satisfiability status of ICPL formulas. Given an ICPL formula, one can construct an equivalent CPL formula and then use the methods of the previous section to decide the satisfiability status of the latter, and therefore also of the former. The discussion above indicates that the first step of this naive algorithm, i.e., transforming an ICPL formula to an equivalent CPL formula, yields an exponential growth in size. Taking into account that reducing CPL to APDL also involves an exponential growth in size, and that APDL can be decided in EXPTIME, we get a 3EXPTIME algorithm for deciding ICPL. However, we can reduce ICPL directly to APDL (over oapt-models) by using a cross-product automaton to handle the '$\|$' – case (as we did in the '$\cap$' – case in the reduction from CPL to APDL). This way we get a 2EXPTIME decision procedure for ICPL.

Before getting into the technical details of the proof we would like to draw the reader's attention to the following difference between ICPL and CPL. Let f and g be *state formulas*. In CPL, one can replace $f \cap g$ by $f \cdot g$ (and vice versa) without affecting the semantics. However, this is not the case in the presence of '$\|$', that is, in ICPL. For example, consider the state formulas $P \cap Q$ and $P \cdot Q$, where $P, Q \in$ASF. Although these two are equivalent in ICPL, they are not interchangeable. For example, if $a \in$ ATF, then $P \cap Q \parallel a$ is satisfied in paths of the form

$$\xrightarrow{P\cdot Q \qquad a} \qquad \xrightarrow{a \qquad P\cdot Q}$$

while $P \cdot Q \parallel a$ is also satisfied in paths of the form

$$\xrightarrow{P \qquad a \qquad Q}$$

This is because the interleaving operator permits 'breaking up' concatenation, but not intersection (of state formulas). To reflect this in the reduction to APDL we have to modify the construction of the '$\cap$'-automaton of Lemma 5. (For instance, the '$\cap$'-automaton constructed there for $P \cap Q$ would accept $P?Q?$. However, this word would also be accepted by the automaton for $P \cdot Q$; therefore, the distinction noted above would not be preserved.)

We are now ready to carry out the reduction from ICPL formulas to APDL programs. As was done for ordinary CPL, we can restrict our attention to oapt-models, since it is easy to see that Lemma 3 holds for ICPL formulas too.

Lemma 13. *For every* ICPL *formula* f *there exists an* APDL *program* (NFA) A_f, *such that, for every path* p *in every* oapt-*model,* $p \in Stat(\rho(f))$ *iff* $p \in A_f$.

Proof. We build the APDL program $A_f = \langle K_f,\ s_f,\ \delta_f,\ F_f\rangle$ corresponding to the ICPL formula f by induction on the structure of f. The *tests of* A_f is the set $T(A_f) = \{t \mid \cup_{s\in K_f}\delta_f(s,\ t?) \neq \emptyset\}$. The construction of A_f for the '$\cap$' and '$\|$' cases is given below (all other cases are carried out precisely as in the proof of Lemma 5).

– $g \cap h$: given A_g and A_h, let

$$A_{g\cap h} = \langle\ K_g \times K_h \times \{g \wedge h,\ g,\ h\},\ (s_g,\ s_h,\ g \wedge h),\ \delta_{g\cap h},$$
$$F_g \times F_h \times \{g \wedge h,\ g,\ h\}\rangle,$$

where

(i) $\delta_{g\cap h}((k,k',g \wedge h),(t \wedge r)?) = \delta_g(k,t?) \times \delta_h(k',r?) \times g \wedge h,\quad t \in T(A_g) \wedge r \in T(A_h)$
(ii) $\delta_{g\cap h}((k,k',g \wedge h),t?) = \delta_g(k,t?) \times \{k'\} \times g,\quad t \in T(A_g)$
(iii) $\delta_{g\cap h}((k,k',g),t?) = \delta_g(k,t?) \times \{k'\} \times g,\quad t \in T(A_g)$
(iv) $\delta_{g\cap h}((k,k',g \wedge h),r?) = \{k\} \times \delta_h(k',r?) \times h,\quad r \in T(A_h)$
(v) $\delta_{g\cap h}((k,k',h),r?) = \{k\} \times \delta_h(k',r?) \times h,\quad r \in T(A_h)$
(vi) $\delta_{g\cap h}((k,k',g \wedge h),a) = \delta_g(k,a) \times \delta_h(k',a) \times g \wedge h,\quad a \in \mathrm{ATF}$
(vii) $\delta_{g\cap h}((k,k',g),a) = \delta_g(k,a) \times \delta_h(k',a) \times g \wedge h,\quad a \in \mathrm{ATF}$
(viii) $\delta_{g\cap h}((k,k',h),a) = \delta_g(k,a) \times \delta_h(k',a) \times g \wedge h,\quad a \in \mathrm{ATF}$

– $g \parallel h$: given A_g and A_h, let

$$A_{g\|h} = \langle K_g \times K_h,\ (s_g,\ s_h),\ \delta_{g\|h},\ F_g \times F_h\rangle,$$

where

(i) $\delta_{g\|h}((k,k'),t?) = \delta_g(k,t?) \times \{k'\},\quad t \in T(A_g)$
(ii) $\delta_{g\|h}((k,k'),r?) = \{k\} \times \delta_h(k',r?),\quad r \in T(A_h)$
(iii) $\delta_{g\|h}((k,k'),a) = \delta_g(k,a) \times \{k'\} \cup \{k\} \times \delta_h(k',a),\quad a \in \mathrm{ATF}.$ □

It is not difficult to prove by induction on f, that the size of the automaton A_f of the previous lemma is at most exponential in $|f|$, so that we have:

Theorem 14. *Satisfiability of* ICPL *formulas with* $\mathrm{ATF} \subseteq \{a_1,\ldots,a_n\}$ *can be decided in* 2EXPTIME.

5. ICPL with synchronization

ICPL is suited for modeling asynchronous concurrency. To model synchronous concurrency as well, we introduce ICPL with synchronization (SICPL). All ICPL formulas are SICPL formulas. In addition, if f and g are SICPL formulas and *syn* is a subset of ATF, then $f \mid syn \mid g$ is a SICPL formula. (The set *syn* has to be written out in full, for example as in $(a \cdot b)^* \cdot P \mid a,b \mid (a \cup b)$.) Intuitively, $f \mid syn \mid g$ represents the interleaving of f and g synchronized w.r.t. *syn*. See the example in Section 1.

To present the formal semantics of SICPL we have to modify each step in the definition of $\rho_M(f \parallel g)$. First, we extend the algorithm presented in Section 4.4. Recall that for any given computations c and d, the algorithm was used to generate all computations $e \in (c \parallel d)$. Given a set $syn \subseteq \text{ATF}$, to obtain all computations $e \in (c \mid syn \mid d)$, add the following to the algorithm:

```
– to step (1) add the substep: (d) let Suspend$_c$:=Suspend$_d$:=$false$.
– in step (3) (respectively, (4)) add (prior to substep (a')) the step:
  (check) if $next(c) \cap syn \neq \emptyset$ (respectively, $next(d) \cap syn \neq \emptyset$) then Suspend$_c$:=$true$
  (resp. Suspend$_d$:=$true$), and go to step (5).
– at the end of the algorithm add:
  (5) if (Suspend$_c$ and Suspend$_d$)
     then begin
        (a) $Tran_e := Tran_e; (next(c) \cup next(d))$.
        (b) if Last then $Val_e := Val_e\hat{}(true)$.
        (c) let $next(c)$ be the next formula portion of $c$.
        (d) let $next(d)$ be the next formula portion of $d$.
        (e) Suspend$_c$:=Suspend$_d$:=$false$, Last:=$true$, and go to step (2).
          end
     else go to step (2).
```

Note that if the synchronization set is empty ($syn = \emptyset$), then these additions have no effect on the algorithm. And indeed, to impose no synchronization constraints amounts to asynchronization. (This is also reflected in the notation, since if $syn = \emptyset$, then $f \mid syn \mid g$ is simply $f \parallel g$.) Now, for every model M and every synchronization set $syn \subseteq \text{ATF}$, let

$$\rho_M(f|syn|g) = \{r | Stat_r \text{ is } CP(M) \text{ consistent with } c_r \text{ and } \\ c_r \in (c_p|syn|c_q), \text{ for some } p \in \rho_M(f),\ q \in \rho_M(g)\}.$$

Reducing the satisfiability problem of SICPL to that of APDL can be done essentially as for ICPL. Given a SICPL formula f, we use the transformation described in the proof of Lemma 3 to construct a formula f', with atomic transition formulas from ATF$'$,[4] such that f is satisfiable iff f' is oapt-satisfiable. Then, we proceed as in Lemma 13 to construct the corresponding APDL program (automaton) $A_{f'}$, where for the $g \mid syn \mid h$ case, we use

$$A_{g|syn|h} = \langle K_g \times K_h,\ (s_g,\ s_h),\ \delta_{g|syn|h},\ F_g \times F_h \rangle,$$

where

(i) $\delta_{g|syn|h}((k,k'),t?) = \delta_g(k,t?) \times \{k'\}, \quad t \in T(A_g)$,
(ii) $\delta_{g|syn|h}((k,k'),r?) = \{k\} \times \delta_h(k',r?), \quad r \in T(A_h)$,
(iii) $\delta_{g|syn|h}((k,k'),a) = \delta_g(k,a) \times \delta_h(k',a), \quad a \in syn$,
(iv) $\delta_{g|syn|h}((k,k'),a) = (\delta_g(k,a) \times \{k'\}) \cup (\{k\} \times \delta_h(k',a)), \quad a \in \text{ATF} - syn$. □

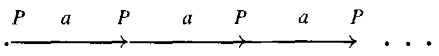
[4] This time, elements of ATF appearing in synchronization sets should also be transformed.

Constructing $A_{f'}$ from f' costs at most an exponential in added size (as was the case for ICPL), so that we have:

Theorem 15. *Satisfiability of* SICPL *formulas with* ATF $\subseteq \{a_1, \dots, a_n\}$ *can be decided in* 2EXPTIME .

6. Infinite computations

CPL (and its extensions ICPL, SICPL) are input/output oriented and are therefore appropriate for stating properties concerning programs with finite computations. We wish, however, to make it possible to reason about processes with possible *infinite* computations. For example, we would like to say that the model illustrated below, admits in addition to the finite computations described by $(P \cdot a)^*$ also the infinite computation $(P \cdot a)^\omega$.



With this idea in mind, we introduce the extension ωCPL. Before giving the formal definition let us try to explain informally the route we have chosen to follow.

Basically, one would like ωCPL to extend CPL by employing the new operator 'ω' and use formulas of the form f^ω, where f is a CPL formula. The most intuitive interpretation of f^ω is simply to associate with it infinite paths that result by fusing infinitely many (finite) paths of f (that is, take $\rho(f^\omega)$ as $\rho(f)^\omega$). Choosing this interpretation, however, forces one to make a distinction between 'ω-formulas' (those with possibly infinite paths corresponding to the ω) and 'finite formulas'. This is necessary in order to interpret (or to forbid) formulas of the form $f^\omega \cdot g$, $f^\omega \cdot g^\omega$, $(f^\omega)^*$ etc.

To enable a uniform representation, we have decided to adopt a more modest interpretation of f^ω, as follows. We shall consider f^ω rather as a test, true in states (i.e., paths of length 0) from which it is possible to repeatedly carry out computations of f infinitely often. The advantage of using this interpretation is that even though paths associated with formulas are finite, and hence all CPL operators are applicable and retain their usual meaning, it is still possible to make assertions concerning infinite computations.

Let us now define ωCPL. As preparation, we need a definition.

Definition 16. An *ω-path over a set S is an infinite sequence of elements of S. For a set $\mathscr{P}$ of* finite *paths, let $\mathscr{P}^\omega = \{p_1 \cdot p_2 \cdot p_3 \cdots \mid \forall i \geq 1,\ p_i \in \mathscr{P}\}$. That is, $\mathscr{P}^\omega$ is the set of finite and infinite paths obtained by repeatedly fusing (finite) paths from $\mathscr{P}$ infinitely often.*

The syntax is such that ωCPL contains all CPL formulas, and in addition if f and g are ωCPL formulas, then so are $(\neg f)$, (f^*), (f^ω), $(f \cdot g)$, $(f \cup g)$ and $(f \cap g)$. As for semantics, ωCPL is interpreted over the same models as CPL. Given a model M and an ωCPL formula f, $\rho_M(f)$ is defined exactly as in CPL with the addition of the clause:

$$\rho_M(f^\omega) = \mathit{first}((\rho_M(f))^\omega).$$

ωCPL can be considered to be a 'path version' of RPDL. Indeed, we can extend the embedding of PDL in CPL to an embedding of RPDL in ωCPL by: $(\mathit{repeat}(\beta))' = (\beta')^\omega$. Thus, ωCPL's expressive power is at least as that of RPDL, which is known to be high (for example it exceeds that of CTL* [2]).

Proving that ωCPL is elementary decidable is done by reducing its satisfiability problem to that of ARPDL (the automata version of PDL+*repeat*). Here, we omit the details, and only mention that this reduction costs at most an exponential in added size. Thus, using the fact that ARPDL is decidable in EXPTIME [3], we have

Theorem 17. *Satisfiability of* ωCPL *formulas with* ATF $\subseteq \{a_1, \ldots, a_n\}$ *can be decided in* 2EXPTIME .

References

[1] R. Danecki, Nondeterministic propositional dynamic logic with intersection is decidable, 5th Symp. on Fundamentals of Computation Theory, Lecture Notes in Computer Science, vol. 208, Springer, Berlin 1984, pp. 34–53.

[2] E.A. Emerson, in: J. Van Leeuwen, (Ed.), Handbook of Theoretical Computer Science, vol. B, Elsevier, Amsterdam, 1990, pp. 996–1072.

[3] E.A. Emerson, C.S. Jutla, The complexity of tree automata and logics of programs, Proc. 29th IEEE Symp. Found. Comput. Sci., 1988, pp. 328–337.

[4] M.Fürer, The complexity of the inequivalence problem for regular expressions with intersection, Proc. 7th Int. Colloq. on Automata, Languages, and Programming, Lecture Notes in Computer Science, vol. 85, Springer, Berlin, 1980, pp. 234–245.

[5] M.J. Fischer, R.E. Ladner, Propositional dynamic logic of regular programs, J. Comput. Systems Sci. 18 (1979) 194–211.

[6] D. Harel, D. Kozen, R. Parikh, Process logic: expressiveness, decidability, completeness, J. Comput. Systems Sci. 25 (1982) 144–170.

[7] D. Harel, D. Peleg, Process logic with regular formulas, Theoret. Comput. Sci. 38 (1985) 307–322.

[8] D. Harel, R. Sherman, Looping vs. repeating in dynamic logic, Inform. Control (1982) 175–192.

[9] D. Harel, R. Sherman, Propositional dynamic logic of flowcharts, Inform. Control 64 (1985) 119–135.

[10] J.Y. Halpern, Deterministic process logic is elementary, Proc. 23rd IEEE Found. Comput. Sci., 1982, pp. 204–216.

[11] D. Kozen, R. Parikh, An elementary proof of the completeness of PDL, Theoret. Comput. Sci. 14(1) (1981) 113–118.

[12] R. Milner, A Calculus of Communicating Systems, Lecture Notes in Computer Science, Springer, Berlin, vol. 92, 1980.

[13] A.J. Mayer, L.J. Stockmeyer, The complexity of word problems – this time with interleaving, Inform. and Comp. 115(2) (1994) 293–311.

[14] D. Peleg, Regular process logics, M.Sc. Thesis, Bar-Ilan Univ., Ramat Gan, Israel, 1982 (in Hebrew).

[15] A. Pnueli, The temporal logic of programs, Proc. 18th IEEE Symp. Found. Comput. Sci., 1977, pp. 46–57.

[16] V.R. Pratt, Semantical considerations on Floyd–Hoare Logic, Proc. 17th IEEE Symp. Found. Comput. Sci., 1976, pp.109–121.
[17] R. Sherman, A. Pnueli, D. Harel, Is the interesting part of process logic uninteresting?: a translation from PL to PDL, SIAM J. Comput. 13 (1984) 825–839.
[18] M. Vardi, P. Wolper, Yet another prcess logic, Proc. Symp. on Logics of Programs, Lecture Notes in Computer Science, vol. 164, Springer, Berlin, 1983, pp. 501–512.