Theoretical Computer Science 55 (1987) 1-45
North-Holland

CONCURRENT PROGRAM SCHEMES AND THEIR LOGICS

David PELEG*

Department of Applied Mathematics, The Weizmann Institute of Science, Rehovot 76100, Israel

Communicated by E. Engeler
Received June 1986
Revised June 1987

Abstract. We define and investigate several classes of concurrent program schemes, including goto schemes and two versions of structured schemes, based on extensions of the regular expressions to trees. The schemes are studied on the first-order, Boolean-variable and propositional levels. We also define and study the dynamic logics based on these classes of schemes, including issues of decidability and axiomatization.

Contents

* Present affiliation: Department of Computer Science, Stanford University, Stanford, CA 94305, U.S.A.

1. Introduction

The theory of programs deals extensively with the issue of concurrency. Various formalisms appear in the literature for defining and describing parallel computations and programs. In particular, ***concurrent program schemata*** were introduced in [13] and then appeared (in different forms) in several other works. However, they have not received a widely accepted ***structured*** format analogous to the sequential ***while*** schemes, and issues of semantics and logic are still under extensive research, in numerous different models.

Most models for concurrent computation assume a situation in which there is some kind of sharing in resources, e.g., memory. This corresponds to the concept of multiprocessing in a single-processor environment, in which when two or more processes work in parallel, they may affect the same memory locations or variables.

In this paper we follow a different model of concurrency based on the notion of and/or-trees, and concerning essentially separate, independent processes. This model corresponds, for instance, to a network of processors. The model gives rise to the concurrent goto schemes appearing in [2, 14]. These programs may contain such commands as goto l_1 or l_2, facilitating nondeterministic choice, as well as commands like goto l_1 and l_2, causing a split into two parallel independent branches. This naturally reflects in the semantics of a concurrent program. Consider, for instance, the two schemes described in Fig. 1, interpreted over the natural numbers. In a usual shared environment, the interpretations of the two programs (a) and (b) are equivalent, i.e., their input/output relation is $\{([i,j],[i+1,j+1]) \mid i,j \geqslant 0\}$, where $[i,j]$ represents the initial values of x and y. Thus, there is a single set of variables, affected by both branches. However, we view the concurrent program (a) differently, and give it a semantics as follows: upon splitting into two branches, each of the new processes receives a private copy of the variables, and proceeds on its own. Therefore, the left process changes only x, while the right one changes only y.

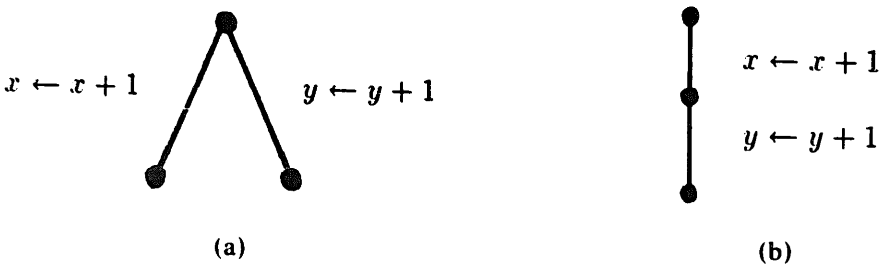

Fig. 1.

Still, our semantics differs from that of [2, 14] and is closer to that of the language IND of [9]. The overall semantics given therein for the scheme as a whole interprets it as an essentially sequential program, whose 'start-end' relation leads from a single state to a single state. This is done by taking the following view. The processes are assumed to be running independently with different and unrelated speeds. When one of the processes terminates, its final state, i.e., the result of its computation, is taken as the result of the whole program. All the other processes are assumed to abort once that first process halted. Thus parallelism is interpreted as essentially another form of nondeterminism. In contrast, we define the semantics so that a program may lead us from a single state to a *set* of states, thus retaining the parallel nature of the run. For example, the semantics of the above program (a) changes in the interpretation of [2] to be

$$\{([i,j],[i+1,j]),([i,j],[i,j+1])\mid i,j\geqslant 0\},$$

while we interpret it as $\{([i,j],\{[i+1,j],[i,j+1]\})\mid i,j\geqslant 0\}$.

An important consequence is that in our formalism, *all* processes are required to halt for the program to converge. On the other hand, in the version of [2, 14], a program may halt successfully even if some branch of it contains a subprogram with an infinite loop in it; as long as *one* process of the program may halt, its interpretation is nonempty. In this respect, our formalism extends the sequential schemes in the same sense that alternating TM's [3] extend the usual ones.

The next logical step is to define a *structured* version of concurrent program schemes. The structured description of while programs was found to have several advantages, both in practical aspects such as better programming and in theoretical aspects such as simpler and better understood semantics, and cleaner methods for verification and analysis. In particular, tools like *dynamic logic*, aimed to enable reasoning about programs and their properties, owe their elegance and simple axiomatization in part to the regular structure of their programs.

We propose two versions of structured schemes. Both approaches are based on extensions of regular expressions to trees. The one is based on a simple extension of regular expressions using a new *concurrency connective* $\cap$. This connective is the dual of the union connective $\cup$ in exactly the same sense as the duality between the 'and' and 'or' steps of an ATM [3]. This version, called the *concurrent regular schemata*, was defined and studied in [18] in the framework of dynamic logic. The second version, which we call *sticky schemata*, is based on regular expressions on trees (cf. [6, 21]). These expressions are based on a concatenation and star operation, in which one uses labels to mark leaves of the tree, and may 'stick' new subtrees to these leaves in a controlled fashion, according to the labels.

The resulting classes of goto and structured schemes are denoted in this paper by **c-goto, c-reg** and **sticky**. The 'c' prefix stands for *concurrent*, and is used to distinguish these classes from the corresponding classes of *sequential* schemes, denoted simply **reg** and **goto**.

The classes of schemes are defined and studied on three levels. The basic one is the propositional level, where atomic programs are left unspecified. This level is meant to provide an abstraction of the discussed notions, constructs and mechanisms, so as to enable an analysis of their fundamental properties and behavior (cf. [7, Section 1.3]). Next comes the intermediate Boolean-variable level, where Boolean variables are available in addition to the basic propositional environment. Another, equivalent, view would be to allow a program to change the truth value of propositional variables (cf. [22]). Finally, on the first-order level, the environment is that of a first-order structure, and assignments are taken to be the basic operations. The three levels are denoted by P, B and Q, respectively.

A second parameter in the classification of schemes is the formalism used for *tests* in the schemes. Two particular formalisms we use are the propositional calculus, denoted PC, and the quantifier-free subset of first-order logic with equality, denoted QF. Thus, for instance, **sticky**(P, PC) denotes the class of propositional sticky schemes with propositional tests.

An additional feature we consider is the *kill* command, which enables a program to terminate a process and discard it completely, i.e., drop its end state from the set of final states of the whole computation. Throughout, we consider also versions of schemes based on this additional command. We refer to these schemes as *K-schemes*, and denote the classes with the superscript K, e.g., $\mathbf{c\text{-}reg}^{\mathrm{K}}$.

Our results regarding the schemes are as follows. On the Boolean-variable and first-order levels, all three types of concurrent schemes are shown to be equal in expressiveness. (This holds for both deterministic and nondeterministic schemes, although we consider almost exclusively only nondeterministic schemes.) On the propositional level, the family of **c-goto** schemes is shown to be equivalent to the **sticky** schemes, while **c-reg** schemes are strictly weaker.

We also compare propositional schemes to Boolean ones, similar to the comparison carried out in [1] for while schemes in the framework of PDL, and show that propositional **c-goto** schemes are as powerful as Boolean schemes. This parallels the result of [1] for sequential while schemes (or actually for the corresponding dynamic logics).

We consider also the dynamic logics obtained by taking classes of concurrent schemes as the underlying sets of programs. On the Boolean-variable (respectively first-order) level since the program schemes are equal, the three resulting versions of BDL (respectively QDL) turn out to be equivalent as well. On the propositional level we have two classes of logics (versions of PDL). The first class contains the logic obtained by considering **c-reg** schemes, denoted **c-reg**-PDL. This logic was defined and discussed in [18], and was given a complete axiomatization and an elementary decision procedure. Its main advantages are its simple semantics and its elegant axiom system.

It seems, however, that the logics of the second class, i.e., the one based on **c-goto** schemes (**c-goto**-PDL), and particularly the one utilizing **sticky** schemes (**sticky**-PDL), possess several properties indicating that they might very well be the 'right' logics, on the propositional level, for this model of concurrency. For one thing, **sticky**-PDL is equivalent to **c-goto**-PDL, which parallels the equivalence between the versions of PDL based on sequential **goto** and **reg** schemes. In fact, we conjecture that the two classes are different, i.e., that **c-reg**-PDL is strictly less expressive than these logics.

Also, Abrahamson's result [1], mentioned previously, has a natural 'concurrent' analogue in **sticky-BDL**, i.e., one can show that **sticky-BDL** = **sticky-PDL**.

Thirdly, on the first-order level, **c-reg-QDL** (with random assignments) is shown in [18] to be equivalent to the continuous μ-calculus, CQ_μ. The same holds also for **sticky-QDL**. Analogously, on the propositional level, **sticky-PDL** is shown here to be equivalent to the continuous propositional μ-calculus of [12], CL_μ.

Our results on expressiveness of the program classes and logics are summarized in Figs. 2 and 3 respectively.

As an analogue to the issue of axiomatization in [18], we present here a complete axiom system for **sticky-PDL** and provide it with a nondeterministic exponential time decision procedure for validity. This is in no contradiction with the double

(1) Equivalences:

$$\begin{array}{ccccc}
\textbf{c-reg}(P,L_1) & \leq_{(*)} & \textbf{c-goto}(P,L_1) & = & \textbf{sticky}(P,L_1) \\
 & & \|_l \ (**) & & \\
\textbf{c-reg}(B,L_2) & = & \textbf{c-goto}(B,L_2) & = & \textbf{sticky}(B,L_2) \\
 & & & & \\
\textbf{c-reg}(Q,L_3) & =_{(***)} & \textbf{c-goto}(Q,L_3) & = & \textbf{sticky}(Q,L_3)
\end{array}$$

(*) for L_1 = PC we have strict inequality.
(**) assuming $L_1 =_l L_2$ (cf. Section 8.1).
(***) assuming L_1 contains at least QF.

(2) Inequivalences:

$$\begin{array}{c}
\textbf{reg}(X,L) \\
\wedge \\
\textbf{c-reg}(X,L) \\
\wedge \\
\textbf{c-reg}^k(X,L)
\end{array}$$

These results apply to all three levels, and hold with either (P,PC), (B,PC) or (Q,QF) substituted for (X,L).

Fig. 2. Expressiveness relationships between scheme classes.

(1) On the propositional level:

$$\begin{tikzcd}
 & & & & \textbf{PDL} \arrow[d, phantom, "\wedge"] \\
\text{c-reg}^K\text{-PDL} & = & & & \textbf{c-reg-PDL} \arrow[d, phantom, "/\wedge"] \\
\text{continuous } \mu\text{-calculus} & = & \textbf{sticky-PDL} & = & \textbf{c-goto-PDL}
\end{tikzcd}$$

(2) On the Boolean-variable level:

$$\begin{tikzcd}
 & & & & \textbf{BDL} \arrow[d, phantom, "/\wedge"] & & \\
\text{c-reg}^K\text{-BDL} & = & \text{c-reg-BDL} & = & \textbf{sticky-BDL} \arrow[d, phantom, "\|/"] & = & \textbf{c-goto-BDL} \\
 & & & & \textbf{sticky-PDL} & &
\end{tikzcd}$$

(3) On the first-order level:

$$\begin{tikzcd}
 & & & & \text{QDL} \arrow[d, phantom, "/\wedge"] & & \\
\text{c-reg}^K\text{-QDL} & = & \text{c-reg-QDL} & = & \text{sticky-QDL} & = & \text{c-goto-QDL}
\end{tikzcd}$$

Fig. 3. Expressiveness relationships between logics.

exponential lower bound set by Abrahamson for **reg-BDL** since the translation from **reg-BDL** (or **c-reg-BDL**) into **sticky-PDL** itself causes an exponential blow-up in the size of the formula.

In [19] we considered the introduction of some communication mechanisms into **c-reg-PDL**. Strong mechanisms such as *channels* or *shared variables* were shown to considerably increase the computational power of the schemes. We also considered a weak *message* mechanism, and conjectured that the resulting language is more expressive than **c-reg-PDL**. Similar mechanisms can also be incorporated in the stronger types of program schemes studied in this paper, with similar behavior w.r.t. expressiveness and decidability.

The rest of the paper is arranged as follows. Section 2 reviews notions from the theory of finite tree languages and automata. The major part of this paper concerns the propositional level: Section 3 reviews sequential schemes, and in Section 4 we define several classes of concurrent schemes. Section 5 contains some relationships between these classes. Section 6 contains definitions and comparisons of the dynamic logics based on concurrent classes of schemes, and Section 7 concerns issues of validity and axiomatization for these logics. Finally, in Sections 8 and 9 we consider the Boolean-variable and first-order levels respectively.

2. Tree languages

We need some concepts from the theory of finite tree languages and automata, adapted for our setting. In particular, we consider only binary trees. (Throughout the sequel, the term *binary tree* refers to a tree with 0, 1 or 2 sons for every node.) We also define a slightly different system of regular expressions than that in the literature. A good coverage of the general theory can be found in, e.g., [6].

2.1. Tree terms

The alphabet $\Sigma = \Sigma_0 \cup \Sigma_1 \cup \Sigma_2$ consists of the following: Σ_2 contains a single dyadic function symbol $\cap$, Σ_1 contains monadic functions d_i, $1 \leq i \leq n$, and Σ_0 contains constants $\$_i$, $1 \leq i \leq k$, for some $k, n \geq 1$.

The collection of *tree terms* (or simply terms) over Σ is the minimal collection T_Σ of words over $\Sigma \cup \{(,)\}$ such that

(1) $\Sigma_0 \subseteq T_\Sigma$;

(2) if $d \in \Sigma_1$ and $t \in T_\Sigma$, then $d(t) \in T_\Sigma$; and

(3) if $t_1, t_2 \in T_\Sigma$, then $(t_1) \cap (t_2) \in T_\Sigma$.

Figs. 4(a)-(c) describe the trees represented by the terms \$, $d(t)$ and $(t_1) \cap (t_2)$ respectively, given that t, t_1 and t_2 are represented by trees T, T_1 and T_2, respectively. A *tree language L* is simply a subset of T_Σ, i.e., a collection of tree terms over the alphabet Σ.

2.2. Regular tree grammars

We will be needing the formalism of *regular tree grammars* (cf. [6]). A regular tree grammar over an alphabet Σ as in the previous section is a context-free grammar

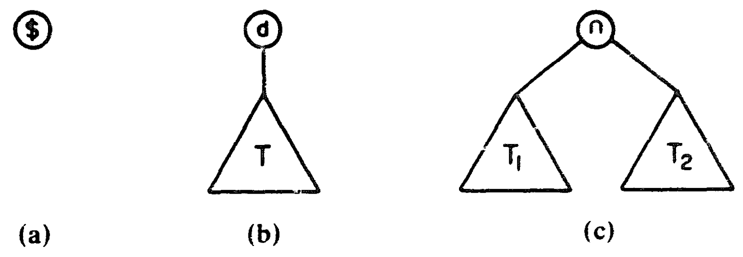

Fig. 4.

$G = \langle V, P, Z\rangle$ (cf. [11]), where V is the collection of variables, Z is a collection of start symbols ($Z \subseteq V$), and P is a collection of production rules of the following forms:

$$A \to (B) \cap (C), \qquad A \to d(B),$$
$$A \to \$, \qquad A \to B,$$

where $A, B, C \in V$, $d \in \Sigma_1$ and $\$ \in \Sigma_0$.

The grammar operates over $\Sigma \cup \{(,)\}$ in the usual way so that the derived words are terms from T_Σ. A tree language $L \subseteq T_\Sigma$ is *generable* iff there exists a grammar G generating precisely the tree terms in L.

2.3. *Tree-regular languages*

Let us first define the operations of product and closure on tree languages. Given an alphabet Σ as above, and two languages $U, V \subseteq T_\Sigma$, define, for every $\$_i \in \Sigma_0$,

$$U \cdot^i V = \{t \in T_\Sigma \mid \exists t' \in U \ (t \text{ is obtained by replacing every occurrence of } \$_i \text{ in } t' \text{ by some term of } V)\}.$$

From a 'tree-point-of-view', we start with a tree T' of U, and replace every leaf labeled $\$_i$ with some tree taken from V.

Similarly, for every $\$_i \in \Sigma_0$, define a closure operation by

$$U^{*_i} = \min V(\{\$_i\} \in V \text{ and } \forall t, t_1, t_2 ((t_1 \in V, t_2 \in U, t \text{ is obtained by replacing one occurrence of } \$_i \text{ in } t_1 \text{ by } t_2) \Rightarrow t \in V)),$$

where the minimum is taken w.r.t. the usual subset ordering.

An equivalent definition sets $U^{*_i} = \bigcup_{0 \leq j} U_j$ where $U_0 = \{\$_i\}$ and $U_{j+1} = U_j \cup U \cdot^i U_j$.

We define also the following two operations:

$$U \cap V = \{(t_1) \cap (t_2) \mid t_1 \in U, t_2 \in V\},$$
$$d(U) = \{d(t) \mid t \in U\} \quad \text{for every } d \in \Sigma_1.$$

We now give an inductive definition for the set of ***Σ-tree-regular expressions*** (over an alphabet Σ).

(1) $\$_i$ is a Σ-tree-regular expression for every $\$_i \in \Sigma_0$;

(2) if α_1, α_2 are Σ-tree-regular expressions, then so are $(\alpha_1) \cap (\alpha_2)$, $(\alpha_1) \cup (\alpha_2)$, $(\alpha_1) \cdot^i (\alpha_2)$, $(\alpha_1)^{*_i}$ and $d(\alpha_1)$, for every $\$_i \in \Sigma_0$ and $d \in \Sigma_1$.

The *language* L_α represented by each Σ-tree-regular expression α is defined as usual by the corresponding operations, with $\$_i$ representing the language $\{\$_i\}$ for every $\$_i \in \Sigma_0$ as a base step, and $(\alpha_1) \cup (\alpha_2)$ representing $L_{\alpha_1} \cup L_{\alpha_2}$.

A tree language $L \subseteq T_\Sigma$ is ***Σ-tree-regular*** iff there is a Σ-tree-regular expression representing it. L is ***tree-regular*** iff there is an alphabet $\Sigma' = \Sigma \cup \Sigma'_0$ such that L is Σ'-tree-regular (i.e., it might take some extra constants to give it a regular description).

2.1. Theorem (Gecseg and Steinby [6, Theorems 3.6 and 5.8]). *A tree language is generable iff it is tree-regular.*

2.2. Note. While the '$\cup$' symbol is used in its standard set-theoretic sense, the '$\cap$' symbol represents here an operation symbol occurring in the syntax of tree terms and should not be confused with either the set-theoretic intersection operator, or the 'relation-theoretic' one appearing in [7, Section 2.5.5].

2.3. Note. The conventional definition of the Σ-tree-regular sets is slightly different, namely the closure of all *finite* subsets of T_Σ under $U \cdot^i V$, U^{*_i} and $U \cup V$. It is easy to see that the two resulting definitions of tree-regularity coincide.

2.4. Note. A third equivalent representation for the tree-regular languages is obtained by means of *tree automata.* There are several classes of deterministic and nondeterministic automata which recognize precisely these languages (cf. [6]). In fact, one may also define classes of *alternating* tree automata for this and other families of languages [20]. In the sequel, we will use sets of tree terms for representing alternating (and/or) program schemes. However, the notion of alternation in our classes of program schemes is fully captured by the $\cup$ and $\cap$ operations discussed above, and the syntax corresponds in a straightforward way to that of tree grammars and regular expressions. Therefore, we do not need to introduce any class of automata.

3. Sequential program schemes

In this section we briefly survey some classes of conventional sequential program schemes. The schemes are presented on the propositional level, which gives a high level of abstraction, by referring to the atomic operations as unspecified; all that we know of a program a is that it takes us from some states in our state-space to other states, according to its semantic interpretation. This approach appears, for example, in [4], where the formalism of propositional dynamic logic (PDL) is proposed for a propositional analysis of program schemes.

We begin with a general description, relying mainly on intuition as to the meaning of the schemes, and then give a precise definition for the semantics of the schemes. A general survey of (first-order) schemes can be found in [5].

3.1. Goto schemes

A *goto scheme* is a linear representation for a flowchart, and is one of the most widely accepted formalisms of describing a simple sequential program. Formally, a *propositional nondeterministic* goto scheme is a program composed of a sequence of labeled commands of the following types:

(ATOMIC) $l: a_i$,

(TEST) $l: P?$

(N-GOTO) l: goto l' or l'',

where P is a formula in a given logic L, interpretable over models as described below, and a_i is an unspecified atomic command, taken from a collection AP of atomic programs.

Throughout the paper we adopt the following notation for classes of schemes. Each basic control structure of schemes is given a name. In addition, two parameters are to be fixed, namely, the *level*, or the types of *atomic operations* allowed and the *logic* used for tests. For instance, the class just described will be denoted **goto**(P, L), meaning that it is based on sequential goto schemes, defined in the propositional level (i.e., uses atomic unspecified operations), and allows tests from a (propositional) logic L.

In the literature, schemes usually employ tests within the if-goto command, l: if P then goto l' else goto l''. However, it is clear that the two mechanisms are equivalent in the presence of nondeterminism. (We interpret an infinite loop as an aborted run so that a test $l: P?$ can be simulated by l: if P goto $l+1$ else goto l.) This particular choice was made for reasons of compatibility with other classes of schemes to be described later.

3.2. Regular schemes

A *propositional nondeterministic regular scheme* is defined as a regular expression over an alphabet Σ_{PR} consisting of the set of atomic programs AP and the tests $\{P? \mid P \in L\}$. (The alphabet Σ_{PR} is not necessarily finite, but every scheme defines a finite subset of it.) Call this class **reg**(P, L), for any appropriate logic L. The symbol Σ_{PR} is used in the sequel to denote the set of basic propositional schemes, as defined above. We use the symbol ";" for concatenation, to follow standard notation.

Again, it is conventional to consider the class of while schemes, defined in a slightly different (but expressively equivalent) way, as the inductive closure of assignment commands, viewed as atomic steps, under the constructs $\alpha;\beta$ and "if P then α else β" and "while P do α", where α, β are schemes and P is a test as above. (Nondeterminism may be added by the construct "α or β".) However, the form based on regular expressions is useful in providing a clear representation and suggesting connections with goto schemes and direct extensions to concurrent schemes. For instance, the fact that **reg**(P, L) and **goto**(P, L) are equivalent in computational power (assuming semantics as defined in the next paragraph for both classes) is easily derived using standard techniques from automata theory.

3.3. Semantics of sequential program schemes

A model for our schemes is a triple $\mathcal{M} = \langle S, \pi, \rho \rangle$, where S is the state space, $\pi : \mathrm{L} \to 2^S$ is the interpretation function for formulas of L (i.e., for every $P \in \mathrm{L}$, $\pi(P)$ is the set of states satisfying P), and $\rho : \mathrm{AP} \to 2^{S \times S}$ is the transition relation of atomic programs: for every $a \in \mathrm{AP}$ and $s, s' \in S$, $(s, s') \in \rho(a)$ means that a can be executed in s to reach s'.

The interpretation ρ is extended to arbitrary **reg**(P, L) schemes as follows:

$$\rho(P?) = \{(s, s) \mid s \in \pi(P)\},$$

$$\rho(\alpha \cup \beta) = \rho(\alpha) \cup \rho(\beta),$$

$$\rho(\alpha;\beta) = \{(s, s') \mid \exists s''((s, s'') \in \rho(\alpha) \wedge (s'', s') \in \rho(\beta))\},$$

$$\rho(\alpha^*) = \{(s, s') \mid \exists k \geq 0, \exists s_0, \ldots, s_k, \forall 0 \leq i < k((s_i, s_{i+1}) \in \rho(\alpha))\}.$$

The situation is slightly more involved for **goto**(P, L) schemes since in order to identify the end-state of a computation of a goto scheme, we have to characterize the whole sequence of intermediate states. For an atomic program or a test, the definition of ρ is just as before. For an arbitrary **goto**(P, L) scheme $\alpha = (1:\gamma_1, \ldots, m:\gamma_m)$, we first define a *computation sequence* of α as a sequence $((l_1, s_1), (l_2, s_2), \ldots, (l_k, s_k))$ with the following properties:

(1) $l_1 = 1$ and $l_k = m+1$;

(2) For every $1 \leq i \leq k-1$, exactly one of the following holds:

  (a) γ_{l_i} is an atomic a or a test $P?$, $l_{i+1} = l_i + 1$ and $(s_i, s_{i+1}) \in \rho(\gamma_{l_i})$, or

  (b) γ_{l_i} is goto l' or l'', $l_{i+1} \in \{l', l''\}$ and $s_{i+1} = s_i$.

Now, $\rho(\alpha)$ contains a pair (s, s') iff there exists a computation sequence of α as described such that $s = s_1$ and $s' = s_k$.

4. Concurrent program schemes

In this section we define the different versions of (propositional) concurrent program schemes to be discussed later. The first version is that of *concurrent goto schemes*. Then we give some versions of *structured* concurrent program schemes. We define two types of structured schemes. The first is the class of schemes with *sticky labels*, or simply *sticky schemes*, based on tree-regular expressions. The second is a subset of the first class, referred to as *concurrent regular schemes*, based on an extension of the sequential regular schemes given in Section 3.2.

4.1. Concurrent goto schemes

A *propositional concurrent goto scheme* is a sequence of commands of the types described in Section 3.1, i.e., (ATOMIC), (TEST), (N-GOTO), and, in addition, commands of the type

(PAR) l: goto l' and l''.

The class of such schemes is called in the sequel **c-goto**(P, L) for any appropriate logic L.

Informally, when a process reaches a command of type (PAR), it splits into two parallel processes which are in identical states. (Giving this 'real life' interpretation we may say that the memory locations held by the original process are duplicated, and each of the two new processes receives an identical private copy.) Now, one of the processes proceeds to execute the command labeled l', and the other proceeds to l'', and from now on these two processes remain separate and independent, with no connection whatsoever.

The semantics of a concurrent scheme is still based on a model as described in Section 3.3, i.e., $\mathcal{M} = \langle S, \pi, \rho\rangle$, where $\rho(a) \subseteq S \times S$ for an atomic program a. Thus, *atomic* operations remain essentially sequential. However, for an arbitrary scheme α, we assign a concurrent interpretation $\rho(\alpha) \subseteq S \times M(S)$, where $M(S)$ is the collection of multisets of elements of S. Thus $(s, V) \in \rho(\alpha)$ for $s \in S$, $\text{set}(V) \subseteq S$ (where set(V) denotes the set containing precisely the elements of V) means that α can be executed from state s to reach precisely all states of V. In this view, a pair $(s, s') \in \rho(a)$ for an atomic program a is handled as $(s, \{s'\})$. (In [18] we followed the game logic of [16] and defined a version in which we let atomic programs have a (possibly) concurrent interpretation too.)

The relation ρ describes the 'input-output' behavior of a program or, more precisely, its 'start-end' relation. However, in order to define it we have to describe the whole computation. This description can no longer consist of a sequence; rather, it has to take the form of a tree.

Let us first describe a simple, direct semantics for the **c-goto** schemes, similar to that given in Section 3.3 for the **goto** schemes.

A *trace* of a concurrent goto scheme $\alpha = (1 : \gamma_1, \ldots, m : \gamma_m)$ is a binary tree with a set $\{1, \ldots, k\}$ of vertices, where 1 is the root, each vertex i is labeled by a pair (l_i, s_i), where l_i is an integer, $1 \leq l_i \leq m+1$, and s_i is a state of $\mathcal{M}$, and the following properties hold:

(1) $l_1 = 1$,

(2) for every leaf i, $l_i = m+1$,

(3) for every internal (non-leaf) node i exactly one of the following hold:

(a) γ_{l_i} an atomic program a or a test $P?$, i has a single son j, $l_j = l_i + 1$ and $(s_i, s_j) \in \rho(\gamma_{l_i})$,

(b) γ_{l_i} is goto l' or l'', i has a single son j, $l_j \in \{l', l''\}$ and $s_j = s_i$,

(c) γ_{l_i} is goto l' and l'', i has two sons j_1, j_2, $l_{j_1} = l'$, $l_{j_2} = l''$ and $s_{j_1} = s_{j_2} = s$.

Now, $\rho(\alpha)$ contains a pair (s, V) for $s \in S$, $\text{set}(V) \subseteq S$ iff there exists a trace of α as described such that $s_1 = s$ and the multiset of all markings on its leaves is precisely $\{(m+1, s') \mid s' \in V\}$.

Note. The usual notion of a computation tree (for a *sequential* nondeterministic program) refers to a tree describing different *possibilities* of a run. An *actual* run is described by a single path from the root to one of the leaves. Here, the trace describes an actual, deterministic run, i.e., after making all nondeterministic choices.

While this definition of the semantics directly captures the behavior of runs of the program and traces its control changes in full, for later purposes we need also a slightly different semantical definition, based on the concepts of tree terms and tree grammars.

It is well-known that the semantics of a (nondeterministic) sequential program α can be defined on the basis of a set $\mathcal{T}(\alpha)$ of *deterministic* sequence programs, or seq's (cf. [15]) describing the possible runs of α so that $\rho(\alpha)=\bigcup_{\beta\in\mathcal{T}(\alpha)}\rho(\beta)$. Analogously, the semantics of a (nondeterministic) concurrent program α can be based on a collection of deterministic *tree programs*, or trec's (cf. [18]).

A trec is actually a tree term from T_Σ for an alphabet Σ whose Σ_1 component is Σ_{PR} (defined in Section 3.2), and whose Σ_0 component includes constants $\$_i$ for $i\geq 1$. (In fact, the **c-goto** schemes can be defined using a *single* constant $\$$; the more general trec's are required for the **sticky** schemes, to be defined in Section 4.2.)

In order to define the semantics of trec's we first introduce the notion of *labeled sets*. A labeled set is a pair $(i:U_i)$ where $i\geq 1$ and U_i is a multiset of states such that $\mathrm{set}(U_i)\subseteq S$. Intuitively, $(i:U_i)$ describes the set of states of the trace which are labeled by $\$_i$. A tuple of labeled sets is a set $U=\{(i_j:U_{i_j})\mid 1\leq j\leq k\}$ where $1\leq i_1<\cdots<i_k$.

We write $i\in U$ as a shorthand for $\exists U_i((i:U_i)\in U)$. Similarly, we may write $\$_i\in\alpha$ meaning "$\$_i$ occurs in α". For every trec α, $\rho^{\$}(\alpha)$ will be defined as a collection of *semantic pairs* (s, U) where $s\in S$ and U is a tuple of labeled sets with a labeled set $(i:U_i)$ for every label $\$_i$ in α (if U does not contain $(i:U_i)$ for some $\$_i\in\alpha$, then it is interpreted as containing $(i:\emptyset)$).

Given $\mathcal{M}=\langle S,\pi,\rho\rangle$ we first define $\rho^{\$}(\alpha)$ inductively as follows:

$$\rho^{\$}(\$_i)=\{(s,\{(i:\{s\})\})\mid s\in S\},$$

$$\rho^{\$}(a(\alpha))=\{(s,U)\mid\exists s'((s,s')\in\rho(a)\wedge(s',U)\in\rho^{\$}(\alpha))\},$$

$$\rho^{\$}(A?(\alpha))=\{(s,U)\mid(s,U)\in\rho^{\$}(\alpha)\wedge s\in\pi(A)\},$$

$$\rho^{\$}(\alpha\cap\beta)=\{(s,U)\mid\exists V,W((s,V)\in\rho^{\$}(\alpha)\wedge(s,W)\in\rho^{\$}(\beta)\wedge U=V\uplus W)\},$$

where the (multiset) union $\uplus$ is taken componentwise.

Finally, for the whole trec α, we combine all separate subsets of leaves, and let

$$\rho(\alpha)=\Big\{(s,V)\mid\exists U\Big((s,U)\in\rho^{\$}(\alpha)$$
$$\wedge\, U=\{(i_j:U_{i_j})\mid 1\leq j\leq k\}\wedge V=\biguplus_{1\leq j\leq k}U_{i_j}\Big)\Big\}.$$

The **c-goto** schemes are given a semantics by regarding a concurrent goto scheme α as a regular tree grammar G_α over the alphabet Σ_{PR}.

The tree grammar G_α is obtained by taking the command labels to be variables, and translating the commands into production rules in the following manner: Any command $l:a$ (where $a\in\mathrm{AP}$) yields a production rule $l\to a(l+1)$. Similarly for a test $P?$. Any command l: goto l' or l'' yields $l\to l'\mid l''$ and any command l: goto l' and l'' yields $l\to(l')\cap(l'')$. Finally, we add the rule $(m+1)\to\$$, where α contains m commands.

Now we attach to α the tree language $\mathcal{T}(G_\alpha) \subseteq T_\Sigma$ generated by G_α, and set

$$\rho(\alpha) = \bigcup_{\beta \in \mathcal{T}(G_\alpha)} \rho(\beta).$$

4.1. Example. The and/or graph in Fig. 5 is a pictorial description of the **c-goto** scheme prog$_1$ described below (we allow ourselves a more compressed notation for control commands, for the sake of brevity).

```
prog₁
 1:goto 2 or 4 or 6
 2:a
 3:goto 1
 4:P?
 5:goto 14
 6:goto 7 or 9
 7:Q?
 8:goto 14
 9:goto 10 and 12
10:b
11:goto 1
12:c
13:goto 6.
```

The regular tree grammar G_1 obtained from prog$_1$ is listed below. A possible tree of this scheme is $\tau = b(P?(\$)) \cap c(Q?(\$))$. Fig. 6 gives a full description (a trace) of a possible run of prog$_1$ which corresponds to τ in a model in which $(s_1, s_2) \in \rho(b)$, $(s_1, s_3) \in \rho(c)$, $s_2 \in \pi(P)$ and $s_3 \in \pi(Q)$. Note that, in every run of the program, all leaves must satisfy P or Q.

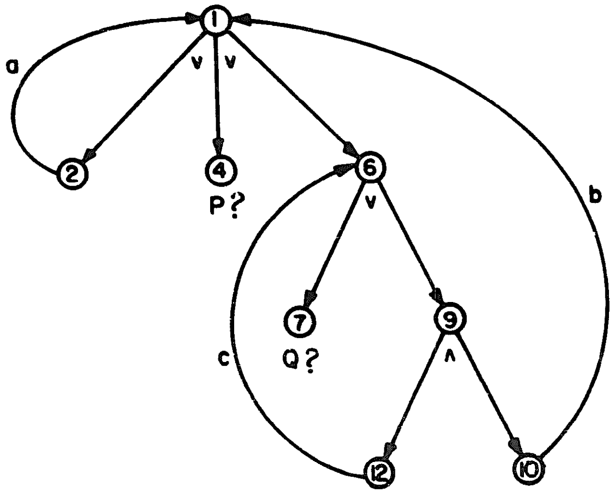

Fig. 5. The graph description of prog$_1$. Labeled arrows denote the execution of an atomic program plus transfer of control and unlabeled arrows denote transfer of control alone.

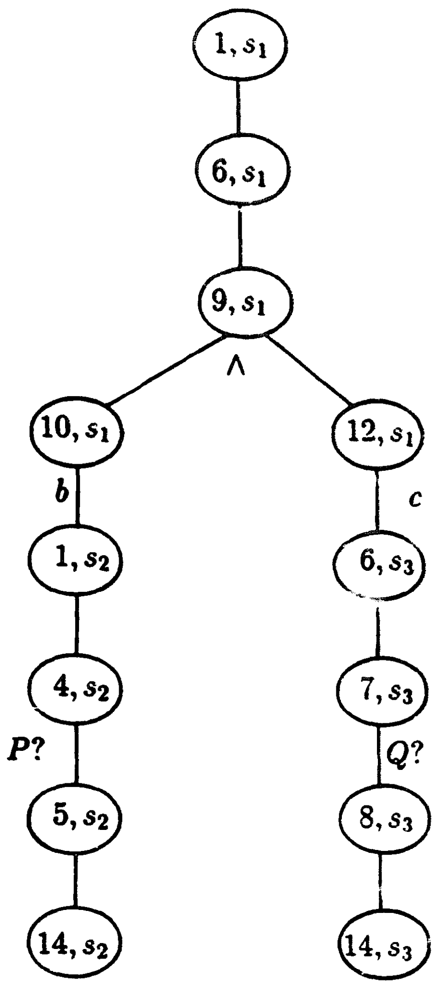

Fig. 6. A possible trace of prog_1 corresponding to τ.

The grammar G_1

$$\begin{array}{ll}
A_1 \to A_2 | A_4 | A_6, & A_8 \to A_{14}, \\
A_2 \to a(A_3), & A_9 \to (A_{10}) \cap (A_{12}), \\
A_3 \to A_1, & A_{10} \to b(A_{11}), \\
A_4 \to P?(A_5), & A_{11} \to A_1, \\
A_5 \to A_{14}, & A_{12} \to c(A_{13}), \\
A_6 \to A_7 | A_9, & A_{13} \to A_6, \\
A_7 \to Q?(A_8), & A_{14} \to \$.
\end{array}$$

4.2. Example. (*leaf counting* mod 2). Considering models in the form of full binary a/b trees, the scheme even_1, run from the root state of the model, halts successfully iff there is an even number of leaves satisfying P. (We assume the existence of a predicate leaf to be true in exactly the leaves of the tree.)

We say that a state s is *even* if the subtree rooted at s has an even number of leaves satisfying P, and similarly for odd states. The node labeled 1 in the program graph (Fig. 7) has to verify that the current state is even, while the node labeled 6 has to verify that the current state is odd. For instance, node 1 operates by checking

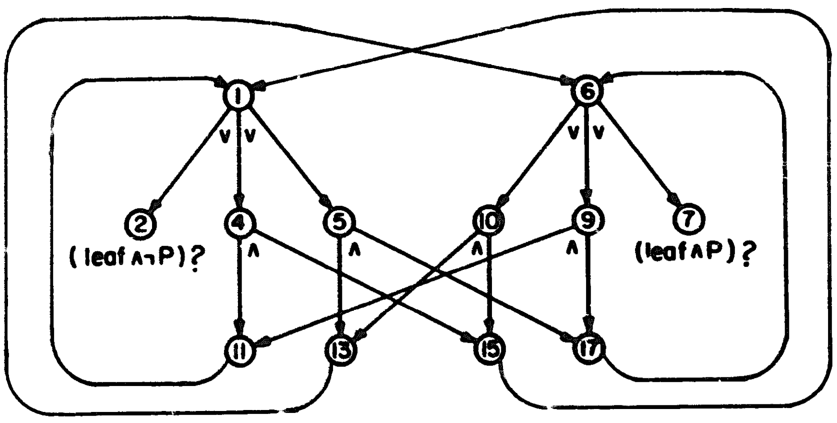

Fig. 7. The graph description of even_1.

whether the current state is a leaf satisfying $\neg P$ and if not, by splitting into two parallel processes which have to verify that its a and b children are both even or both odd.

```
even_1
 1: goto 2 or 4 or 5
 2: (leaf ∧ ¬P)?
 3: goto 19
 4: goto 11 and 15
 5: goto 13 and 17
 6: goto 7 or 9 or 10
 7: (leaf ∧ P)?
 8: goto 19
 9: goto 11 and 17
10: goto 13 and 15
11: a
12: goto 1
13: a
14: goto 6
15: b
16: goto 1
17: b
18: goto 6.
```

We will return to these examples in subsequent sections, and consider them in other types of schemes.

4.2. *Sticky schemes*

We now turn to defining a structured version of concurrent schemes, based on tree regular expressions. Compared to the class of sequential regular schemes **reg**, the main additional connective is the *concurrency* connective $\cap$. Informally, $\alpha \cap \beta$

is to be read as 'split into two parallel processes, one performing α and the other performing β'. Thus, here too, a specific run takes the form of a tree. We use *sticky labels* $\$_i$ to identify the end-points of any run of a scheme, and enable controlled concatenation, by means of the connectives $;^i$.

Formally, *propositional sticky schemes* are tree-regular expressions over an alphabet Σ whose Σ_1 component is Σ_{PR}, and whose Σ_0 component includes constants $\$_i$ for $i \geq 1$. We use $;^i$ instead of $\cdot^i$, to follow convention. Denote the resulting class of schemes by **sticky**(P, L), for any appropriate logic L.

A trace of a sticky scheme α is a binary tree with a set $V = \{1, \ldots, k\}$ of vertices, where 1 is the root, each vertex i is labeled by a state s_i and each of the leaves (and possibly some of the internal nodes) is labeled also by a sticky label, i.e., a letter of Σ_0 (different leaves are allowed to be labeled by different sticky labels, even when their states are the same). Each such tree corresponds to some run of α. The set of traces of a program α in a given model can be defined by induction on the structure of α. For instance, for a letter $\$_i$, a trace is a single node labeled by some state s and the label $\$_i$. For a program $a(\beta)$, traces are obtained by taking a trace T_β of β whose root is labeled s_2, where $(s_1, s_2) \in \rho(a)$, creating a new root labeled s_1 and attaching T_β by an edge to the new root. Traces for a subprogram $\beta \cap \gamma$ are constructed by starting with a trace T_β of β and a trace T_γ of γ whose roots are labeled with the same state s, introducing a new root node labeled s and connecting the roots of T_β and T_γ as children of the new root. Traces for a subprogram $\beta ;^i \gamma$ are obtained by starting with a trace T_β of β and attaching, to each leaf v labeled by a state s and a sticky label $\$_i$, some trace T_γ of γ whose root is labeled by the state s (by identifying the root of T_γ with v). The repetition connective in sticky schemes α^{*_i} is interpreted with a similar meaning.

We omit a more formal definition of the traces, since the semantics of **c-reg** schemes can again be defined directly, by interpreting them as tree languages, without having to define the whole trace.

For every sticky scheme α, let $\mathcal{T}(\alpha)$ be the tree language $U \subseteq T_\Sigma$ associated with α as a tree-regular expression. Given $\mathcal{M} = \langle S, \pi, \rho \rangle$ we define $\rho(\alpha) = \bigcup_{\beta \in \mathcal{T}(\alpha)} \rho(\beta)$, where the interpretation of trec's $\rho(\beta)$ remains as before.

4.3. Example. Consider the following **sticky** scheme $\alpha = \beta ;^2 Q?(\$_0)$, where $\beta = (a(\$_1) \cap b(\$_2))^{*_1}$. A possible trec of β is $a(a(a(\$_1) \cap b(\$_2)) \cap b(\$_2)) \cap b(\$_2)$. The corresponding trec of α is

$$\tau = a(a(a(\$_1) \cap b(Q?(\$_2))) \cap b(Q?(\$_2))) \cap b(Q?(\$_2)).$$

Figure 8 gives a full description (a trace) of a possible run of α which corresponds to τ, in a model in which

$$\rho(a) = \{(s_i, s_{i+1}) \mid i \geq 0\}, \qquad \rho(b) = \{(s_i, s_{i+2}) \mid i \geq 0\}$$

and $\pi(Q) = \{s_i \mid 8 \geq i \geq 2\}$. The input/output pair contributed by this particular run

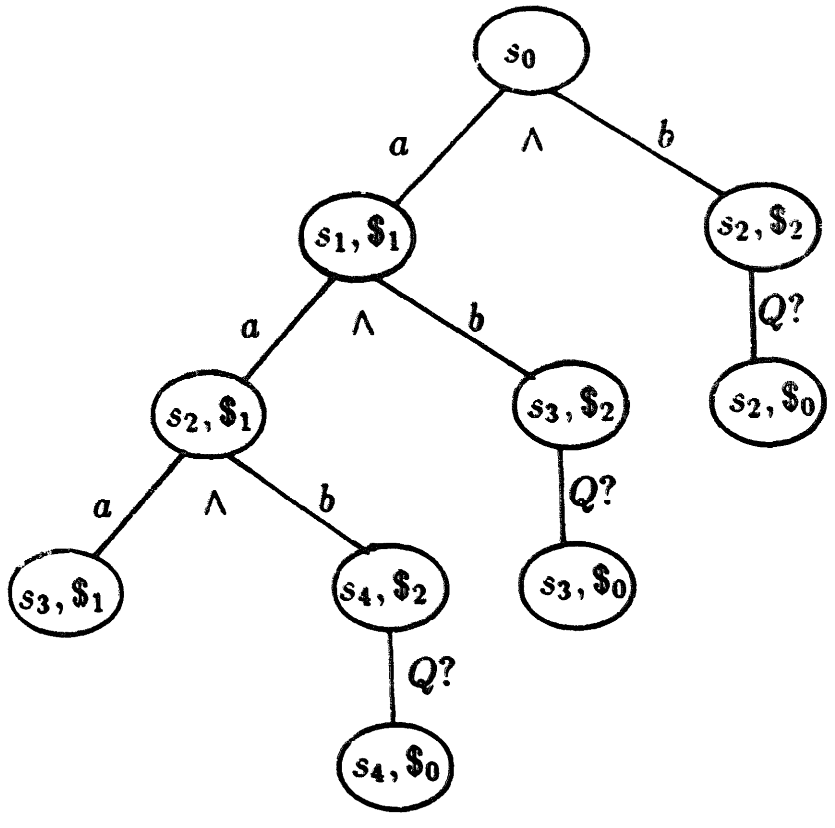

Fig. 8.

to $\rho^{\$}(\alpha)$ is (s_0, U) where

$$U = \{(0: U_0), (1: U_1)\}, \qquad U_0 = \{s_2, s_3, s_4\} \text{ and } U_1 = \{s_3\}.$$

The pair contributed to the final $\rho(\alpha)$ is $(s_0, \{s_2, s_3, s_3, s_4\})$. In general, by the semantic definitions we get, denoting $S_{i,j} = \{s_{i+2}, \ldots, s_{j+1}\}$ for $j \geq i \geq 0$,

$$\rho^{\$}(\beta) = \{(s_i, U_{i,j}) \mid j \geq i \geq 0\},$$

where $U_{i,j} = \{(1:\{s_j\}), (2: S_{i,j})\}$, and

$$\rho^{\$}(\alpha) = \{(s_i, U'_{i,j}) \mid 8 \geq j \geq i \geq 0\} \cup \{(s_i, \{(1:\{s_i\})\}) \mid i \geq 9\},$$

where $U'_{i,j} = \{(0: S_{i,j}), (1:\{s_j\})\}$, and finally,

$$\rho(\alpha) = \{(s_i, \{s_j\} \uplus S_{i,j}) \mid 8 \geq j \geq i \geq 0\} \cup \{(s_i, \{s_i\}) \mid i \geq 9\}.$$

4.4. Example. The following scheme is equivalent to (i.e., has the same interpretation ρ as) prog$_1$ of Example 4.1.

$$\text{prog}_2: (a(\$_1) \cup P?(\$_0) \cup (Q?(\$_0) \cup (b(\$_1) \cap c(\$_2)))^{*_2})^{*_1}; {}^1\text{false?}; {}^2\text{false?}.$$

Intuitively, the sticky labels $\$_1, \$_2$ correspond to the labels 1, 6 respectively in prog$_1$. While the semantics of **c-goto** ensures that all branches reach a nonexistent label in order to halt, and thus all leaves have to satisfy P or Q, this requirement has to be forced in the **sticky** scheme by the last two tests, which ensure that all leaves in the final trace are labeled by $\$_0$. Without these tests we might have also legal traces with leaves not satisfying $P \vee Q$ and labeled $\$_1$ or $\$_2$.

4.5. Example. Let

$$\alpha_1 = \$_1 \cup (a(\$_1) \cap b(\$_1)) \cup (a(\$_2) \cap b(\$_2)),$$

$$\alpha_2 = \$_2 \cup (a(\$_1) \cap b(\$_2)) \cup (a(\$_2) \cap b(\$_1)),$$

and

$$\text{even}_2 = (\alpha_1^{*_1} ;^2 \alpha_2^{*_2})^{*_1} ;^2 (\text{leaf} \wedge P)?(\$_0) ;^1 (\text{leaf} \wedge \neg P)?(\$_0).$$

This last scheme is equivalent to even$_1$ of Example 4.2. (The labels $\$_1$ and $\$_2$ correspond to the labels 1 and 6 therein.)

4.3. Concurrent regular schemes

Propositional concurrent regular schemes were introduced in [18] within the framework of concurrent propositional dynamic logic. The class of such schemes, which we denote here by **c-reg**, can be viewed as the class of **sticky** schemes which use only a single sticky label $\$_0$. (The semantics as defined there is slightly different but the changes bear no influence on the results described therein.) Taking this view, no special definition is necessary for this language. (We simplify the syntax somewhat by using the conventional ";" and "*" instead of "$;^0$" and "$*_0$" respectively.)

We note that **c-reg** schemes might not be flexible enough. The program $\alpha ; \beta$ means that β is to be executed from *every* end-point of α. This may cause difficulties in some cases in which we might want to distinguish between two (or more) types of end-states, and proceed differently from each type. Likewise, we might want to write a program similar to α^*, but so that α is allowed to be re-executed only from *some* of the end-states. The lack of these capabilities in the **c-reg** schemes is the reason why both programs of Examples 4.1 and 4.2 are conceivably inexpressible in **c-reg**(P, PC) (where PC stands for propositional calculus), and explains the motivation for introducing the mechanisms of concurrent goto schemes and sticky schemes described before.

4.4. The kill *command*

It is sometimes desirable to be able to terminate the execution of some of the processes of a concurrent program. For example, we may want to run several processes of a program α in parallel, and then terminate some of the processes and proceed in executing a program β in the remaining ones. Recall that, in the present situation (for **c-reg** schemes), $\alpha ; \beta$ is interpreted so that β is executed from the end-state of *every* branch of the executed program of α. Even in the **sticky** schemes, it is possible to 'bypass' some branches in applying the next subprogram (by using sticky labels), but their end-states will nevertheless appear in the final 'start-end' relation.

The *kill* command is meant to enable the termination of some branches of a program, so that further subprograms, and the final evaluation of the scheme, will

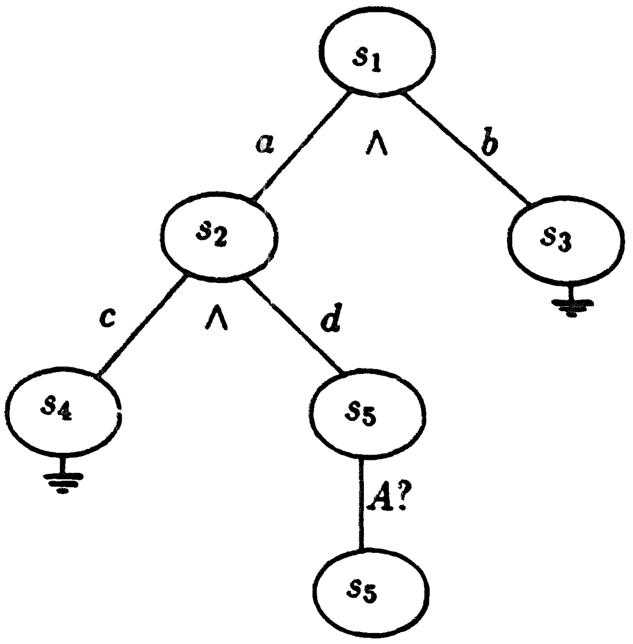

Fig. 9. A possible trace of α.

not refer to these branches. For example, consider the program

$$\alpha = (a \cap (b; \text{kill})); ((c; \text{kill}) \cap d); A?.$$

Figure 9 presents a possible trace corresponding to a run of α in a model in which $(s_1, s_2) \in \rho(a)$, $(s_1, s_3) \in \rho(b)$, $(s_2, s_4) \in \rho(c)$, $(s_2, s_5) \in \rho(d)$ and $s_5 \in \pi(A)$.

The semantics of the kill command may be defined by $\rho(\text{kill}) = \{(s, \emptyset) \mid s \in S\}$. Thus, the input-output pair contributed by the trace of Fig. 9 to $\rho(\alpha)$ is $(s_1, \{s_5\})$. Note that if $(s, \emptyset) \in \rho(\alpha)$, then $(s, \emptyset) \in \rho(\alpha; \beta)$ for every program β.

The resulting classes of schemes are denoted by the superscript K, e.g., **c-reg**^{K}.

5. Interconnections

This section contains some comparisons between the different scheme types defined in the previous sections. Throughout, we use the following uninterpreted (schematic) notions of equivalence. Two formulas φ, ψ are equivalent ($\varphi \equiv \psi$) if $\pi(\varphi) = \pi(\psi)$ in every model $\mathcal{M}$. For two logics L_1, L_2 interpretable over models as above, we say that $L_1 \leq L_2$ iff for every formula $\varphi \in L_1$ there is an equivalent formula $\psi \in L_2$. $L_1 = L_2$ iff $L_1 \leq L_2$ and $L_2 \leq L_1$. $L_1 < L_2$ iff $L_1 \leq L_2$ and not $L_2 \leq L_1$. Similarly, two schemes α, β are equivalent ($\alpha \equiv \beta$) if $\rho(\alpha) = \rho(\beta)$ in every model $\mathcal{M}$. For two classes of schemes C_1, C_2 interpretable over models as above, we say that $C_1 \leq C_2$ iff for every scheme $\alpha \in C_1$ there is an equivalent scheme $\beta \in C_2$. $C_1 = C_2$ iff $C_1 \leq C_2$ and $C_2 \leq C_1$. $C_1 < C_2$ iff $C_1 \leq C_2$ and not $C_2 \leq C_1$.

Throughout, whenever showing equivalence of two classes of schemes, we assume that the logics used by the two classes for tests are equivalent in expressiveness. That is, if schemes in C_1 use formulas of L_1 in tests and C_2 uses L_2, then we assume

$L_1 = L_2$. No other constraints are imposed on the tests. On the other hand, strict inequality results will be shown only with tests in restricted logics, typically PC on the propositional level and QF (the quantifier-free subset of first-order logic with equality) on the first-order level. For instance, the following is obvious, due to the fact that the concurrent semantics naturally extends the sequential one, and a concurrent scheme like $\alpha = a \cap b$ may have a pair $(s_1, \{s_2, s_3\})$ in its interpretation $\rho(\alpha)$, which no sequential scheme has.

5.1. Theorem. reg(P, PC) < c-reg(P, PC).

It is also clear from the definitions that the following theorem holds.

5.2. Theorem. c-reg(P, L_1) ≤ sticky(P, L_2), *assuming* $L_1 \leq L_2$.

Furthermore, when the logics are not too powerful (e.g., PC), the class **sticky** is strictly stronger.

5.3. Theorem. c-reg(P, PC) < sticky(P, PC).

Proof. Consider the **sticky(P, PC)** scheme

$$\text{seg} = (a(\$_1) \cap b(\$_2))^{*_1} ;^1 a(\$_2).$$

Consider a model in which $S = N$ (the integers), $\rho(a) = \{(i, i+1) \mid i \geq 0\}$, $\rho(b) = \emptyset$ for every atomic program $b \neq a$ and $\pi(Q) = \emptyset$ for every atomic proposition Q. In this model

$$\rho(\text{seg}) = \{(i, \{i+1, \ldots, j\}) \mid j > i \geq 0\}.$$

We show that in this model, no **c-reg(P, PC)** scheme has the same interpretation as seg. This is done as follows. Observe that the interpretation of seg has two interesting properties: it has unbounded concurrency, in the sense that the sets of end-states are of arbitrary (countable) cardinality, yet in each such set all states occur with multiplicity 1 (so we actually have sets of end-states, rather than multisets). We show that every **c-reg(P, PC)** scheme violates at least one of these properties.

Note that in the model at hand, the interpretation of every **c-reg(P, PC)** scheme α can be fully described by a collection $R(\alpha) = \{\Delta_1, \ldots\}$, where each Δ_i is a multiset of integers, $\Delta_i \subset N$, so that

$$\rho(\alpha) = \{(i, \{i + j \mid j \in \Delta\}) \mid i \geq 0, \Delta \in R(\alpha)\}.$$

The collections $R(\alpha)$ are defined inductively. The definition is similar in nature to that of the interpretation ρ; $R(\alpha)$ can in fact be thought of as the restriction of $\rho(\alpha)$ to pairs whose start-state is 0. The formal definition is left to the reader. The fact that $R(\alpha)$ completely characterizes $\rho(\alpha)$ in the above sense can be shown by induction on the structure of the schemes and is omitted too.

Using this characterization, we say that a scheme α has bounded concurrency, or BC, if $\exists K_\alpha \in N$ such that $\forall \Delta \in R(\alpha)(|\Delta| \leq K_\alpha)$. We say that α is multiple, or MULT, if $\exists \Delta \in R(\alpha)(\Delta \neq \text{set}(\Delta))$.

It remains to prove that every scheme in **c-reg**(P, PC) is either BC or MULT. This is again proved by induction on the structure of schemes. Atomic programs and tests are clearly all BC. Consider the case $\alpha = \beta \cup \gamma$. If either β or γ are MULT, then so is α. Otherwise both are BC, so α must be BC too, with $K_\alpha = \max\{K_\beta, K_\gamma\}$. The cases of $\alpha = \beta \cap \gamma$ and $\alpha = \beta ; \gamma$ are proved similarly, with the new bound K_α (when applicable) being $K_\beta + K_\gamma$ and $K_\beta \cdot K_\gamma$ respectively. Finally, consider the case $\alpha = \beta^*$. If every $\Delta \in R(\beta)$ is a singleton, then the same holds for $R(\alpha)$, so α is BC. Otherwise there exists some $\Delta \in R(\beta)$ with at least two elements $i, j \in \Delta$. Now, if $i = j$, then β is MULT, so α is MULT too. Otherwise, there is an execution of α corresponding to two consecutive executions of β in which Δ is used in all applications of β. This execution yields a multiset $\Delta' \in R(\alpha)$ in which the number $i+j$ appears at least twice. (The first execution of Δ from a state k leads to the states $k+i$ and $k+j$, and the second execution leads to the states $(k+i)+j$ and $(k+j)+i$ (among others).) Thus again α is MULT. □

Next we prove the equivalence of **c-goto** and **sticky** schemes.

5.4. Theorem. **c-goto**$(\mathrm{P}, \mathrm{L}_1) =$ **sticky**$(\mathrm{P}, \mathrm{L}_2)$, *assuming* $\mathrm{L}_1 = \mathrm{L}_2$.

Proof. Let α be a scheme in **sticky**(P, L$_1$). By Theorem 2.1, there exists a regular tree grammar G_α equivalent to α (i.e., generating the same set of trees). G_α can be translated into a **c-goto**(P, L$_1$) scheme β with the same trec's (modulo substitution of symbols $\$_i \in \Sigma_0$). This is done in three stages. In the first stage, for each variable A with production rules $A \to \varphi_1 | \cdots | \varphi_k$, we introduce new variables $A_1, \ldots, A_k$ and replace these rules by the rules

$$A \to A_1 | \cdots | A_k \quad \text{and} \quad A_1 \to \varphi_1, \ldots, A_k \to \varphi_k.$$

This obviously yields an equivalent grammar.

In the second stage we attach a unique (even) command label l_A to each variable A, and change production rules into program segments in the following way. Any production rule $A \to A_1 | \cdots | A_k$ is transformed into l_A: goto l_{A_1} or $\cdots$ or l_{A_k}. Similarly, any production rule $A \to (B) \cap (C)$ transforms into l_A: goto l_B and l_C. Production rules of the form $A \to d(B)$ transform into the two consecutive commands $l_A: d$; $l_A + 1$: goto l_B, and any production rule $A \to \$$ is transformed into l_A: goto l_{end}.

Finally, in the third stage we organize the resulting program segments sequentially in some arbitrary order, except that the segment belonging to the start symbol appears first (we assume, w.l.o.g., that the start symbol is unique). The labels are now renumbered consecutively in a consistent way, and l_{end} is replaced by $m+1$, where m is the number of command lines in the final program.

It is easy to see that the resulting scheme β (or actually, the corresponding grammar G_β described in Section 4.1) has the same set of trec's as G_α, except that G_β uses a single symbol \$ while G_α may use several such symbols. However, this has no influence on the semantics since after deriving the trec's of a program, all sticky labels are erased from the semantical interpretation ρ. Therefore, $\rho(\beta)=\rho(\alpha)$. Finally, obtain an equivalent scheme γ in **c-goto**(P, L_2) by replacing each test Q? in β with an equivalent test Q'? from L_2.

The converse direction is shown in a similar (somewhat simpler) manner. □

Note. The underlying translation algorithm from **c-goto** to **sticky**, based on the proof of Theorem 2.1 (see [6, Lemma 5.7]), might yield a **sticky** scheme (at most) exponentially larger than the original **c-goto** scheme. This fact bears influence on complexity issues discussed later.

Finally, note that the program kill itself is not programmable in any of the classes since $\rho(\text{kill})=\{(s,\emptyset)\mid s\in S\}$, and, for every program α in the above classes, in every pair $(s, U)\in\rho(\alpha)$, $U\neq\emptyset$. Therefore, it is obvious that the classes with kill are proper extensions of the original ones. For instance, we have the following theorem.

5.5. Theorem. c-reg(P, PC) < **c-reg**$^{\text{K}}$(P, PC).

6. Dynamic logics of concurrent schemes

6.1. The logics and their semantics

Dynamic logic (DL) is a logical framework for reasoning about programs (cf. [7]). Most research in this field has concentrated on sequential programs, i.e., flowcharts and while schemes, as well as certain higher-level versions such as context-free and recursively enumerable programs (cf. [7]). In [18] we proposed an extension of DL, named CDL, which is capable of dealing with concurrent regular schemes of the kind described in Section 4.3. The logic was discussed both on the propositional and first-order level. In the sequel we will refer to this logic (on the propositional level) as **c-reg**-PDL, in order to distinguish it from other versions. The schemes were allowed to inductively use 'rich tests', i.e., the logic defining the class of allowed tests was **c-reg**-PDL itself.

In this section we discuss the dynamic logics obtained by admitting **c-goto** and **sticky** schemes, namely **c-goto**-PDL and **sticky**-PDL respectively (both logics allow rich tests), in addition to **c-reg**-PDL.

We first give a precise definition of **c-goto**-PDL. Let $\text{AP}=\{P_i\mid i\geq 0\}$ be our collection of atomic propositions. The formulas of **c-goto**-PDL are defined as follows: Every $P\in\text{AP}$ is a formula, and if A, B are formulas, L is a set of formulas (i.e., a subset of **c-goto**-PDL) and α is a **c-goto**(P, L) scheme, then $A\vee B$, $\neg A$ and $\langle\alpha\rangle A$ are formulas too.

Formulas of **c-goto-PDL** are interpreted over models as described earlier, which contain a basic interpretation π for the atomic propositions of AP alone: $\pi(P)$ is the set of states satisfying P for every $P \in \mathrm{AP}$. This interpretation is extended as follows.

$$\pi(A \vee B) = \pi(A) \cup \pi(B), \qquad \pi(\neg A) = S - \pi(A),$$

$$\pi(\langle\alpha\rangle A) = \{s \mid \exists V((s, V) \in \rho(\alpha), \mathrm{set}(V) \subseteq \pi(A))\},$$

where $\rho(\alpha)$ is the interpretation of the scheme α as defined earlier. Note that the definitions of ρ and π are interleaved inductively since formulas involve schemes and schemes involve formulas (in tests).

Next we define **sticky-PDL**. This logic is based on the full interpretation relation of sticky schemes $\rho^{\$}$. It combines formulas and programs by admitting a formula $\langle\alpha\rangle A$, where $A = \{(i_j : A_{i_j}) \mid 1 \leq j \leq k\}$ is a tuple of *labeled formulas*, and $1 \leq i_1 < \cdots < i_k$. The semantics attached to such a formula is

$$\pi(\langle\alpha\rangle A) = \{s \mid \exists U((s, U) \in \rho^{\$}(\alpha) \wedge \forall i((i \in A \wedge i \in U) \Rightarrow \mathrm{set}(U_i) \subseteq \pi(A_i)))\}.$$

Thus, the states of U_i, interpreted as the leaf states labeled by $\$_i$, are required to satisfy A_i. If there are no leaves labeled $\$_i$, then the formula A_i is ignored. Conversely, if there is no formula A_i in A, then no requirement is made of the leaves labeled $\$_i$ (or, put another way, we take A_i to be true).

The logic **c-reg-PDL** is defined in a similar way, on the basis of the class **c-reg**(P, L).

It is also possible to define a version of **sticky-PDL** based on the restricted relation ρ, i.e., interpreting all leaves in the same way and ignoring the sticky labels. In such a version, the connection between programs and formulas is achieved in the usual way, by a formula $\langle\alpha\rangle A$, where a single formula A is required to hold at all leaves, regardless of their labels. This version is equivalent in expressive power to the present one since $\langle\alpha\rangle A$ can always be written as $\langle\alpha\,;^{i_1} A_{i_1}?(\$_0)\,;^{i_2}\cdots;^{i_k} A_{i_k}?(\$_0)\rangle$true. However, the logic in our definition lends itself better to axiomatization, as programs may be conveniently decomposed.

The logics obtained by allowing the kill command are denoted by the superscript K, as for the classes of schemes. Note that $\langle\mathrm{kill}\rangle A$ is valid for every formula A. Hence, $\langle\alpha \cap (\beta\,;\mathrm{kill})\rangle A$ for example is equivalent to $\langle\alpha\rangle A \wedge \langle\beta\rangle$true.

6.1. Example. Let us consider Examples 4.1 and 4.4 once again. The formulas $\langle\mathrm{prog}_1\rangle$true of **c-goto-PDL** and $\langle\mathrm{prog}_2\rangle$true of **sticky-PDL** are satisfiable in a state s iff $\mathrm{prog}_1/\mathrm{prog}_2$ are executable successfully from s. An equivalent formula is expressible in **c-reg**$^{\mathrm{K}}$**-PDL**:

$$\mathrm{form}_3 : \langle a^*\,;((P \vee Q)?\,;\mathrm{kill} \cup$$

$$(c\,;(Q?\,;\mathrm{kill} \cup \mathrm{true}?) \cap b\,; a^*\,;((P \vee Q)?\,;\mathrm{kill} \cup \mathrm{true}?))^*)\,;\mathrm{false}?\rangle\mathrm{true}.$$

The final false is intended to force all branches to terminate, satisfying the Q or $(P \vee Q)$ tests.

Moreover, we show in the next section that the kill command adds no power to **c-reg-PDL** (due to the presence of 'rich tests'). Indeed, an equivalent formula in **c-reg-PDL** is:

$$\text{form}_4: \langle a^*; (P? \cup (((b; a^*\rangle P)?; c \cup (b; a^* \cap c))^*; Q?)\rangle \text{true}.$$

Intuitively, the main * in this formula computes the task of the node labeled 6 in prog_1. If $b; a^*$ can be executed to reach a state satisfying P, then only c is executed. Otherwise, $b; a^* \cap c$ is executed. All branches must end with leaves satisfying Q.

The rest of this section contains some expressiveness results concerning the dynamic logics defined above.

6.2. Interconnections

Our first claim is that **c-goto-PDL** and **sticky-PDL** are equivalent in expressiveness.

Define the following sequences of classes of schemes GP_i and logics GPL_i for $i \geq 0$: Let GPL_0 be the language of propositional calculus involving atomic propositions $\text{AP} = \{P_j \mid j \geq 0\}$. For every $i \geq 0$,

(1) let GP_i be **c-goto**(P, GPL_i), and

(2) let GPL_{i+1} be the subset of **c-goto-PDL** based on the schemes of GP_i alone (i.e., the collection of formulas containing atomic propositions and closed under $\varphi \vee \psi$, $\neg\varphi$ and $\langle\alpha\rangle$true for a scheme $\alpha \in \text{GP}_i$).

Define the sequences SP_i and SPL_i similarly, based on **sticky** schemes. Then **c-goto-PDL** $= \bigcup_{i \geq 0} \text{GL}_i$ and **sticky-PDL** $= \bigcup_{i \geq 0} \text{SL}_i$.

6.2. Lemma. *For every* $i \geq 0$,

(1) $\text{SPL}_i = \text{GPL}_i$, *and*

(2) $\text{SP}_i = \text{GP}_i$.

Proof. By induction on i. For $i = 0$, statement (1) is trivial and statement (2) follows from Theorem 5.4 together with the first statement. For $i > 0$, each of the two directions of statement (1) is proved by induction on the structure of formulas, where formulas of the form $\langle\alpha\rangle$true are resolved by part (2) of the inductive hypothesis for $i-1$, and statement (2) follows as in the case of $i = 0$. □

6.3. Theorem. c-goto-PDL = sticky-PDL.

Proof. First we have to overcome the differences in the format of formulas involving the 'diamond' connective. To this end, we assume that every formula $\langle\alpha\rangle A$ in **c-goto-PDL** satisfies $A = \text{true}$. This causes no loss of generality since every formula $\langle\alpha\rangle A$ has an equivalent formula $\langle(\alpha, m+1 : A?)\rangle$true, where α contains m commands. We make a similar assumption regarding **sticky-PDL**, where here any formula $\langle\alpha\rangle A$ where $A = \{(i_j : A_{i_j}) \mid 1 \leq j \leq k\}$ is simulated by the formula $\langle\alpha; {}^{i_1}A_{i_1}?(\$_{i_1}); {}^{i_2}\cdots; {}^{i_k}A_{i_k}?(\$_{i_k})\rangle$true.

Now the proof is completed using Lemma 6.2. □

Note that Theorem 6.3 holds also for the same logics with 'poor' tests, e.g., confined to propositional tests.

Following a remark made in the previous section, we note also that formulas in **c-goto-PDL** may be exponentially more succinct than the equivalent ones in **sticky-PDL**. This is in accordance with a similar observation made in [10] regarding the relationships between regular PDL and PDL of flowcharts.

Finally, we observe that, in contrast with the situation for schemes, the kill command does not strengthen the corresponding logics, not even **c-reg-PDL** as expressed in the following theorem.

6.4. Theorem. c-reg$^{\mathrm{K}}$-PDL = c-reg-PDL.

Proof. See Appendix A. □

A major problem left open by this paper is proving (or disproving) the following conjecture.

6.5. Conjecture. c-reg-PDL < sticky-PDL.

6.3. Relationships with the μ-calculus

In this section we discuss relationships with the μ-calculus. The propositional μ-calculus L_μ, defined by Kozen in [12], can be viewed as another extension of PDL. Its syntax contains atomic formulas P_i, atomic programs a_i, and the following construction rules:

(1) each P_i is a formula;

(2) if A, B are formulas, a is an atomic program and $F(R)$ is a formula with positive appearances of a new atomic symbol R (i.e., such that F contains no subformulas of the form $\mu R.F'$), then $A \wedge B$, $A \vee B$, $\neg A$, $\langle a\rangle A$ and $\mu R.F(R)$ are formulas.

The semantics interprets formulas over models $\langle S, \pi, \rho\rangle$, where S is a set of states, π attaches a subset $\pi(P)$ of S to every atomic formula P, and ρ attaches a subset $\rho(a)$ of $S \times S$ to every atomic program a. We extend π to every formula by the following rules:

$$\pi(A \vee B) = \pi(A) \cup \pi(B), \qquad \pi(A \wedge B) = \pi(A) \cap \pi(B),$$

$$\pi(\neg A) = S - \pi(A), \qquad \pi(\langle a\rangle A) = \{s \mid \exists t((s, t) \in \rho(a) \wedge t \in \pi(A))\},$$

and

$$\pi(\mu R.F(R)) = \min\{U \mid U \subseteq S \wedge U = F(U)\},$$

where $F(U)$ stands for $\pi(F(R))$ with $\pi(R) = U$, and the minimum is taken w.r.t. the subset ordering.

For example, the PDL formula $\langle \alpha^*\rangle A$ can be expressed in L_μ as $\mu R.(A \vee \langle \alpha\rangle R)$.

Consider the *continuous* sublanguage of L_μ, named CL_μ, obtained by the additional requirement that, in each formula $\mu R.F(R)$, appearances of R are under no negations at all. (This definition parallels that of Park [17] for the first-order continuous μ-calculus.)

6.6. Theorem. CL_μ = c-goto-PDL.

Proof. ($\leq$): Let A be a formula in CL_μ involving only k μ-subformulas, $\mu R_i.F_i(R_i)$, $1 \leq i \leq k$. A can be described as being constructed from occurrences of the symbols R_i, $1 \leq i \leq k$, and m maximal $\boldsymbol{R}$-free formulas $G_1, \ldots, G_m$, combined by the connectives $B \wedge C$, $B \vee C$, $\langle a \rangle B$ and $\mu R_i.F_i(R_i)$. Negation may appear only within the subformulas $G_1, \ldots, G_m$, due to the continuity requirement.

We construct a c-goto(P, PC) program α_A such that $\langle \alpha_A \rangle$true $\equiv A$. For every subformula B of A as just described, add a program segment starting with the label l_B as follows:

$B = G_i$ ($\boldsymbol{R}$-free):	$l_B : G_i?$
	$-$: halt
$B = R_i$:	l_B : goto l_{F_i}
$B = C \vee D$:	l_B : goto l_C or l_D
$B = C \wedge D$:	l_B : goto l_C and l_D
$B = \langle a \rangle C$:	$l_B : a$
	$-$: goto l_C
$B = \mu R_i.F_i(R_i)$:	l_B : goto l_{F_i}.

The segment starting with l_A has to come first, but otherwise the segments can be combined in an arbitrary order to yield α_A.

($\geq$): A formula in c-goto-PDL is translated into CL_μ by structural induction. Consider the case of a subformula $\langle \alpha \rangle A$. It is first transformed into $\langle(\alpha, m+1 : A?)\rangle$true (assuming $\alpha = (1 : \gamma_1, \ldots, m : \gamma_m)$). By the inductive hypothesis we assume the existence of an equivalent formula A' in CL_μ for every formula A appearing as a test $A?$ in the program. Then we construct a formula A_α in CL_μ s.t. $\langle \alpha \rangle$true $\equiv A_\alpha$.

The idea is to associate a recursion symbol R_l and a formula $\mu R_l.F_l(R_l)$ with every command $l : \gamma_l$ $(1 \leq l \leq m+1)$.

A "goto l" is then interpreted as "R_l", or as "$\mu R_l.F_l(R_l)$" if this is the first occurrence of l in this subformula. A c-goto-PDL formula $\langle \alpha \rangle$true can be translated directly into CL_μ if the graph description of α is a tree with 'backward edges'. That is, its only goto arcs are either forward or to some predecessor (e.g., the program prog$_1$ of Example 3.1 is in such form). In contrast, for a general c-goto scheme α the resulting CL_μ formula may have several (not necessarily identical) subformulas F_l for a label l, within different subformulas, due to the process of unwinding it into the desired form. For example, if

$$\begin{aligned}\alpha = 1&:\text{goto } 2 \text{ or } 3\\ 2&:\text{goto } 3 \text{ or } 4\\ 3&:\text{goto } 2 \text{ or } 4\\ 4&:A?,\end{aligned}$$

then the described translation scheme for A_α first yields $\mu R_1.F_1(R_1)$, and further manipulation yields

$$\mu R_1(\mu R_2.F_2(R_2) \vee \mu R_3.F_3(R_3)).$$

Proceeding to construct F_2 and F_3 we see that each F_i has to call the R_j symbol of the other:

$$\mu R_1.(\mu R_2.(R_3 \vee R_4) \vee \mu R_3.(R_2 \vee R_4)).$$

This is not allowed in the formalism of CL_μ. Thus, for instance, F_2 will have to contain a copy of F_3 in place of R_3. This way we finally obtain

$$\mu R_1.(\mu R_2.(\mu R_3(R_2 \vee \mu R_4.(A')) \vee \mu R_4.(A')) \vee \mu R_3.(\mu R_2(R_3 \vee \mu R_4.(A')) \vee \mu R_4.(A'))),$$

where A' is the equivalent of A in CL_μ.

The formulas are constructed recursively, keeping, for each subformula $F_l(R_l)$, a set V_l of labels i s.t. F_l is internal to a copy of F_i, so R_i may be used to represent "goto i". The construction goes as follows: A_α is set to be $\mu R_1.F_1(R_1)$, and $V_1 = \{1\}$, and then the formulas F_l are constructed according to the command γ_l, by the following rules:

Case $\gamma_l = \text{goto } l' \text{ or } l''$: Let $F_l = B' \vee B''$, where $B' = R_{l'}$ if $l' \in V_l$; otherwise $B' = \mu R_{l'}.F_{l'}(R_{l'})$, where $F_{l'}$ is constructed recursively with $V_{l'} = V_l \cup \{l'\}$. B'' is defined analogously with respect to l''.

Case $\gamma_l = \text{goto } l' \text{ and } l''$: Let $F_l = B' \wedge B''$, where B' and B'' are defined as in the previous case.

Case $\gamma_l = a$: Let $F_l = \langle a \rangle B$, where $B = R_{l+1}$ if $l+1 \in V_l$; otherwise $B = \mu R_{l+1}.F_{l+1}(R_{l+1})$, where F_{l+1} is constructed recursively with $V_{l+1} = V_l \cup \{l+1\}$.

Case $\gamma_l = A?$: A special case is when $l = m+1$. In such a case we just let $F_l = A'$, the equivalent of A in CL_μ by the inductive hypothesis. Otherwise, let $F_l = A' \wedge B$, where B is defined as in the previous case.

It is clear that the construction terminates, as the sets V_l cannot grow larger than size m, which bounds the depth of recursion. It is also clear that the resulting formula is equivalent in meaning to $\langle \alpha \rangle$true. □

6.7. Example. The CL_μ formula corresponding to $\langle \text{prog}_1 \rangle$true of **c-goto-PDL** is

$$\mu R_1.(P \vee \langle a \rangle R_1 \vee \mu R_2.(Q \vee (\langle b \rangle R_1 \wedge \langle c \rangle R_2))).$$

The CL_μ formula equivalent to $\langle \text{even}_1 \rangle$true from Example 4.2, i.e., stating that the subtree rooted at a state s has an even number of leaves, is

$$\mu R_1.((\text{leaf} \wedge \neg P) \vee (\langle a \rangle R_1 \wedge \langle b \rangle R_1) \vee (\langle a \rangle \psi \wedge \langle b \rangle \psi)),$$

where

$$\psi = \mu R_2.((\text{leaf} \wedge P) \vee (\langle a\rangle R_1 \wedge \langle b\rangle R_2) \vee (\langle a\rangle R_2 \wedge \langle b\rangle R_1)).$$

In both examples, the R_1 (R_2) fixpoint corresponds to the node labeled 1 (6) in the **c-goto** graph, or to the sticky label $\$_1$ ($\$_2$) respectively.

7. Validity and axiomatization

In [18], **c-reg-PDL** was given a complete axiom system and an elementary decision procedure. In fact, this was done for a *monotone* version of the logic, which is a subset of Parikh's game logic [16]. The monotone version requires $\rho(\alpha)$ to be monotone for every program α, in the sense that if $(s, U) \in \rho(\alpha)$ and $U \subseteq V$, then also $(s, V) \in \rho(\alpha)$. This requirement may be expressed equivalently by

(1) defining $\rho(P?) = \{(s, U) \mid s \in (U \cap \pi(P))\}$, and

(2) limiting the set of possible models to *monotone models*, i.e., models in which every atomic program is monotone. The axiom system shown complete for this version in [18] is an appropriate axiom system for PDL or the 'dual-free' game logic [16] with the obvious extra axiom for $\cap$:

(A$\cap$) $\langle \alpha \cap \beta\rangle A \equiv \langle \alpha\rangle A \wedge \langle \beta\rangle A$.

This axiom system can be augmented to cover the nonmonotone version by adding the following axiom scheme:

(A$\vee$) $\langle a\rangle(A \vee B) \equiv \langle a\rangle A \vee \langle a\rangle B$ for any atomic program a.

Completeness of the nonmonotone system will not be shown here explicitly. Instead, we give a more general axiom system for (the nonmonotone version of) **sticky-PDL** and prove its completeness, in a way that extends the proof for **c-reg-PDL**.

Recall that labeled formulas are denoted by $A = \{(i_j : A_{i_j}) \mid 1 \leqslant j \leqslant k\}$. Let $A[i/B]$ denote $(A - \{(i : A_i)\}) \cup \{(i : B)\}$, i.e., A with B replacing A_i. The axiom system for **sticky-PDL** is the following:

Axiom schemes

(A1) All tautologies of the propositional calculus,

(A2) $\langle \$_i\rangle\{(i : A_i)\} \equiv A_i$,

(A3) $\langle B?(\beta)\rangle A \equiv \langle B?(\$_i) ;^i \beta\rangle A$,

(A4) $\langle B?(\$_i)\rangle\{(i : A_i)\} \equiv B \wedge A_i$,

(A5) $\langle a(\beta)\rangle A \equiv \langle a(\$_i) ;^i \beta\rangle A$ for an atomic program a,

(A6) $\langle a(\$_i)\rangle\{(i : A \vee B)\} \equiv \langle a(\$_i)\rangle\{(i : A)\} \vee \langle a(\$_i)\rangle\{(i : B)\}$ for an atomic program a,

(A7) $\langle a(\$_i)\rangle\{(i : A)\} \equiv \langle a(\$_j)\rangle\{(j : A)\}$ for an atomic program a,

(A8) $\langle \alpha \cup \beta\rangle A \equiv \langle a\rangle A \vee \langle \beta\rangle A$,

(A9) $\langle \alpha \cap \beta\rangle A \equiv \langle \alpha\rangle A \wedge \langle \beta\rangle A$,

(A10) $\langle \alpha ;^i \beta\rangle A \equiv \langle \alpha\rangle A[i/\langle \beta\rangle A]$,

(A11) $\langle\alpha^{*_i}\rangle A \equiv A_i \vee \langle\alpha\rangle A[i/\langle\alpha^{*_i}\rangle A]$,
(A12) $\langle\alpha\rangle A \equiv \langle\alpha\rangle(A \cup \{(i:B_i)\})$ for $\$_i \notin \alpha$, $i \notin A$,
(A13) $\langle\alpha\rangle A \equiv \langle\alpha\rangle(A \cup \{(i:\text{true})\})$ for $\$_i \in \alpha$, $i \notin A$.

Inference rules

(MP) $$\frac{A,\ A \supset B}{B},$$

(MONO) $$\frac{\{A_i \supset B_i\}_{1\le i\le k}}{\langle\alpha\rangle A \supset \langle\alpha\rangle B},$$

(IND) (for any $1 \le i \le k$) $$\frac{\langle\alpha\rangle A \supset A_i}{\langle\alpha^{*_i}\rangle A \supset A_i}.$$

Axioms (A12) and (A13) cover the cases where the set of sticky labels occurring in α does not coincide with the set of indices of the formulas; (A12) treats a formula B_i with no label $\$_i$ among the leaves of α, while (A13) treats a label $\$_i$ in α with no matching formula.

7.1. Theorem. *The above axiom system is complete for* **sticky-PDL**; *moreover, the validity problem for the logic can be decided in nondeterministic exponential time.*

Proof. See Appendix B. □

By the remark following Theorem 5.4 we have the following theorem.

7.2. Theorem. *The validity problem for* **c-goto-PDL** *can be decided in nondeterministic double exponential time.*

8. Boolean schemes and logics

8.1. Boolean sequential schemes and logics

The introduction of Boolean variables to propositional schemes and logics was proposed by Abrahamson [1] in the framework of propositional dynamic logic. The language of the schemes is extended with a set $\{X_i\}$ of Boolean variables. These variables may appear in the atomic operations

(ASSIGN) $l: X \leftarrow 0$ or $l: X \leftarrow 1$,

and the formulas $X = 0$, $X = 1$ are allowed in the language, including within tests. This gives rise to a new intermediate level between the propositional and the first-order ones, namely, the Boolean-variable level. We may consider either the goto schemes or the regular schemes of Section 3. The resulting classes of schemes are denoted **reg**(B, L) and **goto**(B, L), and the corresponding logics are denoted

reg-BDL and **goto**-BDL. The semantics has to be extended and based on the notion of *Boolean models*. Such models employ extended states, which consist of two parts: a state s and an interpretation $I:\{X_i\}\to\{0,1\}$ for the Boolean variables. Hence, such a Boolean model $\mathcal{M}$ is of the form $\langle S_{\mathcal{M}}, \pi_{\mathcal{M}}, \rho_{\mathcal{M}}, I_{\mathcal{M}}\rangle$, where $S_{\mathcal{M}}$, π_M and $\rho_{\mathcal{M}}$ are as on the propositional level, and $I_{\mathcal{M}}$ is the initial interpretation for the Boolean variables. The definitions of π and ρ need to be extended accordingly, and to address the extended states (s, I). An atomic formula $X=0$ is interpreted as

$$\pi(X=0)=\{(s, I)\mid s\in S\wedge I(X)=0\},$$

and similarly for $X=1$. The atomic operation $X\leftarrow 0$ is interpreted as

$$\rho(X\leftarrow 0)=\{((s, I), (s, I[0/X]))\mid s\in S\},$$

where $I[0/X]$ is an interpretation equivalent to I, except that X is interpreted as 0. (Similar notation will be used in several different contexts in the sequel.) The operation $X\leftarrow 1$ is interpreted analogously.

The usual atomic propositions and programs do not affect the interpretation I. Thus,

$$\pi(P)=\{(s, I)\mid I \text{ is a Boolean interpretation, } s\in\pi_{\mathcal{M}}(P)\},$$

and

$$\rho(a)=\{((s, I), (s', I))\mid I \text{ is a Boolean interpretation, } (s, s')\in\rho_{\mathcal{M}}(\alpha)\}.$$

Extending π and ρ to arbitrary schemes and formulas is done exactly as on the propositional level, regarding extended states (s, I) as states. Finally, a formula A is satisfied in the model $\mathcal{M}$ iff there exists a state $s\in S$ s.t. $(s, I_{\mathcal{M}})\in\pi(A)$. Thus, although our state space contains all possible pairs (s, I) (i.e., for any $s\in S$ and any interpretation I), only the special pairs $(s, I_{\mathcal{M}})$ are considered when defining the notion of satisfiability.

Hereafter, schemes and logics of this kind (i.e., equipped with Boolean variables and interpreted over Boolean models) will be referred to as *Boolean schemes* and *Boolean logics*.

Note. The Boolean-variable level forms a certain synthesis between the propositional level and a version of the first-order level (discussed in the next section) restricted to the fixed domain $D=\{0, 1\}$. There are several plausible ways of defining the precise notions of model, interpretation and satisfiability on such intermediate level. For instance, we could define the set of states as a collection of pairs (s, I), or alternatively discard $I_{\mathcal{M}}$ completely and consider the whole of $\{(s, I)\mid s\in S, I \text{ is some interpretation}\}$ as the model's state space. We chose to follow the version described originally in [1]. Similar results apply for other definitions as well, using the same general methods.

As noted by Abrahamson [1], schemes and logics equipped with Boolean variables are really incomparable with propositional schemes (and logics) since schemes and formulas in the propositional level cannot refer to the variables. However, comparisons can be made by ignoring the 'Boolean' part of the semantics, and considering the 'state' part alone. Such comparisons are dependent on the initial values of the Boolean variables. Thus, every Boolean scheme can be simulated by a set of propositional schemes, one scheme for each possible assignment I of initial values to the variables. The interpretation of any of the propositional schemes is identical to the 'state part' of the original scheme, assuming that initial interpretation.

Let L_1 be a Boolean logic, L_2 be a propositional logic, C_1 be a class of Boolean schemes and C_2 be a class of propositional schemes. We define the following notions. For a formula $A \in L_1$ involving Boolean variables $X = (X_1, \ldots, X_n)$, a formula $B \in L_2$ and a Boolean tuple $b = (b_1, \ldots, b_n)$, we say that $A \equiv_b B$ (*b equivalence*) iff, for every state s and every interpretation I, in every Boolean model,

$$s \in \pi(B) \Leftrightarrow (s, I[b/X]) \in \pi(A).$$

$L_1 \leq_I L_2$ iff, for every formula $A \in L_1$ (involving X) and for every Boolean tuple b, there is a formula $A_b \in L_2$ s.t. $A \equiv_b A_b$. $L_1 =_I L_2$ (*I-equivalence*) iff $L_1 \leq_I L_2$ and $L_2 \leq L_1$. Similarly, for a scheme $\alpha \in C_1$ involving Boolean variables $X = (X_1, \ldots, X_n)$, a scheme $\beta \in C_2$ and a Boolean tuple $b = (b_1, \ldots, b_n)$, we say that $\alpha \equiv_b \beta$ (b equivalence) iff, for every state s_1, s_2 and interpretation I_1, I_2, in every Boolean model,

$$((s_1, I_1[b/X]), s_2, I_2) \in \rho(\alpha) \Leftrightarrow (s_1, s_2) \in \rho(\beta).$$

$C_1 \leq_I C_2$ iff, for every scheme $\alpha \in C_1$ (involving X) and for every Boolean tuple b, there is a scheme $\alpha_b \in C_2$ s.t. $\alpha \equiv_b \alpha_b$. $C_1 =_I C_2$ (I-equivalence) iff $C_1 \leq_I C_2$ and $C_2 \leq C_1$.

These definitions give rise to two possibilities for full comparison of a Boolean system with an I-equivalent propositional one. Each of these possibilities requires a change in one of the logics to form a common basis.

The first alternative is to consider Boolean models. This requires extending the semantics of the propositional system so as to allow it to recognize Boolean variables as atomic predicates (though it cannot *act* upon them).

The other possibility is to consider propositional models. These models do not contain an I_M component, so the initial interpretation for the Boolean variables is undefined ($I(X) = \uparrow$). This requires us to restrict the Boolean system in such a way that to Boolean variables values must be assigned before they are tested. Then the semantic rules for the resulting system will construct the interpretation I for Boolean variables gradually, defining $I(X_i)$ only after X_i is assigned to for the first time. For instance, $\langle X \leftarrow 1; a\rangle(X = 1 \wedge P)$ is a legal formula in the language, but $\langle b; X = 0?\rangle Q$ is not, as it has no interpretation in a propositional model.

Let L_1, L_2, C_1 and C_2 be as above. Denote by L_2^B, C_2^B the systems obtained by changing L_2, C_2 according to the first alternative, and denote by L_1^P, C_1^P the systems obtained by changing L_1, C_1 according to the second one. Then we have the following lemma.

8.1. Lemma. *If* $\mathrm{C}_1 \leqslant_I \mathrm{C}_2$, *then*

(1) $\mathrm{C}_1 \leqslant \mathrm{C}_2^{\mathrm{B}}$,

(2) $\mathrm{C}_1^{\mathrm{P}} \leqslant \mathrm{C}_2$.

Proof. Let the equivalent of $\alpha \in \mathrm{C}_1$ be $\bigcup_b ((X = b)?; \alpha_b)$ in $\mathrm{C}_2^{\mathrm{B}}$, and let the equivalent of $\alpha \in \mathrm{C}_1^{\mathrm{P}}$ be α_b for an arbitrary b, where the scheme α_b is the b-equivalent of α in C_2. □

8.2. Lemma. *If* $\mathrm{L}_1 \leqslant_I \mathrm{L}_2$, *then*

(1) $\mathrm{L}_1 \leqslant \mathrm{L}_2^{\mathrm{B}}$,

(2) $\mathrm{L}_1^{\mathrm{P}} \leqslant \mathrm{L}_2$.

Proof. Similar. □

Abrahamson [1] shows that **reg-BDL** = **reg-PDL**, i.e., that extending **reg**(P, L) with Boolean variables adds no computational power. In the remainder of this section we extend our concurrent schemes with Boolean variables in a similar manner and study the resulting scheme classes and logics. The main result of this section is extending Abrahamson's observation to the concurrent case.

8.2. Boolean concurrent schemes and logics

It is possible to add Boolean variables to the concurrent schemes and logics, just as for sequential programs. This requires extending the semantics accordingly, and including an interpretation $I:\{X_i\} \to \{0, 1\}$ for the Boolean variables in every 'instantaneous description' of the process. In the case of the **c-reg** schemes, for instance, the semantics $\rho(\alpha)$ of a scheme α becomes a set of pairs of the form $((s, I), U)$, where U is a set of pairs $U = \{(s_i, I_i)\}$. For such a set we denote by U^S the set $\{s_i \mid \exists I_i((s_i, I_i) \in U)\}$. The interpretation of the operations $X \leftarrow 0$ $(X \leftarrow 1)$ and the tests $X = 0?$ $(X = 1?)$ is the same as in the sequential case, retaining the sequential nature of these primitives.

Similar semantics can be defined for Boolean variables in **sticky** or **c-goto** schemes. The resulting classes of schemes are denoted **c-reg**(B, L), **sticky**(B, L) and **c-goto**(B, L), and the corresponding logics are denoted **c-reg-BDL**, **sticky-BDL** and **c-goto-BDL**.

Note that we use variables only as local ones; any distinct process (branch) has its own copy of the variables, and has no knowledge of or influence on the variables of other processes. This is in contrast with a version defined in [19], where Boolean variables (along with a test & set operation) were used as *shared* variables, resulting in a much more powerful system, in which communication is available between processes.

These logics can also be defined as the unions of sequences of partial logics, as done before for propositional logics. Define the following sequences of classes of schemes GB_i and logics GBL_i for $i \geqslant 0$:

Let GBL_0 be the language of propositional calculus involving atomic propositions $\mathrm{AP} = \{P_j \mid j \geq 0\} \cup \{X_i = 0, X_i = 1 \mid i \geq 1\}$. For every $i \geq 0$,

(1) let GB_i be **c-goto**$(\mathrm{B}, \mathrm{GBL}_i)$, and

(2) let GBL_{i+1} be the subset of **c-goto-BDL** based on the schemes of GB_i alone (i.e., the collection of formulas containing atomic propositions and closed under $\varphi \vee \psi$, $\neg\varphi$ and $\langle\alpha\rangle$true for a scheme $\alpha \in \mathrm{GB}_i$).

Then **c-goto-BDL** $= \bigcup_{i \geq 0} \mathrm{GBL}_i$.

8.3. Example. Let

$$\alpha_1 = (a; X \leftarrow 0 \cap b; X \leftarrow 0) \cup (a; X \leftarrow 1 \cap b; X \leftarrow 1),$$

$$\alpha_2 = (a, X \leftarrow 0 \cap b; X \leftarrow 1) \cup (a; X \leftarrow 1 \cap b; X \leftarrow 0),$$

$$\mathrm{even}_3 = X \leftarrow 0; ((\neg\mathrm{leaf})?; (X = 0?; \alpha_1 \cup X = 1?; \alpha_2))^*; (\mathrm{leaf} \wedge (P \equiv (X = 1)))?.$$

This last scheme is equivalent to even_1 of Example 4.2, or to even_2 of Example 4.5. (The values $X = 0$, $X = 1$ correspond to the labels $\$_1$ and $\$_2$ there.)

8.3. Expressiveness results

Most of the basic results from the propositional level carry over to the Boolean-variable level. In particular we have, analogous to Theorems 5.1, 5.5, 5.4, 6.3 and 6.4, the following theorem.

8.4. Theorem

(1) **reg**(B, PC) < **c-reg**(B, PC) < **c-reg**$^{\mathrm{K}}$(B, PC).

(2) **c-goto**$(\mathrm{B}, \mathrm{L}_1) =$ **sticky**$(\mathrm{B}, \mathrm{L}_2)$, *assuming* $\mathrm{L}_1 = \mathrm{L}_2$.

(3) **sticky-BDL** = **c-goto-BDL**.

(4) **c-reg**$^{\mathrm{K}}$**-BDL** = **c-reg-BDL**.

One significant difference is that, unlike the propositional case, on the Boolean-variable level 'one sticky label suffices', i.e., **c-reg** schemes are as powerful as **sticky** ones.

8.5. Theorem

(1) **c-reg**$(\mathrm{B}, \mathrm{L}_1) =$ **c-goto**$(\mathrm{B}, \mathrm{L}_2)$, *assuming* $\mathrm{L}_1 = \mathrm{L}_2$.

(2) **c-reg-BDL** = **c-goto-BDL**.

Proof. (1): The $\leq$-direction is trivial. The proof of the $\geq$-direction goes along the lines of the proof that sequential **goto** schemes can be translated into **while** schemes (cf. [8]) using propositional variables. The proof involves, in fact, only a single while. Let $\alpha = (1 : \gamma_1, \ldots, m : \gamma_m)$ be a **c-goto**$(\mathrm{B}, \mathrm{L}_1)$ scheme. The labels of the scheme are traced by a tuple $X = (X_1, \ldots, X_{\lceil \log m \rceil})$ of new Boolean variables whose values correspond to the binary representation of the current label.

For every line l in α construct a subprogram α_l as follows:

Case $l:a$: Let α_l be $X = l?; a; X \leftarrow l+1$, where a is an atomic program or a Boolean assignment.

Case $l: P?$: Let α_l be $X = l?; P'?; X \leftarrow l+1$, where P' is the equivalent of P in L_1.

Case l:goto l' or l'': Let α_l be $X = l?; (X \leftarrow l' \cup X \leftarrow l'')$.

Case l;goto l' and l'': Let α_l be $X = l?; (X \leftarrow l' \cap X \leftarrow l'')$.

Finally, the entire program is replaced by

$$\alpha' = X \leftarrow 1; \left(\bigcup_{1 \leq l \leq m} \alpha_l \right)^*; \; X = m+1?.$$

Note that, contrary to the situation on the propositional level, the **c-reg/sticky** scheme simulating a **c-goto** scheme is only polynomially larger than the original scheme.

(2): Shown by using the technique of Theorem 6.3, and relying on part (1) of Theorem 8.5. □

The main result of this section concerns establishing Abrahamson's result in the concurrent setting. Comparisons of Boolean and propositional concurrent schemes are based on the same notions as for sequential schemes. The only definition which requires some change is that of b-equivalence of schemes. For a scheme $\alpha \in C_1$ using Boolean variables $X = (X_1, \dots, X_n)$, a scheme $\beta \in C_2$ and a Boolean tuple $b = (b_1, \dots, b_n)$, we say that $\alpha \equiv_b \beta$ (b equivalence) iff, for every state s, every set $U = \{(s_i, I_i) \mid 1 \leq i \leq k\}$ and every interpretation I, in every Boolean model,

$$((s, I[\bar{b}/\bar{X}]), U) \in \rho(\alpha) \Leftrightarrow (s, U^s) \in \rho(\beta).$$

We show the following theorem.

8.6. Theorem. **c-goto**(P, L_1) $=_l$ **c-goto**(B, L_2), *assuming* $L_1 =_l L_2$.

Proof. We only have to show that whenever $L_1 \leq_l L_2$, **c-goto**(B, L_1) $\leq_l$ **c-goto**(P, L_2). Let α be a given **c-goto**(B, L_1) scheme, and let b be a given Boolean tuple. Construct 2^n identical copies of α, denoted $\alpha^1, \dots, \alpha^{2^n}$, with the labels of α^j being $1^j, \dots, m^j$. The idea is that a run of the new program will reach label l^j just when the original α had to reach label l with X evaluating to j when read as the binary representation of a number, $X_1 \dots X_n$. Now replace any Boolean assignment $l^j : X_i \leftarrow 0$ by l^j : goto$(l+1)^{j'}$, where j' is identical in binary representation to j except that its ith digit is 0. Similarly for $X_i \leftarrow 1$. A test $l^j : A?$ is replaced by $l^j : A_{b^j}$, the b^j-equivalent of A in L_2, where b^j is the binary representation of j. Finally, concatenate all the 2^n schemes into one (consistently renaming labels) starting with the scheme α^{j_b}, where j_b is the number whose binary representation is b. □

By Theorem 8.1 we get the following theorem.

8.7. Theorem. *Assuming* $L_1 = L_2$,

(1) **c-goto**(P, L_1)^B = **c-goto**(B, L_2), *and*

(2) **c-goto**(P, L_1) = **c-goto**(B, L_2)^P.

Statements similar to Theorems 8.6 and 8.7 hold for **sticky** schemes too.

Turning to the logics and comparing Boolean logics with their propositional counterparts, we have the next result.

8.8. Lemma. *For every* $i \geq 0$,

(1) $\mathrm{GBL}_i \leq_l \mathrm{GPL}_i$, *and*

(2) $\mathrm{GB}_i \leq_l \mathrm{GP}_i$.

Proof. By induction on i. For $i = 0$, part (1) is straightforward. For instance, for the formula $A = (X_i = 0)$ and for a Boolean tuple b, let $A_b =$ true if $b_i = 0$, and false otherwise. Part (2) follows from Theorem 8.6 together with the first part. The $i > 0$ case is handled as in Lemma 6.2. □

Consequently, we have the following theorems.

8.9. Theorem. c-goto-BDL $=_l$ **c-goto-PDL.**

8.10. Theorem

(1) **c-goto-PDL**$^{\mathrm{B}}$ = **c-goto-BDL**, *and*

(2) **c-goto-PDL** = **c-goto-BDL**$^{\mathrm{P}}$.

As for the *size* of schemes and formulas, the Boolean-variable level is in general exponentially more succinct, in both the cases of **sticky** and **c-goto**. This is in accordance with a similar observation of [1] with regard to regular (sequential) PDL vs. BDL.

Due to the exponential blow-up in the translations from **sticky**(B, L), **c-reg**(B, L) and **c-goto**(B, L) to **sticky**(P, L) schemes, we get, by Theorem 7.1, the following theorem.

8.11. Theorem. *The validity problems for* **sticky-BDL**, **c-reg-BDL** *and* **c-goto-BDL** *can be solved in nondeterministic double exponential time.*

9. First-order schemes and logics

First-order sequential goto schemes are defined just like their propositional counterparts, except that the atomic operations are shown to be simple assignments:

$$\text{(ASSIGN)} \qquad l: x_i \leftarrow \sigma.$$

The program uses a tuple of variables $x = \{x_1, \ldots, x_n\}$, and the assignments and tests refer to some fixed signature; σ is a term over the signature involving variables from x. Given an appropriate language L, interpretable in first-order structures over the signature and the variables in x, a test P is simply a formula of L. (Most conventional definitions restrict tests to be either predicates or quantifier-free formulas with equality (QF) over the given signature and x, i.e., 'poor tests', but our equivalence results hold also in a 'rich test' environment.) The resulting class of schemes is denoted **goto**(Q, L).

First-order sequential (nondeterministic) regular schemes are regular expressions over an alphabet Σ_{FO} consisting of the assignment commands $x_i \leftarrow \sigma$ and the tests $P?$ for $P \in \mathrm{L}$, similar to the definition on the propositional level. The resulting class of schemes is denoted **reg**(Q, L).

The semantics is based on a first-order structure $\mathcal{A} = \langle D, P_1, \ldots, f_1, \ldots \rangle$ with a domain D and a collection of predicates P_i and functions f_i. This structure induces an interpretation π of the formulas of the logic L, according to its specific semantics, and also an interpretation for terms σ appearing in the assignment commands. The set of states S associated with such structure is the set of possible interpretations for the variable set x used in the scheme; every possible assignment of values from D to variables in x corresponds to a state. The semantics of an assignment operation is defined as

$$\rho(x_i \leftarrow \sigma) = \{(s, s[\sigma_s/x_i]) \mid s \in S\},$$

where $s[\sigma_s/x_i]$ represents a state s' similar to s except that x_i is interpreted in s' as σ_s, the evaluation of the term σ in s.

Extending ρ to arbitrary schemes is done just as on the propositional level, relying on the interpretation of assignments as atomic programs, and formulas of L appearing in tests.

The associated dynamic logics are denoted **reg**-QDL and **goto**-QDL, and they are defined analogously to the propositional logics, on the basis of first-order logic with equality (cf. [7]). Thus, the atomic formulas are predicates $P(\sigma)$ where σ is a tuple of terms. Their truth value in a state is determined by the value of $P(\sigma_s)$ in the structure. Here we use rich tests, i.e., the language of tests is **reg**-QDL (or **goto**-QDL) itself.

The classes of concurrent goto and structured schemes, and the corresponding logics, are obtained on the first-order level just as was done on the propositional level. The resulting classes of schemes are denoted **c-goto**(Q, L), **c-reg**(Q, L) and **sticky**(Q, L), for any appropriate logic L. The extension of the semantics again parallels what was done in Section 4 for propositional concurrent schemes. The corresponding logics are denoted **c-goto**-QDL, **c-reg**-QDL and **sticky**-QDL.

Again, the results of the Boolean-variable level carry over to the first-order level, and we have the following theorem.

9.1. Theorem.

(1) **reg**(Q, QF) < **c-reg**(Q, QF) < **c-reg**$^{\mathrm{K}}$(Q, QF).

(2) **c-goto**(Q, L_1) = **sticky**(Q, L_2), *assuming* $\mathrm{L}_1 = \mathrm{L}_2$.

(3) **c-goto**(Q, L_1) = **c-reg**(Q, L_2), *assuming* $\mathrm{L}_1 = \mathrm{L}_2$ *and both contain* QF.

(4) **c-goto**-QDL = **sticky**-QDL = **c-reg**-QDL = **c-reg**$^{\mathrm{K}}$-QDL.

The proof of part (3) is identical to that of Theorem 8.5, except that the basic operations are assignments, rather than just unspecified atomic letters. The requirement that the logics used for tests contain at least QF is necessary in order to enable tests to keep track of the labels. One problematic point is that the use of variables as Booleans implicitly assumes the existence of at least two distinct elements in the Herbrand universe based on the structure plus the input values of the variables. However, it is easy to test this directly, and handle the exceptional case of a singleton (or empty) universe separately.

A final remark concerns relationships with the μ-calculus. A result analogous to Theorem 6.6 holds also on the first-order level, as is shown in [18] (with CQ_μ denoting the continuous μ-calculus of [17]).

9.2. Theorem (Peleg [18]). **c-reg-QDL** $=\mathrm{CQ}_\mu$.

Appendix A. Programming "kill" in c-reg-PDL

In this appendix we prove Theorem 6.3. The described algorithm for the elimination of the kill command from **c-reg** schemes is based on the following simple observation. A program $\alpha \cap \beta$; kill can be viewed as the execution of α plus a 'test' for the feasibility of executing β. Thus, such a program should, in principle, be replaced by $(\langle\beta\rangle\mathrm{true})?;\alpha$.

In order to overcome technical difficulties, we need the following definitions and lemmas.

For every program α and every formula A in **c-reg**-PDL define a program Θ^A_α in **c-reg**-PDL by induction on α as follows:

$$\Theta^A_a = a, \qquad \Theta^A_{P?} = P?, \qquad \Theta^A_{\beta\cup\gamma} = \Theta^A_\beta \cup \Theta^A_\gamma,$$

$$\Theta^A_{\beta\cap\gamma} = (\Theta^A_\beta \cap \Theta^A_\gamma) \cup (\langle\beta\rangle A)?;\Theta^A_\gamma \cup (\langle\gamma\rangle A)?;\Theta^A_\beta,$$

$$\Theta^A_{\beta;\gamma} = \Theta^{\langle\gamma\rangle A}_\beta ; \Theta^A_\gamma, \qquad \Theta^A_{\beta^*} = (\Theta^{\langle\beta^*\rangle A}_\beta)^*.$$

A.1. Lemma. *For every program α and every formula A in* **c-reg**-PDL,

$$\rho(\Theta^A_\alpha) = \{(s, U') \mid \exists U (\emptyset \subset U' \subseteq U \wedge (s, U) \in \rho(\alpha) \wedge \mathrm{set}(U - U') \subseteq \pi(A))\}.$$

Proof. By induction on the structure of α. We present the $\subseteq$-direction of the case $\alpha = \beta ; \gamma$. Assume $(s, U') \in \rho(\Theta^A_\alpha)$. Then $(s, U') \in \rho(\Theta^{\langle\gamma\rangle A}_\beta ; \Theta^A_\gamma)$ and, by the inductive hypotheses on β and γ, there exist V', V, s_1, U'_1, U_1, s_2, U'_2, $U_2, \ldots$ s.t. $\emptyset \subset V' \subseteq V$, $\mathrm{set}(V - V') \subseteq \pi(\langle\gamma\rangle A)$, $(s, V) \in \rho(\beta)$, $V' = \{s_1, s_2, \ldots\}$, $U' = \biguplus_i U_i$, and

$$\forall i (\emptyset \subset U'_i \subseteq U_i,\ \mathrm{set}(U_i - U'_i) \subseteq \pi(A) \text{ and } (s_i, U_i) \in \rho(\gamma)).$$

Denoting $V-V'=\{q_1, q_2, \ldots\}$ and applying the definition of $\pi(\langle\gamma\rangle A)$, we get that for every q_i there exists a W_i s.t. $(q_i, W_i)\in\rho(\gamma)$ and $\text{set}(W_i)\subseteq\pi(A)$ (see Fig. 10). Choosing $W_i'=\emptyset$ for every i, we get that

$$\forall i(\emptyset=W_i'\subseteq W_i,\ \text{set}(W_i-W_i')\subseteq\pi(A) \text{ and } (q_i, W_i)\in\rho(\gamma)).$$

Denoting $U=\biguplus_i U_i \uplus \biguplus_i W_i$, we get that $\emptyset\subset U'\subseteq U$, $(s, U)\in\rho(\beta;\gamma)$ and

$$\text{set}(U-U')\left(=\text{set}\left(\biguplus_i (U_i-U_i')\uplus\biguplus_i W_i\right)\right)\subseteq\pi(A). \qquad \square$$

A.2. Lemma. *For every program α and every formula A in* **c-reg**-PDL,

$$\rho(\alpha;(\text{true}?\cup A?;\text{kill}))=\rho(\Theta_\alpha^A\cup(\langle a\rangle A)?;\text{kill}).$$

Proof. Assume $(s, U)\in\rho(\alpha;(\text{true}?\cup A?;\text{kill}))$. Then there exist $V, s_1, U_1, s_2, U_2, \ldots$ s.t. $(s, V)\in\rho(\alpha)$, $U=\biguplus_i U_i$, $V=\{s_1, s_2, \ldots\}$, and

$$\forall i((s_i, U_i)\in\rho(\text{true}?\cup A?;\text{kill})),$$

or

$$\forall i(U_i=\{s_i\}\vee(s_i\in\pi(A)\wedge U_i=\emptyset)).$$

Let us divide the discussion into two cases, according to whether U is empty or not. If U is empty, then U_i is empty and $s_i\in\pi(A)$ for every i, hence $(s, V)\in\rho(\langle\alpha\rangle A?)$ and for every i, $(s_i, U)\in\rho(\text{kill})$, so $(s, U)\in\rho(\langle\alpha\rangle A?;\text{kill})$.

In the other case, clearly $\emptyset\subset U\subseteq V$, $\text{set}(V-U)\subseteq\pi(A)$ and $(s, V)\in\rho(\alpha)$, so, by Lemma A.1, $(s, U)\in\rho(\Theta_\alpha^A)$.

The other direction is proved along similar lines. $\square$

For every program α of **c-reg**^{K}-PDL define a program α' and a formula φ_α in **c-reg**-PDL by induction on α as follows as shown in Table 1.

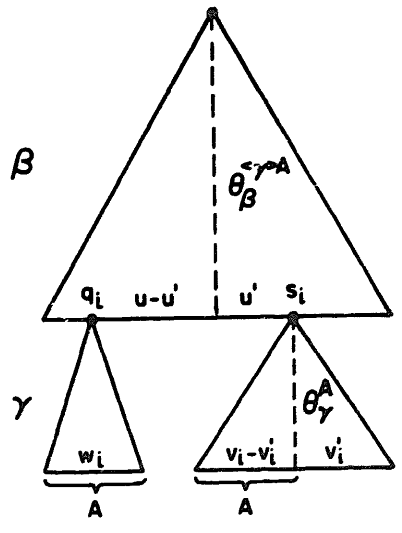

Fig. 10.

Table 1.

α	α'	φ_α
a	a	false
$P?$	$P?$	false
kill	false?	true
$\beta \cup \gamma$	$\beta' \cup \gamma'$	$\varphi_\beta \vee \varphi_\gamma$
$\beta \cap \gamma$	$(\beta' \cap \gamma') \cup \varphi_\beta?; \gamma' \cup \varphi_\gamma?; \beta'$	$\varphi_\beta \wedge \varphi_\gamma$
$\beta ; \gamma$	$\Theta^{\varphi_\gamma}_{\beta'}; \gamma'$	$\varphi_\beta \vee \langle\beta'\rangle\varphi_\gamma$
β^*	$\Theta^{\varphi_\beta}_{\beta'^*}$	$\langle\beta'^*\rangle\varphi_\beta$

A.3. Lemma. *For every program α of* **c-reg$^{\mathrm{K}}$-PDL**, $\rho(\alpha) = \rho(\alpha' \cup \varphi_\alpha?;\text{kill})$.

Proof. By induction on the structure of α. We present the case of $\alpha = \beta ; \gamma$. By the inductive hypotheses,

$$\rho(\alpha) = \rho(\beta ; \gamma) = \rho((\beta' \cup \varphi_\beta?;\text{kill});(\gamma' \cup \varphi_\gamma?;\text{kill}))$$
$$= \rho(\beta';(\gamma' \cup \varphi_\gamma?;\text{kill})) \cup \rho(\varphi_\beta?;\text{kill}).$$

The first term equals $\rho(\beta';(\text{true}? \cup \varphi_\gamma?;\text{kill});\gamma')$, or $\rho(\beta';(\text{true}? \cup \varphi_\gamma?;\text{kill})) \cdot \rho(\gamma')$, which, by Lemma A.2, equals $\rho(\Theta^{\varphi_\gamma}_{\beta'} \cup (\langle\beta'\rangle\varphi_\gamma)?;\text{kill}) \cdot \rho(\gamma')$, or $\rho(\Theta^{\varphi_\gamma}_{\beta'};\gamma' \cup (\langle\beta'\rangle\varphi_\gamma)?;\text{kill})$. Together, we get $\rho(\alpha) = \rho(\Theta^{\varphi_\gamma}_{\beta'};\gamma' \cup (\langle\beta'\rangle\varphi_\gamma \vee \varphi_\beta)?;\text{kill}) = \rho(\alpha' \cup \varphi_\alpha?;\text{kill})$. □

Finally, the desired result follows immediately from the last lemma.

A.4. Lemma. *For every formula A in* **c-reg$^{\mathrm{K}}$-PDL**, *there is an equivalent formula A' in* **c-reg-PDL**.

Proof. By induction on the structure of A. The case of $A = \langle\beta\rangle B$ is handled by choosing $A' = \langle\beta'\rangle B' \vee \varphi_\beta$, where B' is the equivalent of B from the inductive hypothesis. □

Appendix B. Completeness proof for sticky-PDL

We show that the axiom system proposed for **sticky-PDL** is complete. For this purpose we need to define an alternative semantics (ρ^s, π^s) for **sticky-PDL**, which is obtained in two stages. In the first stage we redefine $\rho(\alpha)$ directly, bypassing the tree language interpretation. This is done inductively as follows. The definition of $\rho(\$_i)$, $\rho(a(\alpha))$, $\rho(A?(\alpha))$ and $\rho(\alpha \cap \beta)$ is similar to that of ρ^{S} for trec's, given in Section 4.1. In addition,

$$\rho(\alpha \cup \beta) = \rho(\alpha) \cup \rho(\beta), \qquad \rho(\alpha ;^i \beta) = \rho(\alpha) \cdot^i \rho(\beta),$$
$$\rho(\alpha^{*_i}) = \min R(R = \rho(\$_i) \cup \rho(\alpha) \cdot^i R),$$

where $R_1 \cdot^i R_1$ is defined (for any two sets R_1, R_2 of semantic pairs as described in Section 4.1) as

$$R_1 \cdot^i R_2 = \Big\{(s, U) \mid \exists V\Big((s, V) \in R_1 \wedge \Big((V_i = \emptyset \wedge U = V)$$

$$\vee \Big(\exists s_1, W^1, \ldots, s_l, W^l\Big(V_i = \{s_1, \ldots, s_l\}$$

$$\wedge \forall j, 1 \leq j \leq l((s_j, W^j) \in R_2) \wedge U_i = \bigcup_{1 \leq j \leq l} W_i^j$$

$$\wedge \forall k, 1 \leq k \leq l, k \neq i\Big(U_k = V_k \uplus \biguplus_{1 \leq j \leq l} W_k^j\Big)\Big)\Big)\Big)\Big)\Big\},$$

and finally, for the whole program α, we let

$$\rho(\alpha) = \Big\{(s, V) \mid \exists U\Big((s, U) \in \rho(\alpha) \wedge U = \{(i_j : U_{i_j}) \mid 1 \leq j \leq k\} \wedge V = \biguplus_{1 \leq j \leq k} U_{i_j}\Big)\Big\}.$$

This definition is equivalent to our original one. In the next stage we modify this definition by replacing the multisets and the $\uplus$-operations with simple sets and $\cup$-operations throughout the definition. The resulting interpretation function ρ^s is *not* identical to our original one, even if we modify the original semantics by shrinking each multiset in the final interpretation of a program into a set. This incompatibility can be seen by looking at the following example. Consider the scheme $\alpha = \beta ;^0 c(\$_0)$, where $\beta = (a(\$_0) \cap b(\$_0))$, and assume a model in which $\rho(a) = \rho(b) = \{(s_1, s_2)\}$ and $\rho(c) = \{(s_2, s_3), (s_2, s_4)\}$. Then, in the original interpretation, α has a single trec $a(c(\$_0)) \cap b(c(\$_0))$, so that

$$\rho(\alpha) = \{(s_1, \{s_3, s_3\}), (s_1, \{s_3, s_4\}), (s_1, \{s_4, s_4\})\},$$

and the sets of states reachable from s_1 are $\{s_3\}$, $\{s_4\}$ and $\{s_3, s_4\}$. On the other hand, by the set semantics ρ^s we have $\rho^s(\beta) = \{(s_1, \{s_2\})\}$, and hence $\rho^s(\alpha) = \{(s_1, \{s_3\}), (s_1, \{s_4\})\}$, so that the set $\{s_3, s_4\}$ is not reachable from s_1.

However, we should observe that this choice of semantics for schemes does not affect the interpretation of *formulas* in the logics. The reason is that, as can be easily shown,

(1) $\rho^s(\alpha) \subseteq \rho(\alpha)$ for every program α (identifying a multiset U with the set set(U) consisting precisely of its elements), and

(2) $(s, U) \in \rho(\alpha) \Rightarrow \exists U'(U' \subseteq \text{set}(U), (s, U') \in \rho^s(\alpha))$ for every program α.

These two observations serve to prove that, for any formula A, $\pi^s(A) = \pi(A)$. This holds on both the propositional and first-order levels, and within either of **c-reg, c-goto and sticky**. Therefore the validity/satisfiability properties of **sticky**-PDL formulas are the same under π and π^s, and we may assume the (ρ^s, π^s) semantics in what follows.

Assume we are given a consistent formula A_0, for which we have to show the existence of a satisfying model. We show this first for formulas A_0 using only atomic tests $P?$ in programs. Further, we assume that all occurrences of subprograms $a(\beta)$ or $B?(\beta)$ in A_0 are such that β is $\$_i$ for some i. Clearly, any A_0 may be transformed into an equivalent formula in this form, using Axioms (A3) and (A5). The model is constructed on the basis of the Fischer-Ladner closure of A_0, FL, defined inductively by

(1) $A_0 \in \mathrm{FL}$,
(2) $A \vee B \in \mathrm{FL} \Rightarrow A, B \in \mathrm{FL}$,
(3) $\neg A \in \mathrm{FL} \Rightarrow A \in \mathrm{FL}$,
(4) $\langle\alpha\rangle A \in \mathrm{FL} \Rightarrow A_{i_1}, \ldots, A_{i_k} \in \mathrm{FL}$,
(5) $\langle B?(\$_i)\rangle\{(i : A_i)\} \in \mathrm{FL} \Rightarrow A_i \wedge B \in \mathrm{FL}$,
(6) $\langle\alpha \cup \beta\rangle A \in \mathrm{FL} \Rightarrow \langle\alpha\rangle A \vee \langle\beta\rangle A \in \mathrm{FL}$,
(7) $\langle\alpha \cap \beta\rangle A \in \mathrm{FL} \Rightarrow \langle\alpha\rangle A \wedge \langle\beta\rangle A \in \mathrm{FL}$,
(8) $\langle\alpha ;^i \beta\rangle A \in \mathrm{FL} \Rightarrow \langle\alpha\rangle A[i/\langle\beta\rangle A] \in \mathrm{FL}$,
(9) $\langle\alpha^{*_i}\rangle A \in \mathrm{FL} \Rightarrow \langle\alpha\rangle A, \langle\alpha\rangle A[i/\langle\alpha^{*_i}\rangle A] \in \mathrm{FL}$,
(10) $\langle\alpha\rangle A \in \mathrm{FL} \Rightarrow \langle\alpha\rangle(A - \{(i : A_i)\}) \in \mathrm{FL}$ if $\$_i \notin \alpha$.

Throughout, we identify $\neg\neg A$ with A and $A \wedge B$ with $\neg(\neg A \vee \neg B)$.

Let $\mathrm{FL} = \{A_1, \ldots, A_p\}$. An *atom* is a consistent conjunction $X = \bigwedge_{1 \le i \le p} B_i$, where $B_i \in \{A_i, \neg A_i\}$ for every $1 \le i \le p$. (Here consistency is w.r.t. the axiom system of Section 7.) Let $\mathcal{W} = \{X \mid X \text{ atom}\}$ be our world (or 'state space').

Following conventions set in [23], we write $A \le B$ to denote $\vdash A \supset B$, and let $A^+ = \{X \mid X \in \mathcal{W}, X \wedge A \text{ is consistent}\}$. For a set of atoms $U \subseteq \mathcal{W}$, let $\varphi_U = \bigvee_{X \in U} X$.

We now collect some well-known properties concerning these notions.

B.1. Lemma. *For every atom $X \in \mathcal{W}$ and formula $A \in \mathrm{FL}$, the following conditions are equivalent*:
(1) $X \le A$ (*or* $\vdash X \supset A$);
(2) $X \in A^+$ (*or* $X \wedge A$ *is consistent*);
(3) A *appears positively in* X.

B.2. Lemma
(1) *For every two distinct atoms X and Y, $X \wedge Y$ is inconsistent.*
(2) $(\bigvee_{X \in \mathcal{W}} X) \equiv \mathrm{true}$.

B.3. Lemma. *For every formula A, $\vdash A \supset \varphi_{A^+}$. Furthermore, if $A \in \mathrm{FL}$, then $\vdash A \equiv \varphi_{A^+}$.*

B.4. Lemma. *For every formula $A \in \mathrm{FL}$,*
(1) $(\neg A)^+ = \overline{(A^+)}$, *and*
(2) $(A \vee B)^+ = A^+ \cup B^+$,
where $\bar{U}$ denotes $\mathcal{W} - U$, for any set of atoms U.

B.5. Lemma. *For every set of atoms $U \subseteq \mathcal{W}$,*
(1) $\vdash \varphi_{\bar{U}} \equiv \neg\varphi_U$, *and*
(2) $(\varphi_U)^+ = U$.

B.6. Lemma. *For all formulas A, B, $A \leqslant B \Rightarrow A^+ \subseteq B^+$.*

Now the model we construct is $\mathcal{M} = \langle \mathcal{W}, \pi, \rho^s \rangle$ where π and ρ^s are defined by

$$\pi(P) = P^+,$$
$$\rho^s(a) = \{(X, \{Y\}) \mid X \in (\langle a(\$_1) \rangle \{(1: Y)\})^+\},$$

for every atomic proposition P and every atomic program a appearing in A_0. Throughout the sequel we shall write $\rho(\alpha)$ instead of $\rho^s(\alpha)$, for simplicity. For every labeled set $U = \{(i_j : U_{i_j}) \mid 1 \leqslant j \leqslant n\}$ denote $\varphi_U = \{(i_j : \varphi_{U_{i_j}}) \mid 1 \leqslant j \leqslant n\}$.

The following lemma is proved by simple structural induction.

B.7. Lemma. *If $\vdash A \equiv B$ and D is obtained from C by replacing some occurrences of A by B, then $\vdash C \equiv D$.*

B.8. Lemma. *For every $X \in \mathcal{W}$, $U_{i_1}, \ldots, U_{i_k} \subseteq \mathcal{W}$ and every program α appearing in A_0,*

$$X \wedge \langle \alpha \rangle \varphi_U \textit{ is consistent } \Rightarrow \exists U'((X, U') \in \rho(\alpha) \wedge U' \subseteq U).$$

Proof. By induction on α. We give the most involved case which is $\alpha = \beta^{*_l}$. Assume $X \wedge \langle \beta^{*_l} \rangle \varphi_U$ is consistent. Denote $B = \langle \beta^{*_l} \rangle \varphi_U$. Let V be the smallest set of atoms s.t. $U_l \subseteq V$ and for every atom Y, $Y \in (\langle \beta \rangle \varphi_U[l/\varphi_U])^+ \Rightarrow Y \in V$. An alternative, equivalent definition is $V = V_{2^n}$, where $V_0 = U_l$ and $V_{i+1} = V_i \cup (\langle \beta \rangle \varphi_U[l/\varphi_{V_i}])^+$ for every $i \geqslant 0$. Denote $A_l = \varphi_V$ and $A_{i_j} = \varphi_{U_{i_j}}$, for every $1 \leqslant j \leqslant n$, $i_j \neq l$. We need the following easy claims:

B.8.1. Claim. *$\langle \beta \rangle A \wedge \neg \varphi_V$ is inconsistent.*

B.8.2. Claim. *$\vdash \langle \beta^{*_l} \rangle A \supset \varphi_V$.*

B.8.3. Claim. *$B^+ \subseteq V$.*

We also need the following claim.

B.8.4. Claim. *For every $i \geqslant 0$ and every atom $Z \in V_i$ there exists a $U' \subseteq U$ s.t. $(Z, U') \in \rho(\beta^{*_l})$.*

Proof. By induction on i.

Basis ($i = 0$): If $Z \in U_l$, then $(Z, U_l) \in \rho(\$_l)$, so $(Z, U) \in \rho(\beta^{*_l})$.

Induction ($i+1$): $Z \in V_{i+1}$ means $Z \in V_i$ or $Z \in (\langle \beta \rangle \varphi_U[l/\varphi_{V_i}])^+$. In the first case the result is immediate by inductive hypothesis on i. Now assume $Z \wedge \langle \beta \rangle \varphi_U[l/\varphi_{V_i}]$ is consistent. By the main inductive hypothesis on β there exists a $U'_{i_j} \subseteq U_{i_j}$ (for $1 \leqslant j \leqslant n$, $i_j \neq l$) and $U'_l \subseteq V_i$ s.t. $(Z, U') \in \rho(\beta)$. Let $U'_l = \{Y_1, \ldots, Y_k\}$. By the inductive hypothesis on i, for every $1 \leqslant m \leqslant k$ there exists a $W^m \subseteq U$ s.t. $(Y_m, W^m) \in \rho(\beta^{*_l})$. Choose $T_l = \bigcup_{1 \leqslant m \leqslant k} W^m_l$ and $T_{i_j} = U'_{i_j} \cup \bigcup_{1 \leqslant m \leqslant k} W^m_{i_j}$ (for $1 \leqslant j \leqslant n$, $i_j \neq l$), and then $T \subseteq U$ and $(Z, T) \in \rho(\beta; \beta^{*_l}) \subseteq \rho(\beta^{*_l})$. □

Now the proof of the β^{*_l} case is completed; since $X \in B^+$, by Claim B.8.3, $X \in V$, so $X \in V_{2^n}$, and, by Claim B.8.4, $\exists U'(U' \subseteq U \wedge (X, U') \in \rho(\beta^{*_l}))$. □

B.9. Lemma. *For every $\langle\alpha\rangle A \in \mathrm{FL}$ and every $X \in \mathcal{W}$,*

$$X \wedge \langle\alpha\rangle A \text{ is consistent} \Leftrightarrow \exists U(U \subseteq \overline{A^+} \wedge (X, U) \in \rho(\alpha)).$$

Proof. The ($\Rightarrow$)-direction follows immediately from Lemma B.8, viewing A as φ_{A^+}. The other direction is proved by induction on α. Here we give the following cases.

Case $\alpha = \beta \cap \gamma$: Recall that both $\langle\beta\rangle A, \langle\gamma\rangle A \in \mathrm{FL}$. Assuming $\exists U(U \subseteq \overline{A^+} \wedge (X, U) \in \rho(\beta \cap \gamma))$ we get that there are V, W s.t. $U = V \cup W$, $(X, V) \in \rho(\beta)$ and $(X, W) \in \rho(\gamma)$. Since $V, W \subseteq U \subseteq \overline{A^+}$, by inductive hypothesis, both $X \wedge \langle\beta\rangle A$ and $X \wedge \langle\gamma\rangle A$ are consistent, hence both $\langle\beta\rangle A$ and $\langle\gamma\rangle A$ appear positively in X, so $X \wedge \langle\beta\rangle A \wedge \langle\gamma\rangle A$ is consistent, and by Axiom (A9) $X \wedge \langle\beta \cap \gamma\rangle A$ is consistent too.

Case $\alpha = \beta ;^l \gamma$: Denote $B = \langle\gamma\rangle A$. Both B and $\langle\beta\rangle A[l/B]$ are in FL. Assume $\exists U(U \subseteq \overline{A^+} \wedge (X, U) \in \rho(\beta ;^l \gamma))$. Then there are $V_l = \{Y_1, \ldots, Y_k\}$, $V_{i_j} \subseteq U_{i_j}$ (for $1 \leq j \leq n$, $i_j \neq l$), $W^1, \ldots, W^k$ such that $(X, V) \in \rho(\beta)$, for every $1 \leq m \leq k$, $(Y_m, W^m) \in \rho(\gamma)$, and $U_l = \bigcup_{1 \leq m \leq k} W_l^m$ and $U_{i_j} = V_{i_j} \cup \bigcup_{1 \leq m \leq k} W_{i_j}^m$ for $1 \leq j \leq n$, $i_j \neq l$. Hence, $W^m \subseteq U$ for every $1 \leq m \leq k$. By the inductive hypothesis on γ, for every $1 \leq m \leq k$, $Y_m \wedge \langle\gamma\rangle A$ is consistent, so $Y_m \in B^+$. Hence, $V_l \subseteq B^+$, and $V \subseteq \overline{A^+}[l/B^+]$ and by the inductive hypothesis on β, $X \wedge \langle\beta\rangle A[l/B]$ is consistent. By Axiom (A10), $X \wedge \langle\beta ;^l \gamma\rangle A$ is consistent too. □

B.10. Lemma. *For every $A \in \mathrm{FL}$, $\pi(A) = A^+$.*

Proof. By induction on the structure of A. The claim is immediate for atomic propositions by the definition of π, the $B \vee C$, $\neg B$ cases follow from Lemma B.4(2), (1) respectively, and the case of $\langle\alpha\rangle B$ follows from Lemma B.9. □

Now the satisfiability of A_0 follows from the fact that since A_0 is consistent, A_0^+ is nonempty, as $\varphi_{A_0^+} \equiv A_0 \not\equiv$ false.

Generalization of the proof to formulas with rich tests, and not only propositional, is obtained by defining a sequence of sets of formulas L_i, where L_0 allows only atomic tests, and L_{i+1} allows formulas of L_i as tests. Then the result is proved by induction on i, using the described proof process in each level, and the previous inductive step to account for the test case in Lemmas B.8 and B.9 of the current level.

Acknowledgment

I would like to thank David Harel for many stimulating discussions and valuable suggestions, and an anonymous referee for his helpful comments.

References

[1] K. Abrahamson, Decidability and expressiveness of logics of processes, Ph.D. Thesis, Univ. of Washington, 1980.

[2] A.K. Chandra, The power of parallelism and nondeterminism in programming, in: *Proc. IFIP* (1974) 461-465.

[3] A.K. Chandra, D.C. Kozen and L.J. Stockmeyer, Alternation, *J. ACM* **28** (1981) 114-133.

[4] M.J. Fischer and R.E. Ladner, Propositional dynamic logic of regular programs, *J. Comput. System Sci.* **18** (1979) 194-211.

[5] S.A. Greibach, *Theory of Program Structures: Schemes, Semantics, Verification*, Lecture Notes in Computer Science **36** (Springer, Berlin, 1975).

[6] F. Gecseg and M. Steinby, *Tree Automata* (Akademiai Kiado, Budapest, 1985).

[7] D. Harel, Dynamic logic, in: D. Gabbay and F. G. Guenthner, eds., *Handbook of Philosophical Logic II* (Reidel, Dordrecht, 1984).

[8] D. Harel, On folk theorems, *Comm. ACM* **23** (1980) 379-389.

[9] D. Harel and D.C. Kozen, A programming language for the inductive sets, and applications, *Inform. and Control* **63** (1984) 118-139.

[10] D. Harel and R. Sherman, Propositional dynamic logic of flowcharts, *Inform. and Control* **64** (1985) 119-135.

[11] J.E. Hopcroft and J.D. Ullman, *Formal Languages and their Relation to Automata* (Addison-Wesley, Reading, MA, 1969).

[12] D.C. Kozen, Results on the propositional μ-calculus, in: *Proc. 9th ICALP*, Lecture Notes in Computer Science **140** (Springer, Berlin, 1982) 348-359.

[13] R.M. Karp and R.E. Miller, Parallel program schemata, *J. Comput. System Sci.* **3** (1969) 147-195.

[14] Z. Manna, The correctness of nondeterministic programs, *Artificial Intelligence* **1** (1970) 1-26.

[15] A.R. Meyer and R. Parikh, Definability in dynamic logic, *J. Comput. System Sci.* **23** (1981) 279-298.

[16] R. Parikh, Propositional game logic, in: *Proc. 24th IEEE Symp. on Foundations of Computer Science* (1983) 195-200.

[17] D.M.R. Park, Finiteness is mu-ineffable, *Theoret. Comput. Sci.* **3** (1976) 173-181.

[18] D. Peleg, Concurrent dynamic logic, *J. ACM* **34** (1987) 450-479.

[19] D. Peleg, Communication in concurrent dynamic logic, *J. Comput. System Sci.* **35** (1987) 23-58.

[20] G. Slutzki, Alternating tree automata, *Theoret. Comput. Sci.* **41** (1985) 305-318.

[21] J.W. Thatcher, Tree automata: an informal survey, in: A.V. Aho, ed., *Currents of the Theory of Computing* (Prentice-Hall, Englewood Cliffs, NJ, 1973).

[22] M. Tiomkin, Extensions of propositional dynamic logic, Ph.D. Thesis, The Technion, Haifa, 1983.

[23] D. C. Kozen and R. Parikh, An elementary proof of the completeness of PDL, *Theoret. Comput. Sci.* **14** (1982) 113-118.