NONDETERMINISTIC PROPOSITIONAL DYNAMIC LOGIC WITH INTERSECTION IS DECIDABLE

Ryszard Danecki
Institute of Mathematics, Polish Acad. of Sci.
Mielżyńskiego 27/29, 61-725 Poznań, Poland

INTRODUCTION

Propositional Dynamic Logic (PDL) of [FL] defines meaning of programs in terms of binary input-output relations. Basic regular operations on programs are interpreted as superposition, union, and reflexive-transitive closure of relations. The intersection, cf. [H] , is a binary program forming functor $a \cap b$ with the meaning given by the set-theoretical intersection of relations corresponding to programs a and b. By adding intersection of programs to PDL we obtain a programming logic called PDL with intersection. Harel [H] has proved that the problem of whether or not a formula of PDL with intersection has a deterministic model is highly undecidable (Σ^1_1-hard). The present paper shows that in the general case (nondeterministic models allowed) the satisfiability problem for PDL with intersection is decidable in time double exponential in the length of the formula tested. In comparison with PDL with strong loop predicate [D] , this is more powerful and interesting example of a logic which is decidable in contrast to its deterministic case and despite the lack of finite and even tree model properties.

The entire paper is devoted to the proof of the result which reduces the satisfiability problem to the emptiness problem for special tree automata in the sense of [R70] . This is done in two stages. The first stage proves an analogue of a tree model property for PDL with intersection: a formula has a model iff it has a special model, that is a model which can be represented by a particular, usually infinite labelled tree. The second stage shows how a special tree automaton can recognize trees that represent special models of a given formula. All that is technically organized as follows.

The first two sections present syntax, semantics and all graph notions needed to define special models and their tree representations. In Section 3, executions of programs with intersections are described in terms of well nested graphs, that is, parallel-sequential compositions of paths. Then, in Section 4, special models of a formula are obtained as tree-like compositions of well nested graphs. The first stage of our proof ends with the equivalence: a formula has a model iff it has a consistent validation tree, where the latter is a tree representation of a special model.

The second stage is dominated by a problem which is the main difficulty in every proof of that type: a tree automaton must be able to recognize if any node in which a formula $\langle a \rangle q$ is claimed to be false is not a beginning of a succesful execution of the program $a;q?$. To solve this problem we describe executions of programs in terms of finite state concurrent processes, Section 5, and then seek a way to simulate them by tree automata. The simulation becomes possible due to the following facts. The processes are well nested and admit an appropriate decomposition (Lemma 5.2). Special models have cutpoints which sequentialize processes in such a way that parallel transitions are necessary only between pairs of nodes which can be represented by a single node of the corresponding validation tree (the idea of coupling, Section 6). Finally, the whole simulation can be expressed as the existence of an additional labelling of a validation tree that satisfies some local conditions (Section 7).

Once the main difficulty is solved, all what remains is an easy construction of a special tree automaton which recognizes the set of consistent validation trees of a given formula (Section 8). Recall, that the emptiness problem for special tree automata is solvable in time polynomial of the number of states [R70].

1. SYNTAX AND SEMANTICS

Let A, B, C, ..., be atomic programs, and P, Q, R, ..., atomic formulae. If a, b are programs and p, q are formulae, then $a;b$, $a \cup b$, $a \cap b$, a^*, $p?$ are programs, and $\langle a \rangle p$, $\neg p$ are formulae. As usual we can define $p \& q \equiv \langle p? \rangle q$, $[a] p \equiv \langle a \rangle \neg p$, <u>true</u> $\equiv p \vee \neg p$. Formulae are interpreted in classical PDL structures of the form $\mathcal{M} = (X, \models, \prec\succ)$, where X is a nonempty set of nodes, $\models$ is a satisfiability relation for atomic formulae, $\models \subset X \times \{P, Q, R, \ldots\}$, the set of triples $\langle\,\rangle \subset X \times \{A, B, C, \ldots\} \times X$ defines binary relations $\langle A \rangle \subset X \times X$ giving meaning to every atomic program A. $\mathcal{M}$ is said to be deterministic if for every atomic program A, $\langle A \rangle$ is a function, i. e. $x \langle A \rangle y$ and $x \langle A \rangle z$ imply $y = z$. Relations $\models$ and $\langle\,\rangle$ are extended to arbitrary formulae and porograms as follows: $x \models \neg p$ iff not $x \models p$, $x \models \langle a \rangle p$ iff $\exists\, y \in X$: $x \langle a \rangle y$ and $y \models p$, $\langle a;b \rangle = \langle a \rangle \cdot \langle b \rangle$ (superposition of relations), $\langle a \cup b \rangle = \langle a \rangle \cup \langle b \rangle$, $\langle a \cap b \rangle = \langle a \rangle \cap \langle b \rangle$, $\langle a^* \rangle = \langle a \rangle^*$ (transitive and reflexive closure of $\langle a \rangle$), $\langle p? \rangle = \{(x, x)\colon\ x \models p\}$. $\mathcal{M}$ is a model for a formula p, in short $\mathcal{M} \models p$, if $x \models p$ for some node x of $\mathcal{M}$. A formula is satisfiable if it has a model.

Notation: for sets, $|X|$ and $\underline{P}(X)$ stand for the cardinality and the powerset of X, respectively. For formulae, $|p|$ is the length of p.

2. WELL NESTED AND SPECIAL GRAPHS

By a Δ-graph we mean a directed graph with edges labelled with elements of Δ. Formally, it is a pair $G = (X, E)$, where X is a set of nodes and $E \subset X \times \Delta \times X$ is the set of edges. We say simply "graphs" if the exact form of labels is inessential. For graphs G and G′ every of which has distinguished two nodes, the origin and the sink, we define operations of sequential G;G′ and parallel G//G′ compositions. The graph G;G′ results from disjoint copies of G and G′ by glueing the sink of G with the origin of G′, and the graph G//G′ is obtained by glueing the origin of G with the origin of G′ and the sink of G with the sink of G′. In both cases the origin of G and the sink of G′ become the origin and the sink of the new graph, respectively.

By well nested Δ-graphs we mean the smallest class of Δ-graphs closed under sequential and parallel compositions and containing all single node graphs with no edges (origin equals the sink), and all single edge graphs. In the latter case, the beginning and the end of the edge are the origin and the sink of the graph, respectively, and there are no other nodes. Observe, that a well nested graph may contain loops since a parallel composition with a single node graph glues origin with sink.

Now, assume that every graph has a distinguished node called a root, and if the graph is well nested this is the origin. The operation of grafting G' on G at a node x is the glueing the root of G' with the node x of G. The root of G becomes to be the root of the new graph. The closure of well nested Δ-graphs on a finite or infinite number of grafting operations gives the class of special graphs. (Formal definition in terms of type-2 trees.) If during construction no more than k grafts are made at each particular node, then we say that the resulting special graph has degree k.

The above inductive definitions suggest a natural way in which well nested and special graphs can be represented by trees. The idea is plain, however, very important is the notation and terminology introduced below. By a n-ary tree we mean a tree in which every node has no more than n immediate successors (sons). A root has no predecessors and a leaf has no successors. In a (2k+2)-ary tree T, immediate successors of a node u will be denoted by: left son(u), right son(u), i-th left son(u), i-th right son(u), $i = 1, \ldots, k$. The first two sons are distinguished and play a special role. For a node $u \in T$, T_u is the full subtree of T consisting of u and all its successors, while t_u stands for the restricted subtree with the root u, consisting of u and only those its successors which are reachable by left and right sons.

By a type-1 tree over Δ we mean a finite binary tree t in which every node $u \in t$ is labelled with sign(u)$=\{;, //, \text{"equal"}\} \cup \Delta$ in such a way that if sign(u) $\in \Delta \cup \{\text{"equal"}\}$ then u is a leaf, and if sign(u) $\in \{;, //\}$, then both left and right sons of u are defined. We write $t = t';t''$ or $t = t'//t''$ if sign(root(t)) = ; or //, respectively, and the left (right) son of the root of t is the root of t' (t'').

In an obvious way, every type-1 tree t over Δ defines a well nested Δ-graph $G(t)$. If t consists of a single leaf, then $G(t)$ is a single node, or a single edge $\{(x, \delta, y)\}$ graph with $x \neq y$, depending on whether sign(root(t)) = "equal" or δ, $\delta \in \Delta$. This is extended to all type-1 trees by $G(t;t') = G(t);G(t')$, $G(t//t') =$ $= G(t)//G(t')$.

For technical reasons of Sections 6 and 7, it is convenient to define $G(t)$ in the following equivalent form. For every $u \in t$, the two pairs $(u, 1)$, $(u, 2)$ will be called places. The relation of elementary equivalence of places $\sim$ is defined as follows: (a1): if sign(u) = "equal", then $(u, 1) \sim (u, 2)$, (a2): if v=left son(u), w=right son(u), and sign(u)= ; , then $(u, 1)\sim(v, 1)$, $(v, 2)\sim(w, 1)$, $(w, 2)\sim(u, 2)$, (a3): if v, w are as above and sign(u)=// , then $(u, 1)\sim(v, 1)\sim(w, 1)$, $(u, 2)\sim(v, 2)\sim(w, 2)$. Let $\approx$ be the reflexive and transitive closure of $\sim$, and let ui, $i\in\{1, 2\}$, denote the equivalence class $[(u, i)]_{\approx}$. It is easy to see, that $G(t)=(X, E)$ where X is the set of equivalence classes of places in $\approx$, i. e. $X=(t\times\{1, 2\})/_{\approx}$, and E is the smallest subset of $X\times\triangle\times X$ such that for every $u\in t$, if $\text{sign}(u)=\sigma$, then $(u1, \sigma, u2)\in E$. This definition enables us to see both t and G(t) in one picture, and this is very useful in proofs (cf. Fig. 1).

For nodes x, y of G(t) and u of t, we say "x falls in u" instead of $x\in\{u1, u2\}$, and "x, y are coupled by u" instead of $\{x, y\}\subset\{u1, u2\}$. Observe that the origin and the sink of G(t) are

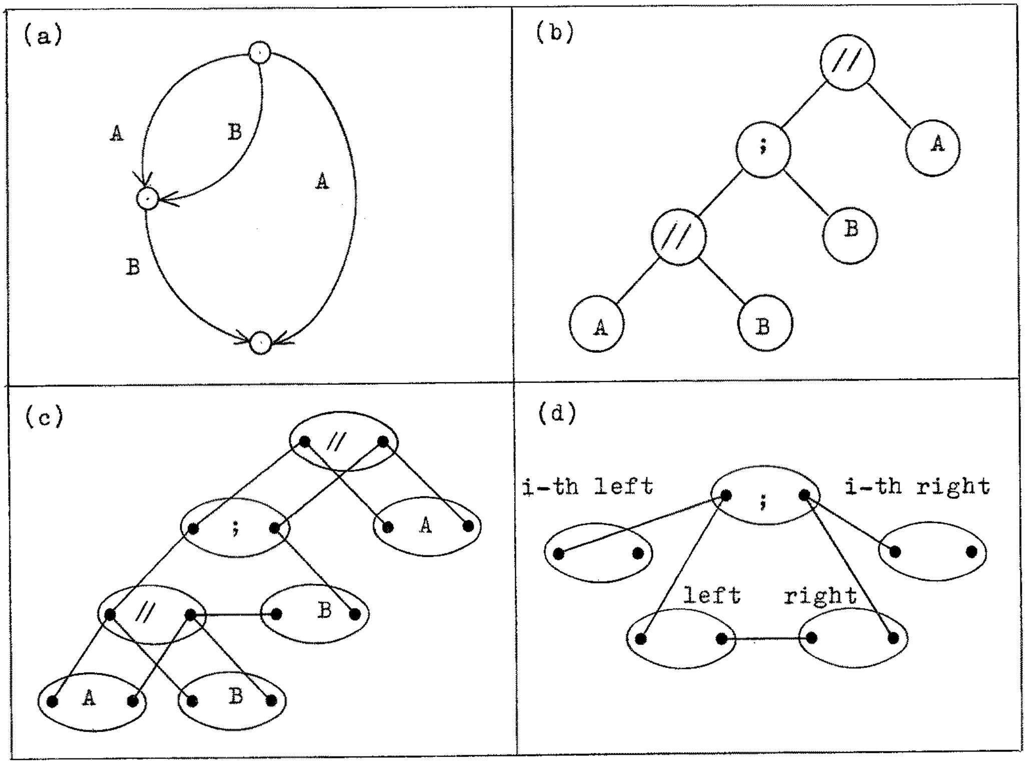

Fig. 1. A well nested graph (a), the corresponding type-1 tree (b), and how to see both of them in one picture (c). In (c) black dots mean places and elementarily equivalent places are connected by straight line segments. The elementary equivalence of places for i-th left (right) sons is presented by (d).

coupled by the root of t, and so on for subgraphs and subtrees.

By a (2k+2)-ary type-2 tree over Δ we mean a (2k+2)-ary tree T (finite or infinite) with nodes $u \in T$ labelled with sign(u) belonging to $\{;, //, \text{"equal"}\} \cup \Delta$ in such a way that every restricted subtree t_u is a type-1 tree over Δ. Remark: if sign(u) $\in \Delta \cup \{\text{"equal"}\}$ then left and right sons of u are undefined, but i-th sons may exist.

Every type-2 tree T over Δ defines a special Δ-graph G(T) according to the following rule. If t, t′ are type-1 trees with roots u, u′, respectively, and v is some node of t, then a new tree T which results from t and t′ by adding a link i-th left son(v)$=$u′, for some $1 \leqslant i \leqslant n$, defines a special graph G(T) which results from G(t) and G(t′) by grafting G(t′) on G(t) at the node v1. The link i-th right son(v)$=$u′ means that G(t′) is grafted at the node v2 of G(t). In general, the construction of G(T) may require an infinite number of grafts, so it is convenient to define G(T) formally by means of places.

If T is type-2 tree, then $T \times \{1, 2\}$ is the set of places, and G(T) is defined as for type-1 trees with the exception that the elementary equivalence of places $\sim$ includes the following additional cases. For every $u \in T$, if v$=$i-th left son(u), w$=$i-th right son(u), then (u, 1)$\sim$(v, 1), (u, 2)$\sim$(w, 1), (cf. Fig. 1 (d)). It should be clear, that for every special graph G of degree k there exists a (2k+2)-ary type-2 tree T such that G$=$G(T). The tree T is usually not unique and in general, a (2k+2)-ary type-2 tree may define a special graph of unbounded degree.

All the above notions will be applied to graphs in which both edges and nodes are labelled. A Δ,Σ-graph G$=$(X, E, F) is a Δ-graph with a node labelling function $F: X \longrightarrow \Sigma$. The introduction of node labels induces the following minor changes and exceptions in definitions. The glueing is allowed if the nodes involved have the same label. This means that parallel and sequential compositions, and grafting are from now on partial operations. For example, G;G′ exists if the sink of G has the same label as the origin of G′.

A type-1 (resp. type-2) tree over Δ,Σ is a type-1 (resp. type-2) tree over Δ such that every node u has two additional labels F1(u), F2(u)$\in \Sigma$. The additional labelling must satisfy the following condition: every elementary equivalence of places $(u, i) \sim (v, j)$ implies the equality of labels $Fi(u) = Fj(v)$, $i, j = 1, 2$. Thus, any such tree T over Δ, Σ defines a Δ, Σ-graph $G(T) = (X, E, F)$ in which nodes are labelled as follows: $F(ui) = Fi(u)$, for every $u \in T$, i, j from $\{1, 2\}$.

3. EXECUTIONS OF PROGRAMS: STATIC DESCRIPTION

A PDL program can be treated as a regular expression which defines a set of words over the alphabet containing atomic programs and tests. These words are often called execution sequences, since they describe all possible runs of the program. If we map an execution sequence into a structure, we obtain a path that connects nodes semantically related by the program. A similar description can be done for programs with intersections, however, we must replace sequences by well nested graphs.

Let Δ be a finite set of atomic programs and let Σ be a powerset of some finite set of formulae Φ. Consider a program a with atomic programs from Δ and tests from Φ. The set $ET(a)$ of all execution trees of the program a is defined by the following induction.

For every atomic program $A \in \Delta$, $ET(A)$ is the set of all single leaf $\{u\}$ type-1 trees over Δ, Σ such that $sign(u) = A$.

For every formula $q \in \Phi$, $ET(q?)$ is the set of all type-1 trees over Δ, Σ consisting of a single node u with sign(u)= "equal" and $q \in F1(u) = F2(u)$.

$ET(a;b) = \{t;t' : t \in ET(a),\ t' \in ET(b)\}$
$ET(a \cap b) = \{t//t' : t \in ET(a),\ t' \in ET(b)\}$
$ET(a \cup b) = ET(a) \cup ET(b)$
$ET(a^*) = ET(\underline{true?}) \cup ET(a) \cup \{t;t' : t \in ET(a),\ t' \in ET(a^*)\}$.

Any $G(t)$ with $t \in ET(a)$ is called an execution graph of the program a. Such a graph has edges labelled with atomic programs from Δ and every node x labelled with a set of formulae $F(x) \subset \Phi$.

By a homomorphism restricted to Δ and Φ from some execution graph G into a PDL structure $\mathcal{M}$ we mean a mapping $h: G \longrightarrow \mathcal{M}$ such that for every atomic program $A \in \Delta$ if there is an edge (x, A, y)

in G, then $h(x) \langle A \rangle h(y)$ in $\mathcal{M}$, and for every formula $q \in \Phi$, $q \in F(x)$ in G iff $h(x) \models q$ in $\mathcal{M}$. If Δ and Φ are not explicitly specified, it means that the homomorphism is restricted to atomic subprograms and all subformulae of a program or a formula in question.

Lemma 3. 1 For every structure $\mathcal{M}$ and every program a, $x \langle a \rangle y$ in $\mathcal{M}$ iff there exist an execution graph G of a and a homomorphism $h: G \longrightarrow \mathcal{M}$ which maps the origin of G on x and the sink of G on y. The homomorphism h is restricted to atomic programs and all formulae contained in a. □

4. VALIDATION TREES AND SPECIAL MODELS

Let p be a formula and let Δ be the set of its atomic programs and Φ the set of all its subformulae. Assume further that $\Sigma = \underline{P}(\Phi)$ and that $\langle a_i \rangle p_i$, $i = 1, \ldots, k$, are all diamond subformulae of p.

By a validation tree of a formula p we mean any (2k+2)-ary type-2 tree T over Δ, Σ with labellings sign, F1, F2, such that the following conditions are satisfied: (c1): $p \in F1(root(T))$, (c2): for every $u \in T$, $i = 1, 2$, $Fi(u)$ is a consistent set of formulae, that is for every subformula $\neg q$ of p, $q \in Fi(u)$ iff $\neg q \notin Fi(u)$, (c3): if $\langle a_i \rangle p_i \in F1(u)$ for some $u \in T$, then $v =$ i-th left son(u) is defined and the restricted subtree t_v is an execution tree of the program $a_i;p_i?$, i. e. $t_v \in ET(a_i;p_i?)$, (c4): if $\langle a_i \rangle p_i \in F2(u)$ for some $u \in T$, then $w =$ i-th right son(u) is defined and t_w is in $ET(a_i;p_i?)$.

The set of all validation trees of p will be denoted by VT(p), and every G(T) with $T \in VT(p)$ is a validation graph of p. Any node x of G(T) is labelled with a consistent set F(x) of subformulae of p. By means of G(T), the validation tree T defines a special PDL structure $\mathcal{M}(T)$ which has the same set of nodes and edges as G(T) and its satisfaction relation $\models$ is defined for any atomic formula q as follows:

(4.1) $x \models q$ in $\mathcal{M}(T)$ iff $q \in F(x)$ in G(T), for any node x of G(T).

If (4.1) holds for every subformula q of p, then we say that T is a consistent validation tree for p. In this case $\mathcal{M}(T)$ is a special model for p.

Lemma 4. 1 A formula p has a model iff it has a special model, that is, iff there exists a consistent validation tree for p.

Proof: Let p be a formula and $\mathcal{M} = (X, \models, \prec\succ)$ a structure. Suppose that $x_o \models p$ for some $x_o \in X$. Now we are going to show how to construct a consistent validation tree T of p together with a homomorphism $h: G(T) \longrightarrow \mathcal{M}$. Let $\langle a_i \rangle p_i$, $1 \leq i \leq k$, be all diamond subformulae of p. For every $1 \leq i \leq k$ and every $x \in X$ with $x \models \langle a_i \rangle p_i$ we choose a tree $t_{ix} \in ET(a_i; p_i?)$ and a homomorphism $h_{ix}: G(t_{ix}) \rightarrow \rightarrow \mathcal{M}$ which maps the origin of $G(t_{ix})$ on x (Lemma 3.1). Let $t_o = = \{u\}$ be a single node tree with sign(u) = "equal", $F1(u) = F2(u) = = \{ q : x_o \models q$, where q is a subformula of p$\}$. There is an obvious homomorphism $h_o: G(t_o) \longrightarrow \mathcal{M}$ with $h_o(u1) = x_o$.

To construct T, we start with t_o as the root of T, and regard it as already constructed part of T. For every node u in the already constructed part of T, if $\langle a_i \rangle p_i \in F1(u)$ and $h(u1) = x$, then we extend the constructed part of T by taking a copy of t_{ix} and defining the link i-th left son(u)=root(t_{ix}). Using h_{ix} we extend the homomorphism h to the current part of G(T). Similarly for $\langle a_i \rangle p_i \in F2(u)$, $h(u2) = x$, but the link is i-th right son(u)= root(t_{ix}). We repeat this procedure until (c3) and (c4) are satisfied. In the limit we obtain a validation tree T with a homomorphism $h: G(T) \longrightarrow \mathcal{M}$.

It remains to show that T is consistent. The proof that (4.1) holds for every subformula q of p is by structural induction. Let us consider only the most interesting case of a diamond subformula $q = = \langle a_i \rangle p_i$ of p. Assume that (4.1) holds for every formula contained in q. By (c3), (c4), and Lemma 3.1, $q \in F(x)$ in G(T) implies $x \models q$ in $\mathcal{M}(T)$. It remains to prove that $x \models q$ in $\mathcal{M}(T)$ implies $q \in F(x)$ in G(T). Indeed, if $x \models q$ in $\mathcal{M}(T)$, then for some $t \in ET(a_i; p_i?)$, there exists a homomorphism $g: G(t) \longrightarrow \mathcal{M}(T)$ which maps the origin of G(t) on x. Under the inductive assumption, the superposition hg is a homomorphism from G(t) to $\mathcal{M}$, and by Lemma 3.1, $h(x) \models q$ in $\mathcal{M}$. Since h is a homomorphism, the fact $h(x) \models q$ in $\mathcal{M}$ implies that $q \in F(x)$ in G(T). The converse implication in Lemma 4.1 is immediate.□

Looking forward to Section 8, we are interested in recognizing whether a given validation tree is consistent. In fact, all easy for tree automata consistency requirements are already contained in the notion of a validation tree. The remaining difficulty is isolated by the following lemma.

Lemma 4. 2 A validation tree T of a formula p is consistent if every diamond subformula $\langle a \rangle q$ of p satisfies the following condition:

(4.2) if every formula contained in a or in q satisfies (4.1), then for every node x of G(T)

$$\langle a \rangle q \notin F(x) \text{ in } G(T) \quad \text{implies} \quad x \not\models \langle a \rangle q \text{ in } \mathcal{M}(T).$$

Proof: Directly from definitions of validation tree and consistency.

5. EXECUTIONS OF PROGRAMS: DYNAMIC DESCRIPTION

Lemma 4. 2 points out a condition in the notion of consistency which must be further transformed to be more suitable for tree automata. In the case of regular programs without intersection this can be done quite easily. For a formula $\langle a \rangle q$ we construct a finite automaton $\mathfrak{A}$ which recognizes the set of execution sequences of the program a;q?. The condition (4.2) is satisfied iff every node x of a validation graph G(T) can be labelled with a set R(x) of "reachable" states of $\mathfrak{A}$ in such a way that the following three conditions hold: (d1): if $\langle a \rangle q \notin F(x)$, then all initial states of $\mathfrak{A}$ belong to R(x), (d2): if nodes x, y are adjanced and there is an $\mathfrak{A}$-transition from a state s in x to a state s' in y, then $s \in R(x)$ implies $s' \in R(y)$, (d3): for any node x, R(x) contains no final states of $\mathfrak{A}$. All (d1)-(d3) can be easily checked by a tree automaton with the input T.

Here, in the presence of intersections, we follow the same idea. However, single finite automaton must be replaced by a system of cooperating automata, To execute a program $a \cap b$ we may start one automaton for a and one for b, allow them to work independently, and then check if they meet in final states in one node of a structure. Since a and b may contain further intersections, it is convenient to implement this idea in the following "token game" style.

To begin an execution of a program a at a node x we put at x a marker beg a. If $a = a_1 ; a_2$, then beg a is replaced in x by the marker beg a_1. If $a_1 = A$ (atomic program), then beg a_1 in x is replaced by a marker end a_1 in some node y with $x \prec A \succ y$, and further, end a_1 is replaced in y by beg a_2. If $a_2 = a_3 \cap a_4$, then beg a_2 is replaced by two markers beg a_3, beg a_4, both in y. If later end a_3 meets with end a_4 in some node z, then they both are replaced by a single marker end a_2 in z, and this in turn is replaced by end a. Such a "game" is nothing but a transformation from regular programs to corresponding finite automata, however, intersections make that automata split and merge. Observe, that if a=A;A, then we must differentiate between markers of the first and the second occurrence of A. This is why in formal definitions we refer to nodes of the syntactical tree of a (i. e. to particular ocurrences of subprograms) rather than to subprograms as such.

This section presents the semantics of programs in terms of markers and transitions. Next two sections will show how to compute sets of reachable states.

Any program, treated as an expression, has a syntactical tree in which leaves are labelled with atomic programs or tests and internal nodes are labelled with program forming functors ;, $\cup$, $\cap$, or $*$. Let a be a program. We define the set Mark(a) of markers of a as the set of all expressions of the form begα, endα, where α is any node in the syntactical tree of a. Some particular sets of markers will be called control states of a. The set of control states Cst(a) and the set of instructions Instr(a) of the program a are defined by the following structural induction on nodes of the syntactical tree.

If α is a leaf labelled with A or q?, respectively, then $Cst(\alpha) = \{\{beg\alpha\}, \{end\alpha\}\}$ and $Instr(\alpha)$ contains the single instruction $\{beg\alpha\} \vdash(A)\rightarrow \{end\alpha\}$, or $\{beg\alpha\} \vdash(q?)\rightarrow \{end\alpha\}$, respectively.

If $\alpha = \beta;\gamma$ or $\alpha = \beta \cup \gamma$, then $Cst(\alpha) = \{\{beg\alpha\}, \{end\alpha\}\} \cup Cst(\beta) \cup Cst(\gamma)$, $Instr(\alpha)$ contains $Instr(\beta) \cup Instr(\gamma)$ and the following instructions. In the case $\alpha = \beta;\gamma$: $\{beg\alpha\} \longmapsto \{beg\beta\}$ $\{end\beta\} \longmapsto \{beg\gamma\}$, $\{end\gamma\} \longmapsto \{end\alpha\}$. In the case $\alpha = \beta \cup \gamma$: $\{beg\alpha\} \longmapsto \{beg\beta\}$, $\{beg\alpha\} \longmapsto \{beg\gamma\}$, $\{end\beta\} \longmapsto \{end\alpha\}$, $\{end\gamma\} \longmapsto \{end\alpha\}$.

If $\alpha = \beta \cap \gamma$, then $Cst(\alpha) = \{\{beg\alpha\}, \{end\alpha\}\} \cup \{S \cup S' : S \in Cst(\beta), S' \in Cst(\gamma)\}$, $Instr(\alpha)$ contains $Instr(\beta) \cup Instr(\gamma)$

and the following instructions: $\{\mathrm{beg}\,\alpha\} \longmapsto \{\mathrm{beg}\,\beta,\ \mathrm{beg}\,\gamma\}$, $\{\mathrm{end}\,\beta,\ \mathrm{end}\,\gamma\} \longmapsto \{\mathrm{end}\,\alpha\}$.

If $\alpha = \beta^*$, then $Cst(\alpha) = \{\{\mathrm{beg}\,\alpha\}, \{\mathrm{end}\,\alpha\}\} \cup Cst(\beta)$, Instr($\alpha$) contains Instr($\beta$) and the following instructions: $\{\mathrm{beg}\,\alpha\} \longmapsto \{\mathrm{beg}\,\beta\}$, $\{\mathrm{end}\,\beta\} \longmapsto \{\mathrm{beg}\,\beta\}$, $\{\mathrm{end}\,\beta\} \longmapsto \{\mathrm{end}\,\alpha\}$.

Let $\mathcal{M} = (X, \models, \prec\ \succ)$ be a structure. A state of a program a in $\mathcal{M}$ is any mapping $Q: S \longrightarrow X$, where $S \in Cst(a)$. We shall treat Q as a subset of $Mark(a) \times X$ and the fact that a marker s is put at x, i. e. $Q(s) = x$, will be written as $(s, x) \in Q$. Every state of the form $Q = S \times \{x\}$ will be called concentrated at x and written as $Q = (S, x)$.

The transition relation $\longmapsto$ between states of a program a in $\mathcal{M}$ is defined as follows. $Q \longmapsto_1 Q'$ in $\mathcal{M}$ iff there exist $S, S' \subset Mark(a)$ and $x, y \in X$, such that $(S, x) \subset Q$, $Q' = (Q \setminus (S, x)) \cup (S', y)$ and one of the following conditions holds. Either (e1): $x = y$ and $S \longmapsto S' \in Instr(a)$, or (e2): $x = y$, $x \models q$ in $\mathcal{M}$, and $S \vdash\!(q?)\!\rightarrow S'$ is in Instr(a), or (e3): $x \prec A \succ y$ and $S \vdash\!(A)\!\rightarrow S' \in Instr(a)$. $Q \longmapsto Q'$ means $Q \longmapsto_k Q'$ for some $k \geqslant 0$, where $Q \longmapsto_0 Q'$ stands for $Q = Q'$, and $Q \longmapsto_k Q'$, with $k > 1$, means $Q \longmapsto_{k-1} Q'' \longmapsto_1 Q'$, for some Q''.

<u>Lemma 5. 1</u> For any nodes x, y of a structure $\mathcal{M}$ and any program a, $x \prec a \succ y$ in $\mathcal{M}$ iff $(\{\mathrm{beg}\ a\}, x) \longmapsto (\{\mathrm{end}\ a\}, y)$. □

We say that a transition $Q \longmapsto_k Q'$, $k > 1$, can be splitted if there exist states $Q_1 \subset Q$, $Q_1' \subset Q'$ such that $Q_1 \longmapsto_m Q_1'$ and $(Q \setminus Q_1) \longmapsto_n (Q' \setminus Q_1')$, where $m+n = k$, and either $m, n \geqslant 1$, or $n = 0$ and $Q \setminus Q_1 \neq \emptyset$.

A transition $Q \longmapsto_k Q'$ can be concentrated at a node z, if there exists a concentrated state (S, z) such that $Q \longmapsto_m (S, z)$, and $(S, z) \longmapsto_n Q'$, where $m, n \geqslant 1$, $m + n = k$.

We end this section with a decomposition lemma which reflects the fact that an execution graph of a program is well nested.

<u>Lemma 5. 2</u> Every transition $Q \longmapsto_k Q'$ with $k > 1$ can be either splitted or concentrated.

Proof: Let $Q_0 \vdash_1 Q_1 \vdash_1 \dots \vdash_1 Q_k$ be a transition, where $Q_i: S_i \longrightarrow X$, $S_i \in Cst(a)$, for every $0 \leq i \leq k$. The whole proof is by a careful analysis of the set of instructions. If $|S_0|=1$, then Q_1 also must be concentrated. If $|S_0| > 1$, then $S_0 \in Cst(b \frown c)$ for some subprograms b, c of a. Now, we ask if end($b \frown c$) appears in any S_i or not. If yes, then the only possibility is that S_i is a singleton $\{end(b \frown c)\}$ and therefore Q_i is concentrated. If not, then every S_i must be a union of disjoint sets S_i' and S_i'', where $S_i' \in Cst(b)$, $S_i'' \in Cst(c)$. This induces the split. □

6. CUTPOINTS AND CONCENTRATIONS IN SPECIAL GRAPHS

This section presents the crucial properties of special graphs that enable tree automata to compute sets of reachable states of programs.

For nodes x, y, z in a directed graph, we say that z is a cutpoint for the pair (x, y) if $x \neq z \neq y$ and every path from x to y must contain z.

Recall, that T_u is the full subtree of T starting with u as the root. Thus, $G(T_u)$ is a subgraph of G(T). A node x of G(T) is inside $G(T_u)$ if it belongs to $G(T_u)$ but is different from the origin u1 and the sink u2 of $G(T_u)$. A node is outside $G(T_u)$ if it does not belong to $G(T_u)$.

Lemma 6. 1 If $x \neq y$ and z is a cutpoint for (x, y), then any transition from a state concentrated at x to a state concentrated at y can be concentrated at z.

Hint to the proof: If $(S, x) \vdash (S', y)$ and x y, then every marker from S makes a trip from x to y through some trajectory that passes z. The whole task is to reorganize the order in which instructions are performed. Here we use the fact that the system of trajectories of markers is a fragment of a well nested graph. In such a fragment, sources of all trajectories are labelled with x, targets with y, and we can find a level in which every point is labelled with z and neither point of the level precedes the other. This means that if a marker from S (strictly speaking, a successor of such a marker) reaches this choosen level, it can wait until remaining markers from S reach this level. This does not affect the final result of the transition nor the number of instructions performed. □

Lemma 6. 2 Let $u \in T$ and let x, y be nodes of $G(T)$ such that x is inside $G(T_u)$ and y is outside $G(T_u)$. Then, either: one of the nodes u1 or u2 is a cutpoint for both (x, y) and (y, x), or: u1 is a cutpoint for (y, x) and u2 is a cutpoint for (x, y).

Proof: Induction on the complexity of T_u. □

Nodes u, v of T are neighbours if u is a son or the father of v. Recall, that nodes x, y of $G(T)$ are coupled by u if they both fall in u, i. e. $x, y \in \{u1, u2\}$. For nodes x, y, z of $G(T)$ we say that z is in the neighbourhood of (x, y) if there exist neighbours u, v in T such that x, y are coupled by u and z falls in u or v.

Lemma 6. 3 If nodes x, y of $G(T)$ are not coupled in T, then the pair (x, y) has a cutpoint in $G(T)$.

Proof: For any nodes x, y of $G(T)$ we can find the shortest undirected path in T, $u_0 u_1 \ldots u_k$, such that x falls in u_0 and y falls in u_k. If x, y are not coupled, then $k>0$. If $k>1$, then we can find $v=u_i$ such that x is inside, and y is outside $G(T_v)$, or vice versa, and then apply Lemma 6.2. If k 1, then the proof is by an immediate analysis of cases. □

Lemma 6. 4 Suppose that nodes x, y of $G(T)$ are coupled in T. If a transition $(S, x) \vdash (S', y)$ can be concentrated, then it can be concentrated in the neighbourhood of (x, y).

Proof: Suppose that $(S, x) \vdash (S'', z) \vdash (S', y)$, $x \neq z \neq y$.
Case 1. $(x=y)$: Find the shortest undirected path $u_0 u_1 \ldots u_k$ in T such that x falls in u_0 and z falls in u_k. If $k \leq 1$, the case is proved. If $k>1$, then x and z are on different sides of u_1 and, by Lemma 6.2, some cutpoint z' for (x, z) falls in u_1. By Lemma 6.1, the transition can be concentrated at z' in the neighbourhood of x.

Case 2. $(x \neq y)$: Find the shortest undirected path $u_0 u_1 \ldots u_k$ in T such that x, y are coupled by u_0 and z falls in u_k, $k>1$. One of the nodes x or y does not fall in u_1. Thus, similarly as in the Case 1, by Lemma 6.2, either the pair (x, z) or the pair (z, y)

has a cutpoint that falls in u_1. In both cases, by Lemma 6.1, the transition can be concentrated in the neighbourhood of (x, y). □

7. SOLUTION TO THE MAIN DIFFICULTY

Let us return to Lemma 4.2 and recall what we mean by the main difficulty. Suppose T is a validation tree for some formula p. Let $\langle a\rangle q$ be a subformula of p, such that every formula q' contained in a or in q satisfies for every node x of G(T) the condition: $q' \in F(x)$ in G(T) iff $x \models q'$ in $\mathcal{M}(T)$. Our task is to transform the following implication to the form suitable for tree automata.

(7.1) If $\langle a\rangle q \notin F(x)$ in G(T), then $x \not\models \langle a\rangle q$ in $\mathcal{M}(T)$, for every node x of G(T).

Such a form will be presented in the last lemma of this section.

By a reachability plan for a program a in G(T) we mean a labelling of T, which to every $u \in T$ assigns two sets R1(u), R2(u) of control states of the program a in such a way that the following condition holds:

(7.2) if $S \in Ri(u)$ and $(S, ui) \vdash (S', wj)$, then $S' \in Rj(w)$, for all $u, w \in T$, $i, j = 1, 2$, $S, S' \in Cst(a)$.

<u>Lemma 7. 1</u> The condition (7.1) is satisfied iff there exists a reachability plan R1, R2 for the program a;q? in G(T) such that for every $u \in T$, $i = 1, 2$, the following conditions hold:

(7.3) $\langle a\rangle q \notin Fi(u)$ implies $\{beg(a;q?)\} \in Ri(u)$,

(7.4) $\{end(a;q?)\} \notin Ri(u)$.

<u>Proof:</u> Immediately from definitions and Lemma 5.1. □

Suppose that for every $u \in T$ there are defined four binary relations $Mij(u) \subset \underline{P}(Mark(a))^2$, i, j = 1, 2, on sets of markers of a program a. We say that the labelling Mij, i, j = 1, 2, of T is a plan of transitions of the program a between coupled nodes of G(T), if for every $u \in T$, $i, j = 1, 2$, $S, S' \in Cst(a)$,

(7.5) $(S, ui) \vdash (S', uj)$ implies $(S, S') \in Mij(u)$.

Lemma 7. 2 Suppose that for every $u \in T$, $R1(u), R2(u) \subset Cst(a)$. The labelling R1, R2 of T is a reachability plan for a in G(T) iff there exists a plan of transitions of a between coupled nodes of G(T), Mij, i, j=1, 2, such that for every $u \in T$, i, j=1, 2, $S, S' \in Cst(a)$, the following two conditions hold:

(7.6) $S \in Ri(u)$ and $(S, S') \in Mij(u)$ imply $S' \in Rj(u)$,

(7.7) if places (u, i) and (w, j) are elementarily equivalent in T, then $Ri(u) = Rj(w)$.

Proof: What the Lemma 7.2 actually says is that, if ui=wj implies $Ri(u) = Rj(w)$ (this is guaranteed by (7.7)), then the condition (7.2) can be restricted to pairs of coupled nodes ui, uj , instead of arbitrary nodes ui, wj. Suppose that (7.2) holds for pairs of coupled nodes, and let $S \in Ri(u)$, $(S, ui) \vdash (S', wj)$ for some $S \in Cst(a)$, ui, $wj \in G(T)$. If the pair (ui, wj) has no cutpoints, then by Lemma 6.3, ui and wj are coupled, and $S' \in Rj(u)$. If there are cutpoints for (ui, wj), then by Lemma 6.1, the transition can be decomposed $(S, ui) = (S_o, u_o) \vdash (S_1, x_1) \vdash \dots \vdash (S_k, x_k) = (S', wj)$, where every pair (x_{i-1}, x_i) has no cutpoints and therefore is coupled. Thus, by superposition and (7.2) for coupled nodes, $S' \in Rj(u)$. □

Lemma 7. 3 Let $Mij(u) \subset \underline{P}(Mark(a))^2$, for every $u \in T$, i, j=1, 2. The labelling Mij, i, j=1, 2, of T is a plan of transitions of the program a between coupled nodes in G(T) iff the following conditions are satisfied (universal quantifiers omitted):

(7.8) if $S \longmapsto S' \in Instr(a)$, then $(S, S') \in Mii(u)$,

(7.9) if $S \vdash\!(A)\!\to S' \in Instr(a)$ and $sign(u) = A$, then $(S, S') \in M12(u)$,

(7.10) if $S \vdash\!(p?)\!\to S' \in Instr(a)$ and $p \in Fi(u)$, then $(S, S') \in Mii(u)$,

(7.11) any relation Mii(u) is reflexive and transitive,

(7.12) if $S_1 \cap S_2 = \emptyset$ and $\{(S_1, S_1'), (S_2, S_2')\} \subset Mij(u)$, then $(S_1 \cup S_2, S_1' \cup S_2') \in Mij(u)$,

(7.13) the elementary equivalence of places $(u, i) \sim (u', i')$ and $(u, j) \sim (u', j')$ implies $Mij(u) = Mi'j'(u')$,
$(u, i) \sim (u, j)$ implies $Mii(u) = Mij(u)$,
$(u, i) \sim (w, j)$ implies $Mii(u) = Mjj(w)$,

(7.14) $Mij(u) \cdot Mji(u) \subset Mii(u)$, $Mii(u) \cdot Mij(u) \cdot Mjj(u) \subset Mij(u)$, (superposition of relations),

(7.15) if sign(u) = ; , v = left son(u), w = right son(u), then
$M12(v) \cdot M12(w) \subset M12(u)$, $M21(w) \cdot M21(v) \subset M21(u)$,
$M12(u) \cdot M21(w) \subset M12(v)$, $M12(w) \cdot M21(u) \subset M21(v)$,
$M21(v) \cdot M12(u) \subset M12(w)$, $M21(u) \cdot M12(v) \subset M21(w)$.

Proof: The only interesting part of the lemma is that conditions (7.8)-(7.15) imply (7.5). The proof is by induction on the size of transitions, where the size of $(S, x) \vdash_k (S', y)$ is $k + |S|$. Observe, that (7.5) says the following: if $(S, x) \vdash (S', y)$ and $x = ui$, $y = uj$, for some $u \in T$, $i, j = 1, 2$, then $(S, S') \in Mij(u)$. That is, we must prove the consequent for every representation of x, y in T. This will be solved in advance in a series of four facts which show that, if (7.5) is proved for one representation, it holds for all remaining. Let us assume that a labelling Mij satisfies (7.8)-(7.15). Recall, that $ui = wj$ in G(T) iff there exists a sequence of places $(u_o, i_o) \ldots (u_k, i_k)$ such that $u = u_o$, $w = u_k$, $i = i_o$, $j = i_k$, and for all $0 < n \leqslant k$, $(u_{n-1}, i_{n-1}) \sim (u_n, i_n)$. The shortest such sequence will be called the evidence of the equality $ui = wj$.

Fact_1: If $ui = uj$, then Mij(u) is reflexive. The proof of this fact is by induction on the of the evidence of $ui = uj$. Basis is provided by (7.13) and (7.11). For induction, observe that an evidence of $ui = uj$ does not enter i-th sons, since it would produce useless loops. Moreover, such an evidence lays fully inside or fully outside $G(T_u)$. Thus, every evidence of $ui = uj$ has either the form $(u, i) \sim (v, i_1) \ldots (v, i_2) \sim (w, j_1) \ldots (w, j_2) \sim (u, j)$, or a simpler one, without w, where one of the nodes u, v, w is the father of remaining two, and the equalities $vi_1 = vi_2$, $wj_1 = wj_2$ have shorter evidences than $ui = uj$. Thus, by inductive assumption, Mkl(v) and Mkl(w) are reflexive for any $k, l = 1, 2$. Now it is easy to combine (7.11)-(7.15) to prove reflexiveness of Mij(u).

Fact_2: If $ui = wj$, then $Mii(u) = Mjj(w)$. This is an obvious induction on the length of the evidence of equality. The basis is assumed in (7.13).

Fact_3: If $ui = uj$, then $Mii(u) = Mij(u)$. The proof is by Fact 1 and (7.14).

Fact 4: If $ui = u'i'$ and $uj = u'j'$, then $Mij(u) = Mi'j'(u')$. Consider the shortest undirected path $u_o u_1 \dots u_k$ in T from $u = u_o$ to $u' = u_k$. Observe, that evidences for both $ui = u'i'$ and $uj = u'j'$ must pass through every node u_n, $1 < n < k$, i. e. for every n there exist $u_n i_n = ui$, $u_n j_n = uj$. Thus, it suffices to prove Fact 4 for u and u' being neighbours in T. If $u = u'$ or $ui = uj$, the case reduces to Fact 3. The analysis of remaining cases shows, that at least one of the equalities has an evidence of length 1, and the remaining, in the worst case, has an evidence of the form, say, $(u, i) \sim (w, i_1) \dots$ $\dots (w, i_2) \sim (u', i')$, where w is a son of u or u'. Since $wi_1 =$ $= wi_2$, by Fact 3, $M11(w) = Mkl(w)$, for every k, $l = 1, 2$. In every particular case of this type it is easy to use (7.15), (7.11) to prove that $Mij(u) = Mi'j'(u)$.

Now, we return to the inductive proof of the Lemma 7.3. Basis: It is not hard to see, that (7.8)-(7.12) and Fact 3 suffice to prove that (7.5) holds for transitions $(S, ui) \vdash_1 (S', uj)$ with any size of S.

Induction: Consider a transition $(S, ui) \vdash_k (S', uj)$ and assume that (7.5) holds for all transitions of smaller size. It is to be proved, that $(S, S') \in Mij(u)$. If the transition can be splitted, then we apply (7.12). Otherwise, by Lemma 5.2, there exists a concentration $(S, ui) \vdash_m (S'', z) \vdash_n (S', uj)$, where $m, n < k$, z is some node of G(T). By Lemma 6.4, we may assume that z is in the neighbourhood of (ui, uj), and by Fact 4, we may further assume that z falls in u or in a neighbour v of u. To prove that (S, S') is in $Mij(u)$ we must analyse all possible situations in the neighbourhood of u. For example, suppose that sign(u) = ; , v = left son(u), w = right son(u), $i = 1$, $j = 2$, $z = w1$. Since $u1 = v1$, $w1 = v2$, $u2 = w2$, then by the inductive assumption $(S, S'') \in M12(v)$, $(S'', S') \in M12(w)$. Thus, by (7.15), $(S, S') \in M12(u)$. In a similar way we deal with other cases. □

Lemma 7. 4 If T is a validation tree of a formula p and $\langle a \rangle q$ is a subformula of p, then the condition (7.1) is satisfied iff for every $u \in T$ there exist $Ri(u) \subset Cst(a;q?)$, $Mij(u) \subset \underline{P}(Mark(a;q?))^2$, $i, j = 1, 2$, such that the conditions (7.3)-(7.4) and (7.6)-(7.15) are satisfied. (The conditions of Lemma 7.3 are taken for the program a;q?.)

Proof: Superposition of Lemmas 7.1 - 7.3 . □

8. THE FINAL RESULT

The essential part of our proof has been completed in Section 7. All what remains is to show that the set of consistent validation trees of a given formula can be recognized by a special tree automaton ([R70]). This is rather routine, that is involves only known techniques, and we shall not go into details. However, all definitions will be recalled and some intermediate claims stated.

By a (full, infinite) n-ary tree we mean the set of words $\mathcal{T}_n = \{1, \ldots, n\}^*$, where the empty word λ is the root of $\mathcal{T}_n$, and u1, u2, ..., un are the sons of u. A n-ary Ω-tree is a mapping $f: \mathcal{T}_n \longrightarrow \Omega$,(i. e. nodes are labelled with elements of Ω).

A special tree automaton over n-ary Ω-trees ([R70]) is a 4-tuple $\mathfrak{A} = (S, M, S_0, F)$, where S is a finite set of states, S_0, F are subsets of S consisting of initial and final states, respectively, and $M \subset S \times \Omega \times S^n$ is a tree transition relation. A tree f is accepted by $\mathfrak{A}$, if there exists a function $r: \mathcal{T}_n \longrightarrow S$, such that the following conditions hold: (f1): $r(\lambda) \in S_0$, (f2): for every $u \in \mathcal{T}_n$, $(r(u), f(u), r(u1), \ldots, r(un)) \in M$, (f3): for every infinite path $u_0 u_1 \ldots$ in $\mathcal{T}_n$, where u_i is a son of u_{i-1}, $r(u_i) \in F$ for infinitely many i.

Every n-ary type-2 tree over Δ, Σ can be extended to a full infinite n-ary tree by adding nodes with the label # . Thus, every such tree is a n-ary $\Omega_{\Delta\Sigma}$-tree with $\Omega_{\Delta\Sigma} = (\{;, //, \text{"equal"}\} \cup \Delta) \times \Sigma \times \Sigma \cup \{\#\}$ and u1, u2, u3, u4, ... corresponding to left-, right-, 1-th left-, 1-th right-, ... sons of u.

Lemma 8. 1 For every formula p there can be effectively constructed a special tree automaton $\mathfrak{A}$ which accepts exactly consistent validation trees of p. The number of states of $\mathfrak{A}$ is O(exp exp c|p|), where c is a constant, and its construction can be done in time polynomial of the number of states.

Hint to the proof: Let p be a formula with n-ary validation trees over Δ, Σ. For a n-ary $\Omega_{\Delta\Sigma}$-tree f, let T_f be the maximal subtree of f which contains the root λ and only these nodes which are not labelled with # . The automaton $\mathfrak{A}$ can be constructed as the conjunction of the following three automata. First, we define $\mathfrak{A}_1$ that recognizes if T_f is a type-2 tree over Δ, Σ . This requires only a constant number of states and the condition (f3) is used to check if every restricted (to left and right sons) subtree of T_f is finite. Then, we construct $\mathfrak{A}_2$ which accepts f iff the following implication holds: if T_f is a type-2 tree, then T_f is a validation tree of p. This can be done using $O(|p|)$ states with no reference to (f3). Here, a useful intermediate step is a construction of an automaton on finite trees which recognizes execution trees of a program. Finally, we construct $\mathfrak{A}_3$ which accepts f iff the fact that T_f is a validation tree implies that T_f is consistent. The construction of $\mathfrak{A}_3$ is based on Lemma 7.4. States of $\mathfrak{A}_3$ guess values of Ri(u), Mij(u), for every diamond subformula of p, and the transition relation of $\mathfrak{A}_3$ checks local conditions. This also does not use (f3). The number of states of $\mathfrak{A}_3$ is $O(\exp \exp c|p|)$ for some c. Generally, we need only a small part of the power of special automata. □

Theorem 8. 2 The satisfiability problem for PDL with intersection is decidable in time double exponential in the length of the formula tested.

Proof: By Lemmas 4.1, 8.1 and the fact, that the emptiness problem for special tree automata is decidable in time polynomial of the size of the set of states and the input alphabet. □

R e f e r e n c e s:

[D] R. Danecki, Propositional Dynamic Logic with strong loop predicate, Proc. MFCS'84, LNCS 176, 573-581 (1984) Springer-Verlag

[FL] M. J. Fisher, R. E. Ladner, Propositional Dynamic Logic of regular programs, JCSS 18:2, (1979), 194-211

[H] D. Harel, Recurring dominoes: Making the highly undecidable highly understandable, Proc. FCT'83, LNCS 158, 177-194, (1983) Springer-Verlag

[R69] M. O. Rabin, Decidability of second-order theories and automata on infinite trees, Trans. AMS 141 (1969), 1-35

[R70] M. O. Rabin, Weakly definable relations and special automata, in: Math. Logic and Found. of Set Theory (Y. Bar-Hillel ed.) North-Holland (1970), 1-23

[S] R. S. Streett, Propositional Dynamic Logic of looping and converse is elementarily decidable, Inform. & Control 54, 121-141 (1982)