MIXED COMPUTATIONS AND ADAPTATION OF PROGRAMS

V. É. Itkin

The concept of mixed computation is briefly described. An equivalence criterion of deterministic multitape automata is established in terms of mixed computations. Some considerations concerning adaptive properties of programs are presented.

MIXED COMPUTATIONS

The concept of mixed computation [1] includes 1) the mapping \( M: P \times D \rightarrow P \times D \), where \( P \) is the set of programs, \( D \) the set of data; the program realizing the mapping \( M \) is called a mixed computer and the computation process \( M(p, d) \) is called a mixed computation, because it involves transformation of the medium consisting of programs and data; the program is treated as a text in some language, possibly the same language as that of the mixed computer and the data; 2) a correctness criterion of mixed computation, represented as an infological invariant \( I: M(p, d) = (p', d') \) implies that \( I(p, d) = I(p', d') \); a suitable invariant is provided by the semantic function \( \text{val} \) or by the infological structure of the pair \( (p, d) \); 3) tools of manipulating incompletely defined information: the program and data structures contain undefined symbols; if \( M(p, d) = (p', d') \), then \( p' \) is called the residual program, because \( p' \) is invoked to complete the computation, possibly by using additional information, and \( d' \) is called an intermediate memory state; \( d' \) may contain a partial result which is included in the complete result of the ordinary computation \( \text{val} \) of the program \( p \) on \( d \) executed with any sufficient supplementary input information; 4) an efficiency criterion of mixed computation.

Mixed computations are designed for adaptation (concretization, specialization, projection, reduction, tuning, coordination) of programs to the available input information, dynamic decomposition of programs, and structuring of programs for purposes of program synthesis, optimization, compiling, and parallelization. Particular cases of mixed computations include decomposition of programs, which corresponds to symbolic mixed computation; projection, which produces the residual program; and approximate computation, which produces a partial result.

The efficiency of mixed computation can be improved by the polyvariant [2-4] strategy, which involves initiating mixed computation processors on alternative data under conditions of input uncertainty. For finite sets of data, mixed computations reduce to polyvariant ordinary computations. In the general case, there are difficulties with data factorization as well as with interaction and stopping of mixed computation processors.

The notion of efficiency of mixed computations essentially coincides with the notion of structuredness of programs [3]. Indeed, structuredness of a program (a text, an article) is its amenability to efficient partial processing. Common efficiency criteria of mixed computation (which in general involve numerous tradeoffs), such as the level of development of the initial program, breadth of the partial result, and compactness of the residual program, are closely linked with algebraic, compositional, topological, informational, and modificational criteria of program structuredness [5].


0011-4235/90/2601-0032$12.50 ©1990 Plenum Publishing Corporation
A program p equipped with a modification labeling g is considered as a new program \((p, g)\) whose semantics may be substantially different from the semantics of p. Here \(T: (p, g) \rightarrow p'\) is the label removal (delabeling) operator that converts \((p, g)\) into an unlabeled program \(p'\) semantically equivalent to \((p, g)\).

In this section, the problem of deciding equivalence of deterministic automata [6] is reduced to the problem of deciding equivalence of \(p\) and \(T(p, g)\) for a certain class of programs and labelings.

Let \(X = \Sigma_1 \cup \Sigma_2 \cup \ldots\) be the input alphabet, \(\Sigma_i\) (the alphabet of tape i) a finite nonempty set; the alphabets of different tapes are nonintersecting. An automaton is a 4-tuple \(p = (Q, q_0, Q_f, \delta)\), where \(Q\) is the (finite) state set, \(q_0 \in Q\) is the initial state, \(Q_f - Q\) is the set of final states, \(\delta\) is the partial transition function from \(Q \times \Sigma\) to \(Q\), and for each \(q \in Q\), there exists \(i \in \{1, 2, \ldots\}\) such that \(\lambda(q, \sigma)\) is defined if and only if \(\sigma \in \Sigma_i\). We denote by \(\Sigma^*\) the set of all words of finite length in the alphabet \(\Sigma\), including the empty word \(\lambda\); \(pr_i(g)\) is the component of word \(g \in \Sigma^*\) on the alphabet of tape i: \(pr_i(g)\) equals \(opr_i(g)\) for \(\sigma \in \Sigma_i\) and \(pr_i(g)\) for \(\sigma \notin \Sigma_i\). The words \(g, h \in \Sigma^*\) are equivalent \((-\) if \(pr_i(g) = pr_i(h)\) for each i. A path of the automaton \((Q, q_0, Q_f, \lambda)\) is the state \(q \in Q\) (the trivial path from q to q) or the word \(q_1\sigma_1\ldots q_n\sigma_{n+1}\), where \(n \geq 1, \lambda(q_1, \sigma_1) = q_2, \ldots, \lambda(q_n, \sigma_n) = q_{n+1}\). The history of the path is \(\lambda\) in the first case and \(\sigma_1\ldots\sigma_n\) in the second case. An automaton path is called perfect if it is a path from the initial node to some final node of the given automaton. We denote by \([p]\) the set of histories of the perfect paths of the automaton p. The relation \(g \in_p [p]\) denotes that for some word \(h\) we have \(h \sim g\) and \(h \in U\), where \(U \supset \Sigma^*\).

Arbitrary sets \(U, U' \subset \Sigma^*\) are equivalent \((-\) if for each \(g \in U\) we have \(g \in_e U'\) and for each \(g' \in U'\) we have \(g' \sim_e U\). The automata p and p' are equivalent \((-\) if \([p] \sim [p']\).

Let us define the difference \(p - g\) of the automaton p and the word g (a partial operation), which may be regarded as a delabeling \(T(p, g)\).

For any \(g, h \in \Sigma^*\), the relation \(g \leq h\) indicates that \(gg' = h\) for some \(g' \in \Sigma^*\); \(g \leq_e h\) indicates that \(pr_i(g) \leq pr_i(h)\) for each i; \(g \gg h\) indicates that \(pr_i(g) < pr_i(h)\) for each i. For arbitrary \(g, h \in \Sigma^*\), the difference \(h - g\) is defined if and only if \(g \geq h\) and if it is defined, then it equals \(g'\) such that \(gg' = h\) and \(g'\) is obtained from h by annihilating the occurrences that correspond to g while preserving the order of the occurrences that remain in g'. We say that the word \(g \in \Sigma^*\) is a prefix of the automaton p (\(g \prec_e p\)) if there exists \(h \in [p]\) such that \(g \leq e h\) and for any \(h' \in [p]\), \(g \gg h'\) implies \(g \leq_e h'\). Denote by \([p] - g\) the set \(\{h - g \mid h \in [p], g \leq_e h\}\). We assume that for an arbitrary word \(g \in \Sigma^*\) and an arbitrary automaton p, the difference \(p - g\) is defined if and only if \(g \leq_e p\), and if defined, it is equal to an automaton \(p'\) such that \([p'] = [p] - g\). An idea of constructing such an automaton by unfolding, reduction, and return of arcs on sections free from reduction was advanced in [3]. Similar transformations of program transition graphs are given in [2].

**Lemma 1.** The automata \((p - g) - h\) and \(p - (gh)\) are defined (subtraction is defined) only simultaneously, and if defined, then they are equivalent.

**Lemma 2.** \(gh \in_e [p]\) if and only if \(p - g\) is defined and \(h \in [p - g]\).

The pair of words \((h_1, h_2)\) is called a key pair for the automaton p if there exist paths \(H_1, H_2, H_3\) and a state \(q\) of the automaton p such that \(H_1\) is a simple path (without repeating occurrences of states) from the initial node to q; \(H_2\) is a path from q to q without repeating occurrences of states other than q, and without intermediate occurrences of q itself in \(H_2\); \(h_1\) is the history of \(H_1\), \(h_2\) is the history of \(H_2\); \(H_3\) is a path from q to any final node of the automaton p.

**Theorem 1.** Arbitrary automata p and p' are equivalent if and only if the following conditions hold: 1) for every simple perfect path of each of these automata, there exists an equivalent (with an equivalent history) perfect (not necessarily simple) path of the other automaton; 2) for each key pair \((h_1, h_2)\) of the automaton p we have \(h_1h_2 \leq_e p', p' - h_1 \sim (p' - h_1) - h_2\), and for each key pair \((h'_1, h'_2)\) of the automaton p' we have \(h'_1h'_2 \leq_e p, p - h'_1 \sim (p - h'_1) - h'_2\).

**Proof. Necessity.** Let \(p \sim p'\). Property 1 is obvious. Let \((h_1, h_2)\) be a key pair for p. We will show that \(h_1h_2 \leq_e p'\). Indeed, first we have \(h_1h_2 \leq_e p\), because \(h_1h_2h_3 \in [p]\) for some \(h_3\); second, the definition of \(\leq_e\) uses only \([p]\) as information about p, so that \(p \sim p'\) implies \(h_1h_2 \leq_e p'\). It now remains to show that \(p' - h_1 \sim (p' - h_1) - h_2\); \([p' - h_1] \sim \{p' - h_1\} - h_2\).

**Insufficiency.** Let conditions 1 and 2 hold. We will show that \(p \sim p'\). From condition 1 it follows that if \([p] = \emptyset\), then \([p'] = \emptyset\). By induction on the length of the perfect path H of the automaton p we prove that there exists an equivalent perfect path of the automaton p'. If H is simple, then the sought result follows from condition 1. Now assume that H is not simple and