Programming and Computer Software, Vol. 26, No. 2, 2000, pp. 61–73.

Application of Finite Automatons for Program Testing

I. B. Burdonov, A. S. Kossatchev, and V. V. Kulyamin

Institute of System Programming, Russian Academy of Sciences, Bol'shaya Kommunisticheskaya 25, Moscow, 109004 Russia
e-mail: igor@ispras.ru, kos@ispras.ru, kuliamin@ispras.ru

Received July 12, 1999

Abstract—The application of the finite automaton theory to the problem of program testing is discussed. The problem is reduced to testing a finite automaton. Testing of automatons using their state graphs, factor graphs, testing using factor graphs, and methods for factor graphs construction are discussed.

1. INTRODUCTION

Automatons provide a widely used model of hardware and software objects. The main difference between an automaton and a purely functional dependence is that the values of output parameters depend not only on input parameter values, but also on the object state. The idea of an automaton is close to such program concepts as abstract data types and class objects, with the automaton being their mathematical model. For example, in object-oriented programming (OOP), one can consider an object of a class as an automaton: the state of the automaton is the object state, the input symbol is an operation in the object with some set of input parameter values, and the output symbol is the set of output parameter values.

The abstraction of automaton is used both for design and design solution analysis. The present paper is dedicated to problems arising in testing program systems that are considered finite automatons; we will call such a kind of testing "finite automaton testing."

The task of testing automation is divided into two relatively independent parts: test actions generation and automatic verification of the test action results, which is usually performed by an oracle program. Recently, the oracle construction problem has usually been treated as the problem of oracle generation using formal specifications. Many studies [1, 3, 6, 7] are dedicated to this problem, and we are not going to discuss it here. For our purposes, it is enough to know that, for any admissible pair, <state, input symbol>, an automaton oracle can define the validity of the transition to the new state (i.e., the validity of the transition function) and the validity of the resulting output symbol (i.e., the validity of the output function). The task of finite automaton testing can be solved under the assumption that its state is or is not available for observation from the testing system. If the state is unavailable (the "black box" automaton), then the testing system must introduce an "abstract state" that models the real state of the automaton. In this situation, the oracle calculates a new abstract state instead of checking the transition function. The correspondence of the real and abstract states is verified indirectly when the output function is checked in the oracle during the testing [3].

A natural criterion for the completeness of the test coverage the automaton testing is the coverage of all transitions of the automaton (i.e., admissible pairs, <state, input symbol>). To satisfy this criterion, it is necessary to generate all required test actions for all states of the automaton. Thus, the task of generation is divided into the task of "traversal" through all states of the automaton and the task of "searching" through the test actions for all states of the automaton. Note that the automaton output function is not used for performing these tasks.

One widespread method of automaton representation is the representation in the form of a graph of states (or transition graph) in which vertices correspond to the automaton states and arcs correspond to possible transitions. In terms of graph theory, the problem of coverage of all transitions of the automaton is formulated as the problem of graph traversal, i.e., passing over a route that contains all arcs of the graph.

There are two main problems related to the traversal of the automaton state graph: indetermism and the overly large size of the graph.

A nondeterministic automaton is one in which the transition function is ambiguous: several arcs of the graph correspond to a single pair <state, input symbol>. Since the choice of arcs is not determined by the test action, it is impossible to guarantee the full traversal of the graph (passing through all arcs and, thus, maybe through all states) during testing. Note that the ambiguity of the output function does not produce any additional problems: the oracle only requires that a certain predicate of the state, input, and output symbol be satisfied.

Remark: *The nondeterminacy of the modeling automaton does not necessarily imply that the object being modeled is nondeterministic. In many cases, the nondeterminacy of the model arises from the natural abstraction from the details of implementation. For example, in the request for memory, we may be not*

0361-7688/00/2602-0061$25.00 © 2000 MAIK "Nauka/Interperiodica"

interested in the algorithm of memory allocation; the only important thing is that the fragment to be allocated does not intersect with any of the fragments already allocated. Specifications often do not define operation results unambiguously because, as a rule, they describe only the requirements the result must obey rather than the algorithm for obtaining the result.

A very large size of the graph naturally leads to a very long traversal time (testing time).

There is a common approach to solving both problems stated above. It is based on the introduction of an equivalence relation of vertices and arcs of the graph. The criterion of coverage of all arcs and vertices is weakened to covering all equivalence classes. On these equivalence classes, the factor graph is constructed that is traversed in the process of testing. The homomorphism of the factor graph (under the incidence relation of vertices and arcs) to the original graph of the automaton states is substantially used in testing algorithms. With a proper definition of equivalence classes, the factor graph can become deterministic and its size can slump.

The concept of operation in an OOP object can serve as an example of decomposition of a set of arcs into equivalence classes. If certain predicates, which split operation domains onto subdomains, are defined on operation domains, then we obtain a more "detailed" decomposition.

Note that the equivalence with respect to operations can be considered as the equivalence input symbols: all *calls* of the same operation with different parameters are considered equivalent. However, the operation subdomain is, in general, a predicate of input parameters and the object state. That is why one should speak of equivalence of transitions of the automaton (i.e., that of the state graph arcs), rather than the equivalent of the input symbols.

We will consider finite automatons with strongly connected state graphs. Automatons corresponding to OOP objects possess this property (strong connectivity is easily obtained by adding the operations of construction and destruction of objects). Such automatons have the following property that is important for testing: after any transition, an opportunity remains to reach any state and test any transition from it.

In subsequent chapters, we will examine automaton state graphs, the testing of automatons using such graphs, factor graphs, testing of automatons using factor graphs, and methods for construction of factor graphs.

2. GRAPH OF AUTOMATON STATES

Furthermore, we will frequently use the following concepts and notations without additional comments:

• equivalence on a set. A is a reflexive, symmetric, and transitive binary relation, $\Sigma \subseteq A \times A$; relations are denoted by a capital Greek letter;

• equivalence Σ on a set A induces the decomposition of A into classes of Σ-equivalence (Σ classes). The set of these classes is denoted by A/Σ. Inversely, any decomposition, A, into non-intersecting classes induces the corresponding equivalence relation;

• the decomposition A/Σ induces the mapping, denoted by a small Greek letter σ: $A \longrightarrow A/\Sigma$. Inversely, any mapping induces a decomposition of the domain of definition and the corresponding equivalence on it;

• subset $\subseteq$, intersection $\cap$, and complement $\neg$, of equivalences are understood in the common set-theoretic sense (as for subsets of a Cartesian product).

Oriented graph, $G = (V, E, \lambda, \rho)$, is determined by two non-intersecting sets: the set of *vertices* V and the set of *arcs* E, and two *incidence functions* λ: $E \longrightarrow V$ and ρ: $E \longrightarrow V$. For any arc, the incidence functions define its initial vertex (*origin*) and its terminal vertex (*end*). A graph is finite if the sets E and V are finite.

The incidence functions λ and ρ define the *adjacency relation* of Ω arcs:

$$\forall e_1, e_2 \in E \quad e_1 \Omega e_2 \Leftrightarrow \rho(e_1) = \lambda(e_2).$$

We will say that a *coloring* (X, χ) is defined on the graph $G = (V, E, \lambda, \rho)$ if a set X, which we will call the *alphabet* of the coloring, and a mapping of the graph arcs onto this set, χ: $E \longrightarrow X$, are defined. We will call a coloring *regular* if multiple arcs are mapped into different elements (symbols) of X:

$$\forall e_1, e_2 \in E \lambda(e_1) = \lambda(e_2) \;\&\; \rho(e_1) = \rho(e_2)$$
$$\Rightarrow \chi(e_1) \neq \chi(e_2).$$

In a regularly colored graph, any arc e is unambiguously defined by the triple, (x, v, v'), where $x = \chi(e)$, $v = \lambda(e)$, and $v' = \rho(e)$.

A graph with an arbitrary set arc equivalence Σ is called Σ-*deterministic* if all arcs originating in the same vertex are Σ-nonequivalent:

$$\forall e_1, e_2 \in E \quad \lambda(e_1) = \lambda(e_2) \Rightarrow e_1 \neg \Sigma e_2.$$

It is clear that the mapping σ: $E \longrightarrow E/\Sigma$ determines a regular coloring with the alphabet E/Σ.

The *route* P is a sequence of adjacent arcs of the graph, $e_0, \ldots, e_t$, such that $e_{i-1} \Omega e_i$ for $1 \le i \le t$. If a regular coloring (X, χ) is defined on the graph, then the route can be defined as a sequence of triples (x_0, v_0, v_0'), $\ldots, (x_t, v_t, v_t')$, where $x_i = \chi(e_i)$, $v_i = \lambda(e_i)$, and $v_i' = \rho(e_i)$ for every $0 \le i \le t$ and $e_{i-1} \Omega e_i$ for any $1 \le i \le t$. The route P of a χ-deterministic graph can be determined by the initial vertex v_0 and a sequence of symbols in the alphabet (i.e., by a word in the alphabet X), $x_0, \ldots, x_t$: $P = (v_0, x_0, \ldots, x_t)$. The *traversal* of an oriented graph is a route that includes all arcs of the graph. For strongly connected finite graphs, a traversal always exists and can begin at any vertex.

An automaton $A = (X, V, Y, \phi, \psi, v_0)$, is defined as an aggregate of six objects:

- the input alphabet, X;
- the set of the automaton states, V;
- the output alphabet, Y;
- the correspondence $\phi \subseteq (X \times V) \times V$ called the transition function;
- the correspondence $\phi \subseteq (X \times V) \times Y$ that has the same domain of definition as ϕ ($Dom\phi = Dom\phi$). It is called the output function;
- the initial state $v_0 \in V$.

An automaton is *finite* if the sets X, V, and Y are finite.

An automaton is called *deterministic* if the transition function is single-valued:

$$\forall (x, v) \in Dom\,\phi\ \exists! v' \in V \quad \phi((x, v), v').$$

In this case, we will write $v' = \phi(x, v)$.

Remark: *The determinancy of an automaton is often understood as being the unambiguity of both the transition and the output functions. However, for our purposes, the unambiguity of the transition function is sufficient.*

For an automaton $A = (X, V, Y, \phi, \psi, v_0)$, we can assign the graph of its states $G = (V, E, \lambda, \rho)$ with a regular coloring (X, χ):

$$E = \{(x, v, v') \mid x \in X \& v \in V \& v' \in V \& \phi((x, v), v')\}$$

$$\lambda((x, v, v')) = v$$

$$\rho((x, v, v')) = v'$$

$$\chi((x, v, v')) = x.$$

We call χ-equivalent arcs having the common initial vertex Δ-equivalent. The equivalence class with respect to Δ, unlike the arc (x, v, v'), is unambiguously determined by the pair (x, v). If the automaton is deterministic, the graph of its states is Δ-deterministic, with every arc being Δ-equivalent to only itself.

3. TESTING AN AUTOMATON USING ITS GRAPH OF STATES

Testing of a deterministic automaton A on the basis of the traversal of its graph of states $P = (v_0, x_0, \ldots, x_t)$ is performed by the following algorithm, $\mathbb{A}(A, P)$:

1. On ith step of the algorithm (with $i = 0$ in the beginning), we are in the state v_i and we have to walk along the arc (x_i, v_i, v_{i+1}). Since the automaton is deterministic, this arc is unambiguously determined by the pair (x_i, v_i).

2. We supply the symbol x_i at the automaton input and apply the automaton oracle.

3. If the oracle produces a negative verdict, the testing process ends with an error discovered. Otherwise, we get a new state, v_{i+1}, from the oracle.

4. If $i < t$, we increase i by 1 and proceed to step 1. Otherwise, the testing process is considered to be completed normally.

The testing time is determined by the length, t, of the traversal of P. For graphs containing n vertices and m arcs, it is well-known that the optimal traversal length has the order of nm.

4. HOMOMORPHIC GRAPH

4.1. Homomorphism of Graphs and Factor Graph

Homomorphism (ξ, π) of a graph G onto a graph G^* is a pair of surjective mappings of vertices, ξ, and arcs, π, preserving the incidence functions, λ and ρ:

$$\forall e \in E\ \ \xi(\lambda(e)) = \lambda(\pi(e)) \ \&\ \xi(\rho(e)) = \rho(\pi(e)).$$

The homomorphism (ξ, π) of the graph G onto the graph G^* induces the congruence (Ξ, Π) on G, which, in turn, defines the factor graph $G/(\Xi, \Pi)$ isomorphic to G^* and the canonical homomorphism G onto $G/(\Xi, \Pi)$. That is why we will further consider, as a rule, the homomorphism of a graph onto its factor graph retaining the same notation G^*, ξ, and π. However, in the testing algorithm, we use the homomorphism of G onto any G^* (not necessarily a factor graph).

The coloring (X^*, χ^*) of the factor graph G^* induces a coloring (X^*, θ)on the graph G; here $\theta = \chi^*\pi$. If the coloring (X^*, χ^*) is regular, then, for every arc $e \in E$, the triple $(\theta(e), (\xi(\lambda(e)), \xi(\rho(e)))$ unambiguously determines the factor arc $e^* = \pi(e)$. We will call the alphabet X^* the generalized alphabet to distinguish it from the alphabet X, whose symbols are used to color the arcs of the state graph G. Note that the generalized alphabet X^*, in general, is not a factor alphabet (i.e., it is not a decomposition of the alphabet X).

Inversely, let a vertex equivalence Ξ and an arc coloring (X^*, θ), which induces the corresponding arc equivalence Θ, be given on the graph G. The equivalence Ξ induces the canonical equivalence of arcs $\Xi^\sim$: two arcs are $\Xi^\sim$-equivalent if their initial and terminal vertices are Ξ-equivalent. It is the intersection of equivalences, $\Theta^\sim = \Theta \cap \Xi^\sim$, that generates the decomposition of arcs into factor arcs: $E^* = E/\Theta^\sim$, that is, $\Pi = \Theta^\sim$. We will speak of the homomorphism (ξ, θ) as meaning the induced homomorphism $(\xi, \theta^\sim)$.

If only a vertex equivalence Ξ and an arc equivalence Θ are given on the graph $G = (V, E, \lambda, \rho)$, one can independently define the factor graph $G^* = (V, E, \lambda, \rho, \Xi, \Theta) = (V^*, E^*, \lambda, \rho)$:

- factor vertex $v^* \in V^*$ is a set of Ξ-equivalent vertices;
- factor arc $e^* \in E^*$ is a set of Θ-equivalent arcs with Ξ-equivalent origins and ends;

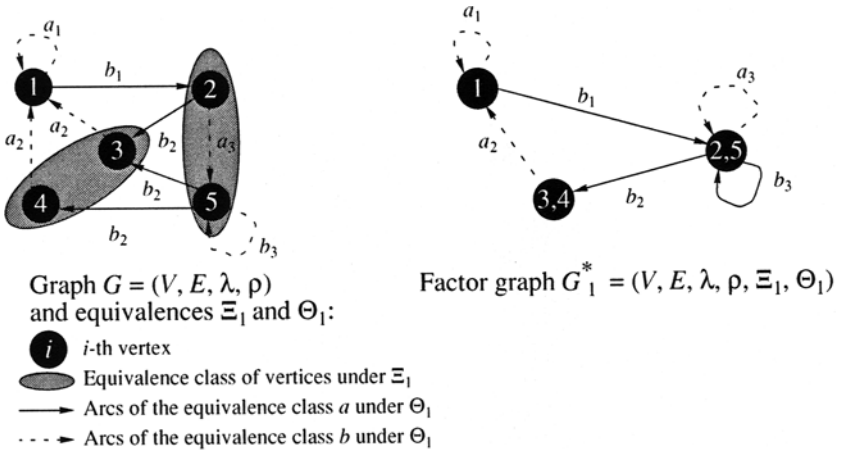


Fig. 1. An example of factor graph.

• if e^* is a factor arc, then $\lambda(e^*)$ is the factor vertex where all arcs from e^* begin;

• if e^* is a factor arc, then $\rho(e^*)$ is the factor vertex where all arcs from e^* end.

For $X^* = X/\Theta$, the coloring (X^*, θ) induces a regular coloring of the factor graph (X^*, θ^*): $\theta(e) = x^* \Rightarrow \theta^*(\theta^{\sim}(e)) = x^*$.

Remark: *Under a proper definition of factor arc coloring by generalized output symbols, one can assign a generalized automaton to the factor graph. In some sense, testing of the initial automaton by its factor graph can be considered as the testing of the corresponding generalized automaton.*

4.2. Deterministic Factor Graph

For an arbitrary equivalence Σ on the set of arcs of the graph G, we call the factor graph $G^* = (V, E, \lambda, \rho, \Xi, \Theta)$, *$\Sigma$-deterministic* if Σ-equivalent arcs with Ξ-equivalent initial arcs belong to the same factor arc; i.e., are Θ-equivalent and have Ξ-equivalent terminal vertices. The corresponding homomorphism of the graphs is called Σ-deterministic.

We will be interested in Δ-determinism and Θ-determinism of factor graphs. Note that Θ-determinism of a factor graph coincides with its Θ^*-determinism when it is considered as a graph; i.e., the image of the Θ-deterministic homomorphism is Θ^*-deterministic.

The criterion for Δ-determinism is the condition $\Delta \subseteq \Theta^{\sim}$. It follows from $\Delta \subseteq \Theta^{\sim}$ and $\Theta^{\sim} \subseteq \Theta$ that $\Delta \subseteq \Theta$. Note that χ-determinism does not follow from Δ-determinism.

The criterion for Θ-determinism is the following condition:

$$\forall e_1, e_2 \in E \;\; \lambda(e_1)\Xi\lambda(e_2) \;\&\; \rho(e_1) \neg \Xi\rho(e_2) \Rightarrow e_1 \neg \Theta e_2.$$

In a Θ-deterministic factor graph, the factor arc $(\theta^*(e^*), \lambda(e^*), \rho(e^*))$ is unambiguously defined by the pair $(\theta^*(e^*), \lambda(e^*))$.

For example, in the original graph G in Fig. 1, two arcs b_2 originating from vertex $\boxed{5}$, are Δ-equivalent; these arcs correspond to the same symbol b_2. The graph G_1 is Δ-nondeterministic. However, the factor graph G_1^* is Δ-deterministic, though it is not Θ_1-deterministic since the two Θ_1-equivalent arcs, b_2 and b_3, have Ξ_1-equivalent initial vertices (the factor vertex $\boxed{2, 5}$), but Ξ_1-nonequivalent terminal vertices ($\boxed{3, 4}$ and $\boxed{2, 5}$).

We will call a factor graph simply *deterministic* if it is Δ-deterministic and Θ-deterministic at the same time. The corresponding homomorphism of graphs will be called a deterministic homomorphism.

4.3. Completely Definite Factor Graph

Since the factor graph G^* is a homomorphic image of the original graph G, any route P in G is mapped onto a certain route P^* in the factor graph G^*. If P contains only some vertices and arcs but passes through all equivalence classes of Ξ and $\Theta^{\sim}$ vertices and arcs, then P^* is a traversal of the factor graph G^*. It is the route P that will traversed in the process of testing, and the successful completion of the traversal P^* route is the criterion of the testing completeness. Since the factor graph G^* contains fewer vertices and arcs than G, P^* (and, thus, P) is shorter than the traversal of G. That is, the testing time by the factor graph is less than that by the original graph.

However, not every route, P^* in the factor graph G^* has to be the image of a route in G. Here the problem of adjacency of arcs arises: a pair of adjacent factor arcs e_1^* and e_2^* that follow one another in the traversal P^* can be the image the pair of arcs $e_1 \in e_1^*$ and $e_2 \in e_2^*$ in G that are not adjacent.

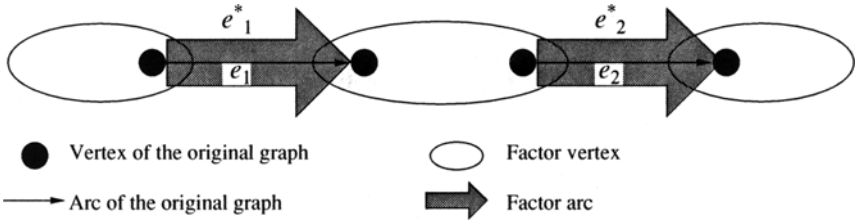


Fig. 2.

There are several approaches to the solution of the problem.

One of them consists in that every time one has to pass over nonadjacent arcs e_1 and e_2, a path in G leading from the terminal vertex of e_1 the initial vertex of e_2 is used [4]. Certainly, this causes a cyclic path to be passed in the factor graph G^*. Thus, instead of passing over P^*, we actually go a longer route P_1^*, which is a homomorphic image of P. This approach is applicable for any factor graph (if the original graph is strongly connected). However, the length of P_1^* (which gives an estimate for the testing time) can be much greater than the length of P^* so that it approaches the length of the route on the original graph G; thus, all the benefit from using the factor graph instead of the original graph is lost.

Another approach consists in the consideration of only factor graphs for which the requirement of *strict adjacency of arcs* is satisfied: if the factor arcs e_1^* and e_2^* are adjacent and $e_1 \in e_1^*$, then $e_2 \in e_2^*$ exists such that e_1 and e_2 are adjacent. In this case, every traversal P^* of the factor graph is an image of at least one route P in the original graph. Thus, the testing time is determined by the length of P^*, which can be significantly less than the length of the traversal of the original graph. It is this approach that is examined in the present paper.

We call a factor arc e^* *completely definite* if the set of initial vertices of arcs $e \in e^*$ coincides with the initial factor vertex of e^*: $\{\lambda(e) | e \in e^*\} = \lambda(e^*)$. A factor graph is completely definite and Θ is a completely definite arc equivalence if all factor arcs are completely definite. The corresponding homomorphism is called completely definite. For a strongly connected graph G, the strict adjacency of the factor graph G^* arcs is equivalent to its complete definiteness.

For example, for the factor graph G_1^* in Fig. 1, the factor arcs a_1, b_1, a_2, and b_2 are completely definite, and a_3 and b_3 are not completely definite.

In terms of graph homomorphisms, one can interpret the strict adjacency of arcs and the complete definiteness as follows.

We call the homomorphism (of algebraic systems) $\tau: A \longrightarrow A^*$ a *strict* (from left) with respect to Σ if:

$$\forall x \in A, y^* \in A^* \tau(x) \Sigma y^*$$
$$\Rightarrow \exists y \in A \tau(y) = y^* \;\&\; x \Sigma y.$$

Strict adjacency of arcs means that the homomorphism of graphs with respect to the equivalence Ω of arc adjacency is strict; complete definiteness means that the homomorphism of graphs by the incidence function λ is strict, with the incidence function interpreted as the relation $v\lambda e$. Note that if the incidence λ is interpreted as the relation $e\lambda v$, then the corresponding property is true for any homomorphism of graphs.

5. AUTOMATON TESTING USING A HOMOMORPHIC GRAPH

5.1. Homomorphism of Graphs and Symbol Calculation Function

Consider a deterministic completely definite homomorphism, (ξ, θ) of the graph G of states of an automaton A onto the graph G^*. It follows from the complete definiteness that any traversal P^* of G^* is the image of a certain route P in the original graph G. From the Θ-determinism, it follows that the traversal P^* can be determined by the initial vertex v_0^* of G^* and the sequence of generalized symbols (a generalized word) of the form $x_0^*, \ldots, x_t^*$. In a strongly connected graph, traversal can be started from any vertex; we choose the image of the initial state v_0 of A (i.e., $\xi(v_0)$) as the starting vertex of P^*. Then the traversal can be determined as $P^* = (v_0, x_0^*, \ldots, x_t^*)$.

In the testing algorithm, the *symbol calculation function* is used. Given an arc $(x^*, v^*, v^{*\prime})$ of G^*, a generalized symbol x^*, and the automaton state v from the preimage of v^*, this function calculates the (input) symbol x of the automaton so that it is guaranteed that any arc (x, v, v') belongs to the preimage of (x^*, v^*, v'). The following requirements must be satisfied:

- first, the pair $(x^*, \xi(v))$ defines only one arc, $(x^*, v^*, v^{*\prime})$. This is guaranteed by the graph Θ-determinism.
- second, for any v from the preimage of v^*, there must be an arc, (x, v, v'), that is mapped into $(x^*, \xi(v))$.

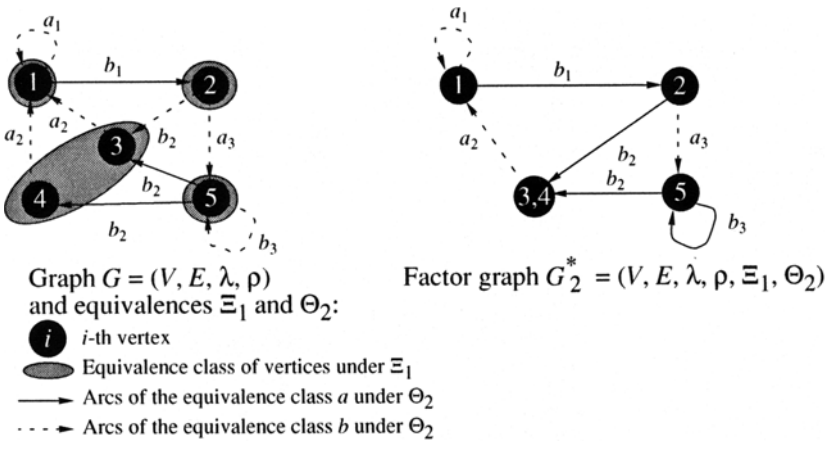


Fig. 3.

This is guaranteed by the complete definiteness of the homomorphism.

• Third, the entire Δ-class (x, v) to which such an arc (x, v, v'), belongs must be mapped is reflected into a single arc $(x^*, \xi(v))$. This is guaranteed by the Δ-determinism of the homomorphism.

Under these conditions, the symbol calculation function finds at least one solution to the equation $\theta(x, v) = x^*$ in x.

Remark: *The symbol calculation function can be implemented as a function that tries all symbol x and tests, whether* $\theta(x, v) = x^*$.

Thus, in fact, the testing algorithm uses only the predefined mappings ξ and θ (i.e., the homomorphism of the original graph G onto G^*), and the given traversal P^* of G^*.

5.2. The Testing Algorithm

Let a deterministic and completely definite homomorphism (ξ, θ) of the state graph $G = (V, E, \lambda, \rho)$ onto the graph $G^* = (V^*, E^*, \lambda, \rho)$ be given for an automaton A. The automaton testing on the basis of the traversal $P^* = (v_0, x_0^*, \ldots, x_t^*)$ on the G^* graph is performed by the following algorithm $\mathbb{A}(A, P^*)$.

1. On the ith step of the algorithm ($i = 0$ at the beginning), we are in the state v_i, and we are going to pass the arc e^* G^* that is unambiguously defined by the pair $(x_i^*, \xi(v_i))$. Using the symbol calculation function, we define x_i, which is the solution to the equation $\theta(x_i, v_i) = x_i^*$.

2. We send the symbol x_i to the input of the automaton and apply the automaton oracle.

3. If the oracle produces a negative verdict, the testing is finished and an error is fixed. Otherwise, we obtain a new state v_{i+1} from the oracle such that $\xi(v_{i+1}) = \rho(e^*)$.

4. If $i < t$, we increase i by 1 and proceed to step 1. Otherwise, the testing is considered to be completed normally.

In the process of testing, we traverse G^* by the route $P^* = (v_0, x_0^*, \ldots, x_t^*)$. In the process, the route $P = (v_0, x_0, \ldots, x_t)$ in the state graph of the automaton A is passed.

5.3. An Example of Testing Using Homomorphic Graph

Consider the factor graph $G_1^* = (V, E, \lambda, \rho, \Xi_1, \Theta_1)$ shown in Fig. 1. This factor graph is deterministic but not completely definite. We modify the relation Ξ_1 so as to obtain the completely definite factor graph $G_2^* = (V, E, \lambda, \rho, \Xi_2, \Theta_1)$ as shown in Fig. 3. For this purpose, it is sufficient to treat vertices $\boxed{2}$ and $\boxed{5}$ as being nonequivalent.

There exist several traversal of G_2^*. For example, P_2^*:

$$\boxed{1} \xrightarrow{a_1} \boxed{1} \xrightarrow{b_1} \boxed{2} \xrightarrow{b_2} \boxed{3,4} \xrightarrow{a_2} \boxed{1} \xrightarrow{b_1} \boxed{2}$$
$$\xrightarrow{a_3} \boxed{5} \xrightarrow{b_3} \boxed{5} \xrightarrow{b_2} \boxed{3,4}$$

The following two possible routes, P_{21} and P_{22}, in the original graph G correspond to P_2^*:

$$P_{21} = \boxed{1} \xrightarrow{a_1} \boxed{1} \xrightarrow{b_1} \boxed{2} \xrightarrow{b_2} \boxed{3} \xrightarrow{a_2} \boxed{1} \xrightarrow{b_1} \boxed{2}$$
$$\xrightarrow{a_3} \boxed{5} \xrightarrow{b_3} \boxed{5} \xrightarrow{b_2} \boxed{3}$$
$$P_{22} = \boxed{1} \xrightarrow{a_1} \boxed{1} \xrightarrow{b_1} \boxed{2} \xrightarrow{b_2} \boxed{3} \xrightarrow{a_2} \boxed{1} \xrightarrow{b_1} \boxed{2}$$
$$\xrightarrow{a_3} \boxed{5} \xrightarrow{b_3} \boxed{5} \xrightarrow{b_2} \boxed{4}$$

These routes differ in the last transition from the vertex $\boxed{5}$ to the vertex $\boxed{3}$ or $\boxed{4}$ by one of the arcs b_2 of same Δ-class. Thus, the choice of P_{21} and P_{22} is not

deterministic. Note that, in both cases, not all arcs of the original graph are passed (arc $\boxed{4} \xrightarrow{a_1} \boxed{1}$ and one of the arcs of the Δ-class, $\boxed{5} \xrightarrow{b_2} \boxed{4}$ or $\boxed{5} \xrightarrow{b_2} \boxed{3}$).

6. CONSTRUCTION OF A TRAVERSAL IN THE PROCESS OF TESTING

The general problem of constructing a traversal of a graph is well known. Here we examine the problem of constructing a traversal of a graph in the process of testing.

6.1. Testing an Automaton Using Its Graph of States

There is a tool implementing the universal algorithm $\mathbb{A}(A, \varnothing)$ for testing any automaton A with deterministic, finite, and strongly connected state graph; simultaneously, a traversal of the state graph is constructed [3]. In contrast to the algorithm $\mathbb{A}(A, P)$, $\mathbb{A}(A, \varnothing)$ does not require the traversal P to be given (it is constructed automatically). Moreover, the algorithm $\mathbb{A}(A, \varnothing)$ does not use any information about the automaton and its graph of states except for what is defined by the following parameters of the algorithm:

- the automaton initial state;
- the operation of state comparison for equality;
- input symbols iterator;
- the automaton oracle.

Given a pair ⟨state, input symbol⟩, the *iterator* of input symbols produces the following input symbol admissible for the given state:

$$It: V \times X \cup \{\varnothing\} \longrightarrow X \cup \{\varnothing\}$$

To obtain the initial input symbol, an "empty" symbol ∅ is specified that is added to the input alphabet. The iterator must guarantee that, for every v from V, the sequence $It(v, \varnothing)$, $It(v, It(v, \varnothing))$, ... runs through all symbols admissible in the state v. At the end of this sequence, the iterator returns the empty input symbol ∅.

Remark: *The iterator may run through all input symbols, rather than only through those allowed in the given state. In this case, a special* **admissibility verification function** *is used to filter out inadmissible input symbols. In terms of formal specifications, the admissibility verification is the verification of the operation precondition.*

The algorithm learns about admissible states of the automaton when, having sent an admissible input symbol to the automaton input, it gets the value of the state the automaton goes to from the oracle.

The length of the traversal constructed by the algorithm $\mathbb{A}(A, \varnothing)$ has the length of the same order that the optimal traversal of the graph (the product of the number of vertices by the number of arcs).

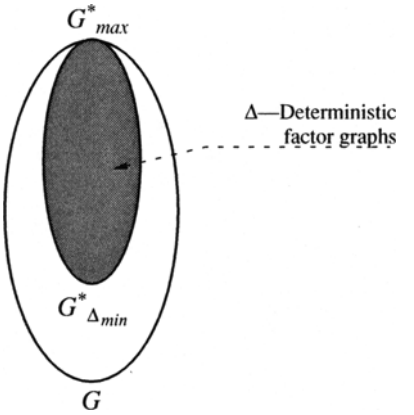


Fig. 4.

6.2. Testing of an Automaton Using Homomorphic Graph

There is a modification of this algorithm for testing by the homomorphic image of the graph. This modification is denoted by $\mathbb{A}^*(A, \varnothing)$. The homomorphism (ξ, θ) of the graph of states G of the automaton A onto the graph G^* must be deterministic and completely definite, and G^* must be finite and strongly connected. Unlike the algorithm $\mathbb{A}^*(A, P^*)$, $\mathbb{A}^*(A, \varnothing)$ uses only the given mappings ξ and θ; i.e., it uses the specified homomorphism of the original graph G onto G^* and does not require the traversal P^* of G^* to be given (the traversal is constructed automatically). Moreover, the algorithm $\mathbb{A}^*(A, \varnothing)$ does not use any information about the automaton, its state graph, and the homomorphic graph except for what is defined by the following parameters of the algorithm:

- the automaton initial state;
- the mapping $\xi: V \longrightarrow V^*$ of the state of the graph G into the state of G^*;
- the generalized input symbol iterator (iterator through X^*);
- the symbol calculation function which, in turn, depends only on the mappings ξ and θ;
- the automaton oracle.

Instead of the mapping ξ, one can use a predicate of two variables on the set of states V implementing the Ξ-equivalence similar to how the operation of comparison for equality is used in the algorithm $\mathbb{A}(A, \varnothing)$.

The traversal of the homomorphic graph G^* that is constructed automatically by $\mathbb{A}^*(A, \varnothing)$ has the length of the same order as the optimal traversal of G^* (the product of the number of vertices by the number of arcs).

7. CONSTRUCTION OF DETERMINISTIC COMPLETELY DEFINITE FACTOR GRAPHS

Suppose that equivalences of vertices Ξ and arcs Θ of an automaton state graph $G = (V, E, \lambda, \rho)$ are defined

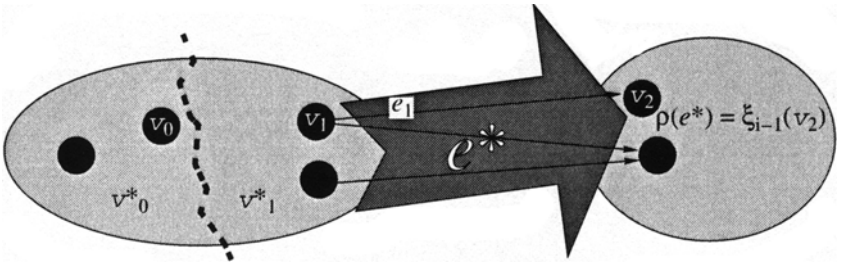


Fig. 5.

on the basis of a certain criterion of test coverage. These equivalences define the following *boundary* requirement: nonequivalent vertices and arcs must be distinguished during the testing. However, it does not mean that we cannot make a more detailed differentiation of vertices and arcs; i.e., use equivalences $\Xi' \subseteq \Xi$ and $\Theta' \subseteq \Theta$. Partly, it already happens in testing by the homomorphic graph; in fact, we distinguish arcs by the equivalence $\Theta^{\sim} \subseteq \Theta$, rather than by Θ. In the general case, the boundary requirement is satisfied for any graph embedded into G^* if graph embedding is defined through the homomorphism mappings or, which is the same, through embedding of equivalences Ξ and $\Theta^{\sim}$ that define it: $G^{*'} \subseteq G^* \Leftrightarrow \Xi' \subseteq \Xi$ & $\Theta'^{\sim} \subseteq \Theta^{\sim}$. (Note that $\Theta' \subseteq \Theta \Rightarrow \Theta'^{\sim} \subseteq \Theta^{\sim}$, but the inverse is not true.)

If, therefore, for a given *initial* equivalences Ξ and Θ, the factor graph $G^* = (V, E, \lambda, \rho, \Xi, \Theta)$ turned out to be Δ-nondeterministic and/or Θ-nondeterministic and/or not completely definite, we can try to construct equivalences $\Xi_{\Delta fd}$ and $\Theta_{\Delta fd}$ such that $\Xi_{\Delta fd} \subseteq \Xi$, $\Theta_{\Delta fd} \subseteq \Theta$, and the factor graph $G^*_{\Delta fd} = (V, E, \lambda, \rho, \Xi_{\Delta fd}, \Theta_{\Delta fd})$ is deterministic and completely definite.

Consider a factor graph $G^*_{\Delta f} = (V, E, \lambda, \rho, \Xi_{\Delta f}, \Theta_{\Delta f})$ that is Δ-deterministic and completely definite, but not $\Theta_{\Delta f}$-deterministic. In this case, we can choose any

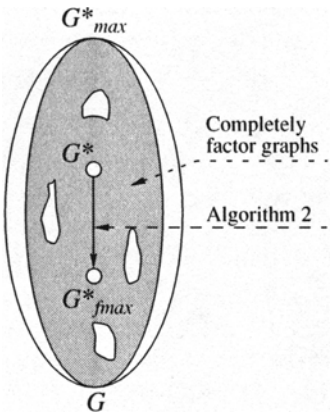


Fig. 6.

equivalence $\Theta_{\Delta fd}$ between $\tilde{\Theta}_{\Delta f}$ and $\Theta_{\Delta f}$ satisfying the $\Theta_{\Delta fd}$-determinism criterion

$$\forall e_1, e_2 \in E \;\; \lambda(e_1)\Xi_{\Delta f}\lambda(e_2)$$
$$\& \; \rho(e_1) \neg \Xi_{\Delta f}\rho(e_2) \Rightarrow e_1 \neg \Theta_{\Delta fd} e_2 .$$

Obviously, such an equivalence exists; e.g., $\Theta_{\Delta fd} = \tilde{\Theta}_{\Delta f}$. In general, all such equivalences $\Theta_{\Delta fd}$ are easily obtained by determining all possible subdecompositions of the set of factor arcs originating from a single factor vertex and the $\Theta^*_{\Delta f}$ equivalent.

So, further, we solve the problem of constructing a Δ-deterministic and completely definite factor graph $G^*_{\Delta f} = (V, E, \lambda, \rho, \Xi_{\Delta f}, \Theta_{\Delta f})$. Since the embedded factor graph has the number of vertices and arcs that is not less (greater, if the embedding is strict) than the initial graph, it is natural to construct maximum (with respect to embedding) equivalences $\Xi_{\Delta fmax}$ and $\Theta_{\Delta fmax}$.

7.1. The Δ-determinism Criterion of the Factor Graph

It is clear that if $G'^* \subseteq G^*$ and G^* is Δ-nondeterministic, then G'^* is also Δ-nondeterministic: $\Theta'^{\sim} \subseteq \Theta^{\sim}$ & $\neg(\Delta \subseteq \Theta^{\sim}) \Rightarrow \neg(\Delta \subseteq \Theta'^{\sim})$.

Since $\Theta^{\sim} = \Theta \cap \Xi^{\sim}$, the Δ-determinism condition, $\Delta \subseteq \Theta^{\sim}$, means the fulfillment of two embedding conditions: $\Delta \subseteq \Theta$ and $\Delta \subseteq \Xi^{\sim}$. We now show that there exists a minimum equivalence of vertices, $\Xi_{\Delta min}$, such that it is necessary and sufficient for $\Delta \subseteq \Xi^{\sim}$; that is, $\Delta \subseteq \Xi^{\sim} \Leftrightarrow \Xi_{\Delta min} \subseteq \Xi$.

Algorithm 1: Construction of the equivalence $\Xi_{\Delta min}$.

1. We declare the terminal vertices of $\Xi_{\Delta min}$-equivalent arcs to be Δ-equivalent;

2. Find the transitive closure of this reflexive and symmetric relation to obtain the required equivalence.

Thus, the Δ-determinism criterion is formulated as the fulfillment of two embedding conditions: $\Delta \subseteq \Theta$ and $\Xi_{\Delta min} \subseteq \Xi$.

This condition is satisfied for all factor graphs in the range $[G^*_{\Delta min}, G^*_{max}]$ with respect to embedding, where $G^*_{\Delta min} = (V, E, \lambda, \rho, \Xi_{\Delta min}, \Delta)$ and $G^*_{max} = (V, E, \lambda, \rho,$

Ξ_{max}, Θ_{max}); here Ξ_{max} and Θ_{max} are maximum equivalences of vertices and arcs (all vertices and arcs are equivalent). The graph G^*_{max} contains one factor vertex and one factor arc (loop).

7.2. Constructing a Completely Definite Factor Graph

It is clear for any factor graph $G^* = (V, E, \lambda, \rho, \Xi, \Theta)$, an embedded completely definite factor graph exists. An example is the state graph G itself. We will construct a maximum (with respect to embedding), completely definite factor graph $G^*_{f\,max} \subseteq G^*$. In the process, we will only change the equivalence of vertices; that is, $G^*_{f\,max} = (V, E, \lambda, \rho, \Xi_{fmax}, \Theta)$, where $\Xi_{fmax} \subseteq \Xi$.

Algorithm 2: Constructing the $G^*_{f\,max}$ factor graph.

The algorithm begins with the factor graph $G^*_0 = G^* = (V, E, \lambda, \rho, \Xi, \Theta)$ and consists of a sequence of steps of the same kind. Every ith step of the algorithm transforms the factor graph G^*_{i-1} into the factor graph G^*_i. The following condition is called the embedding condition: every completely definite factor graph embedded in G^* is embedded into G^*_i. To prove that our algorithm constructs the factor graph $G^*_{f\,max}$, it is sufficient to prove that the algorithm satisfies the following conditions:

1. In the beginning of the algorithm work, the embedding condition is satisfied for the factor graph G^*_0 (which is obvious).

2. Every ith step of the algorithm retains the embedding condition for the factor graph G^*_i if this condition is satisfied in the beginning of the step (for G^*_{i-1}).

3. If G^*_i is completely definite, the algorithm terminates after the ith step.

4. The algorithm terminates after a finite number of steps.

The ith step of the algorithm consists in the following:

If $G^*_{i-1} = (V, E, \lambda, \rho, \Xi_{i-1}, \Theta)$ is a completely definite factor graph, then the algorithm terminates (and, thus satisfies condition 3). Otherwise, a not completely definite factor arc $e^* \in E/\tilde{\Theta}_{i-1}$ exists; that is, $\{\lambda(e)|e \in e^*\} \neq \lambda(e^*)$. Now we decompose the factor vertex $v^* = \lambda(e^*)$ into two subsets, $v^*_1 = \{\lambda(e)|e \in e^*\}$ and $v^*_0 = v^* \setminus v^*_1$. Thus, we obtain a new equivalence Ξ_i and a

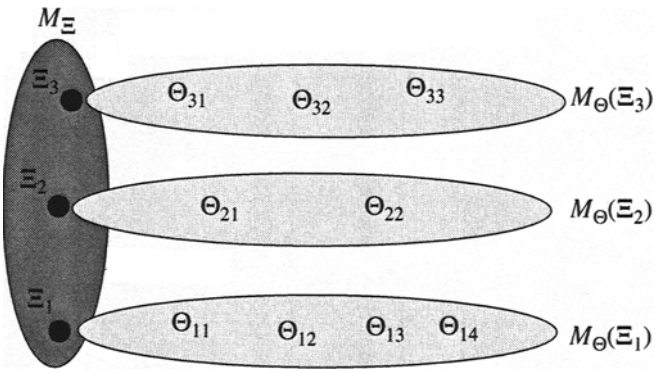
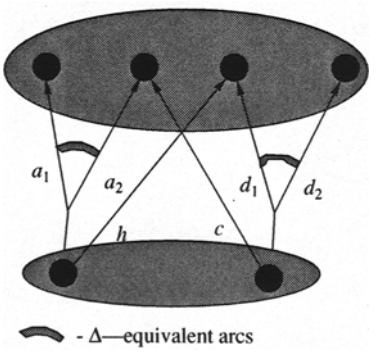


Fig. 7.



Decompositions factor arcs preserving Δ-determinism and complete definiteness:
- $(a1, a2, c)(b, d1, d2)$
- $(a1, a2, d1, d2)(b, c)$

Fig. 8.

new factor graph $G^*_i = (V, E, \lambda, \rho, \Xi_i, \Theta)$:

$$\forall v_1, v_2 \in V v_1 \Xi v_2 \Leftrightarrow (v_1 \Xi_{i-1} v_2)$$

$$\& \neg(v_1 \in v^*_0 \;\&\; v_2 \in v_1 \vee v_1 \in v^*_1 \;\&\; v_2 \in v^*_0).$$

We need to prove condition 2: if the embedding condition was satisfied for G^*_{i-1}, then it is satisfied for G^*_i.

Consider any two vertices $v_1 \in v^*_1$ and $v_0 \in v^*_0$. For v_1, an arc $e_1 \in e^*$ exists such that it has v_1 as its initial and $v_2 \in \rho(e^*)$ as its terminal vertex. It is obvious that $\rho(e^*) \in \xi_{i-1}(v_2)$. Since the embedding condition is satisfied for G^*_{i-1}, the condition $\xi_f(v_2) \subseteq \xi_{i-1}(v_2)$ must be satisfied for the vertex v_2 of any completely definite factor graph G^*_f. No arc from the set e^* originates in the vertex v_0. Since e^* contains (by the definition of the factor arc) all Θ-equivalent arcs leading from v^* to $\rho(e^*)$, no arcs originate in v_0 that are Θ-equivalent to the arc e_1 and lead to $\rho(e^*) = \xi_{i-1}(v_2)$, much the less, to into its subset $\xi_f(v_2)$. Thus, the arc e_1 leads from v_1 to $\xi_f(v_2)$, while no arc Θ-equivalent to e_1 leads from v_0 to $\xi_f(v_2)$. Hence, due to complete definiteness, the vertices v_1 and v_0 cannot belong to the same factor vertex of the factor graph G^*_f.

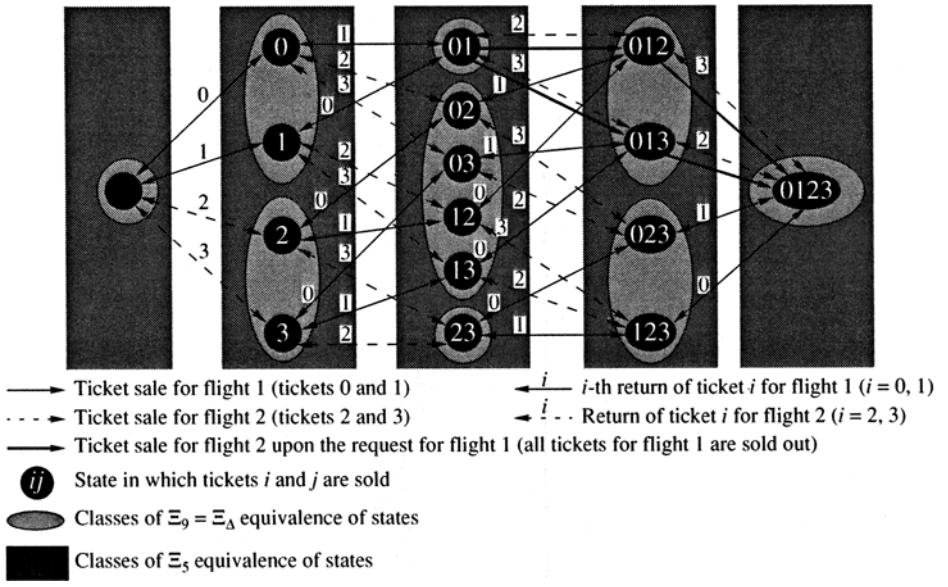


Fig. 9. An example of graph of states.

Therefore, any factor vertex of any completely definite factor graph $G_f^* \subseteq G_{i-1}^*$ embedded in v^* must be embedded in v_1^* or v_0^*. In other words, the decomposition of the factor vertex v^* into two factor vertices v_1^* or v_0^* does not decompose any factor vertex of any completely definite factor graph $G_f^* \subseteq G_{i-1}^*$. Thus, the embedding condition is retained.

It remains to prove condition 4: the algorithm terminates in a finite number of steps. Indeed, every step of the algorithm increases the number of factor vertices in the factor graph, and that number is limited above by the number of vertices of the graph G (the graph is finite).

This completes the proof of conditions 1–4.

All completely definite factor graphs lie in the range $[G, G_{max}^*]$; however, not all factor graphs in this range are completely definite. Nevertheless, the Algorithm 2 shows that, for any factor graph in the range $[G, G_{max}^*]$, one can construct an embedded, completely definite factor graph $G_{f\,max}^*$.

7.3. Existence Criterion and Algorithm of Constructing Δ-deterministic and Completely Definite Factor Graph

Summarizing the above subsections, one can say that not for any factor graph G^* of a given graph G there exists a Δ-deterministic, completely definite factor graph $G_{\Delta f}^* \subseteq G^*$. Algorithm 2 constructs a maximum, completely definite factor graph $G_{f\,max}^* \subseteq G^*$, which can be tested for Δ-determinism using the criterion $\Delta \subseteq \Theta_{fmax}$ & $\Xi_{\Delta max} \subseteq \Xi_{fmax}$, where Ξ_{min} is constructed from Ξ by Algorithm 1. Thus, if factor graphs $G_{\Delta f}^*$ exist, then $G_{f\,max}^*$ is maximal among them; otherwise, $G_{f\,max}^*$ is Δ-nondeterministic.

On the other hand, for any graph $G = (V, E, \lambda, \rho)$, one can find such equivalences $\Xi_{\Delta f}$ and $\Theta_{\Delta f}$ that the corresponding factor graph $G_{\Delta f}^* = (V, E, \lambda, \rho, \Xi_{\Delta f}, \Theta_{\Delta f})$ is Δ-deterministic and completely definite. An example of such a factor graph is provided by $G_{max}^* = (V, E, \lambda, \rho, \Xi_{max}, \Theta_{max})$, in which all vertices are equivalent and all arcs are equivalent.

It is easy to show that, for any Δ-deterministic and completely definite factor graph $(V, E, \lambda, \rho, \Xi_{\Delta f}, \Theta_{\Delta f})$, the factor graph that corresponds to the pair $(\Xi_{\Delta f}, \Theta_{max})$ is also Δ-deterministic and completely definite. Thus, the set of all Δ-deterministic and completely definite factor graphs can be described in two stages:

- One describes the set M_Ξ of equivalences of vertices $\Xi_{\Delta f}$ for which the factor graph, constructed using $\Xi_{\Delta f}$ and Θ_{max}, is Δ-deterministic and completely definite.
- For $\Xi_{\Delta f} \in M_\Xi$, one describes the set $M_\Theta(\Xi_{\Delta f})$ of equivalences of arcs $\Theta_{\Delta f}$ for which the factor graph $(V, E, \lambda, \rho, \Xi_{\Delta f}, \Theta_{\Delta f})$ is Δ-deterministic and completely definite.

The set M_Ξ lies in the range $[\Xi_{\Delta min}, \Xi_{\Delta max}]$. If $(V, E, \lambda, \rho, \Xi_{\Delta min}, \Theta_{max})$ is not completely definite, then the interval is open from below: $(\Xi_{\Delta min}, \Xi_{max}]$. For all Ξ in this range, the factor graph $(V, E, \lambda, \rho, \Xi, \Theta_{max})$ is Δ-deterministic, but not necessarily completely definite. For any Ξ_Δ in this range, Algorithm 2 constructs the maximum embedding $\Xi_{fmax} \subseteq \Xi_\Delta$ for which the factor graph $(V, E, \lambda, \rho, \Xi_{fmax}, \Theta_{max})$ is completely definite.

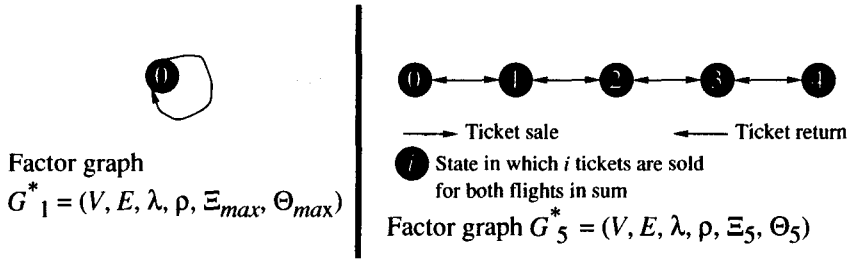


Fig. 10. Factor graphs.

However, Ξ_{fmax} can lie beyond the range: $\Xi_{\Delta min}$ is not embedded in Ξ_{fmax}; thus, $(V, E, \lambda, \rho, \Xi_{fmax}, \Theta_{max})$ is Δ-nondeterministic.

The equivalence of vertices $\Xi_{\Delta f} \in M_\Xi$ induces the equivalence of arcs $\tilde{\Xi}_{\Delta f}$, which determines a certain basic decomposition of arcs for which the factor graph $(V, E, \lambda, \rho, \Xi_{\Delta f}, \tilde{\Xi}_{\Delta f})$ is Δ-deterministic and completely definite. For any basic arc e^*, the equivalence of arcs $\Theta_{\Delta f} \in M_\Theta(\Xi_{\Delta f})$ determines a decomposition for which Δ-determinism and complete definiteness are not broken. Δ-determinism is not broken if the decomposition of e^* does not decompose embedded Δ-classes. Complete definiteness is not broken if every class of the decomposition of e^* includes at least one arc leading from every vertex, $v \in \lambda(e^*)$.

For $\Theta_{\Delta f} \in M_\Theta(\Xi_{\Delta f})$, one can consider equivalences $\Theta_{\Delta fd}$ satisfying the following conditions:

$$\tilde{\Theta}_{\Delta f} \subseteq \Theta_{\Delta fd} \subseteq \Theta_{\Delta f},$$

$$\forall e_1, e_2 \in E \lambda(e_1) \Xi_{\Delta f} \lambda(e_2)$$

$$\& \ \rho(e_1) \neg \Xi_{\Delta f} \rho(e_2) \Rightarrow e1 \neg \Theta_{\Delta fd} e2.$$

(The latter one is a criterion of $\Theta_{\Delta fd}$-determinism.)

All such equivalences $\Theta_{\Delta fd}$ are obtained using all possible subdecompositions of the sets of $\Theta^*_{\Delta f}$-equivalent factor arcs originating from one and the same factor vertex. The corresponding factor graphs will be deterministic and completely definite.

8. AN EXAMPLE OF AUTOMATON STATE GRAPH AND ITS FACTOR GRAPHS

By way of example, we consider the booking of airplane tickets. Two operations are defined: *ticket sale* and *ticket return.*

The *ticket sale* operation has one input parameter—flight number—and returns the number of an available ticket for this flight. In case all tickets for the requested flight are sold out, a ticket to the same destination for a later flight can be issued. The operation has a precondition: there are tickets for the requested or latter flights available. If there are several tickets satisfying the request, the operation is nondeterministic, as it is unspecified exactly which ticket will be sold.

The *ticket return* operation has one parameter—the number of the ticket being returned—and the precondition: the ticket must be one of the earlier sold tickets. As a result of performing this operation, the ticket goes on sale.

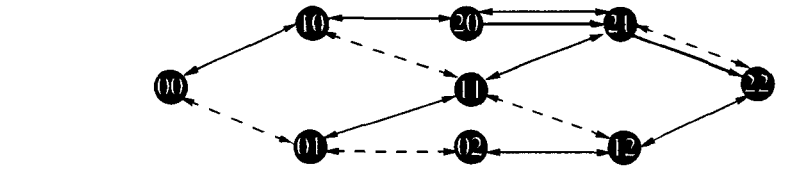
For the convenience of discussion, we restrict ourselves to the case of two flights to the same destination and two-seater airplanes. Tickets are numbered 0 and 1 for flight 1; 2 and 3, for flight 2. The graph of states, G, is shown in Fig. 9. This graph is Δ-nondeterministic.

The set M_Ξ consists of three equivalences: $\Xi_9 = \Xi_{min}$ (nine factor states), $\Xi_1 = \Xi_{max}$ (all states are equivalent—one factor state), and an intermediate equivalence, Ξ_5 (five factor states) that is obtained from Ξ_9 by merging classes shown in one vertical line in Fig. 9.

The set $M_\Theta(\Xi_1)$ consists of the only equivalence of arcs, $\Theta_1 = \Theta_{max}$, for which we do not distinguish operations (all arcs are equivalent; i.e., there is the only factor arc—a loop). This case of the deterministic and completely definite factor graph is not an interesting object for testing.

In the case when the initial equivalence of arcs, Ξ_5, distinguishes the operations (ticket sale or return) but does not distinguish subdomains of operations, the equivalence of states Θ_5 is constructed by Algorithm 2. The factor state is defined by the number of tickets sold (for both flights in total). The factor graph $G^*_5 = (V, E, \lambda, \rho, \Xi_5, \Theta_5)$ is deterministic and completely definite (Fig. 10).

The equivalence of states $\Xi_9 = \Xi_{min}$ is constructed by Algorithm 1. The generalized state is defined by the pair of numbers of tickets sold for flights 1 and 2. Ξ_9 is also constructed by Algorithm 2 if the initial equivalence of arcs, Θ_9, distinguishes the operations (ticket sale or return) and two subdomains of the sale operation. These subdomains are determined by the operation parameter (request for a ticket for flight 1 or 2). The factor graph $G^*_9 = (V, E, \lambda, \rho, \Xi_9, \Theta_9)$ is Δ-deterministic and completely definite; however, it is Θ_9-nondeterministic.The point is that the ticket return takes any factor state except for the initial one (no tickets sold) to different factor states, depending on the flight



Ticket sale for flight 1
Ticket return for flight 1
Ticket sale for flight 2
Ticket return for flight 2
Ticket sale for flight 2 upon the request for flight 1 (all tickets for flight 1 are sold out)
ij state in which *i* tickets for flight 1 and *j* tickets for flight 2 are sold

Factor graph $G_9^{*''} = (V, E, \lambda, \rho, \Xi_9, \Theta_9'')$

Fig. 11.

number of the ticket returned. It is natural to introduce a sub-equivalence $\Theta_9' \subseteq \Theta_9$ that would distinguish two subdomains of the ticket return operation: the return of ticket for flight 1 and for flight 2. In this case, the generalized alphabet consists of four symbols: ticket sale or return for one or another flight. The factor graph $G_9^{*'} = (V, E, \lambda, \rho, \Xi_9, \Theta_9')$ is deterministic and completely definite (Fig. 11). Consider the factor graph G_9'' corresponding to the pair (Ξ_9, Θ_9'') in which one more subdomain of the sale operation is distinguished: the sale of a ticket for flight 2 when a ticket for flight 1 was requested (when all tickets for the earlier flight 1 are sold out). This factor graph is also deterministic and completely definite.

9. COMPARISON WITH ANALOGOUS STUDIES

The problem of finite automaton testing has been under study for a long time; it has even been included in textbooks [2]. But the applications of different generalizations of the state graph for testing OOP classes have only come under study in recent years [1, 4, 5, 8]. In these papers, the concepts of the "abstract model of finite automaton" are used. Homomorphic graphs (factor graphs, in particular) are a special case of such abstract models.

In [8], the methodology of OOP class testing using the class model as a finite automaton is expounded in detail. After the analysis of traditional testing methods, the concepts of a finite automaton, its states and transitions, are introduced. An example is discussed that shows how the states of an abstract automaton are extracted from the description of a C++ class and how transition functions are described as scenarios of the dynamic change of state. Further, stages of generation of test sequences on the basis of the descriptions obtained are examined. Prototypes of tools for test sequence generation and test run are described very briefly. They are test script files compilers rather than their automatic generators. That paper can serve as a good introductory textbook.

In [4], a tool (called ClassBench) for the automatic construction of a traversal of a finite automaton state graph is described. For ClassBench, it is necessary to create an explicit description of the automaton—all vertices of the state graph and all its arcs. All loops (arcs with coinciding initial and terminal vertices) are described as a part of the corresponding vertex. They are passed through upon every passage through the vertex. In addition to this description, the classes *Driver* and *Oracle* are created. The class *Driver* provides for the execution of the testing itself. The following methods are declared in this class: *reset* (resetting the automaton to its initial state), *transit* (transition from the current state to a new one through an arc specified), and *node* (performs all the required actions upon the arrival to the current node (passes through loops)). During the performance of transitions and passages through loops, the class *Driver* invokes methods of the object of the class being tested and uses the class *Oracle* to verify the correctness of the work of the object. ClassBench provides the testing for different test coverage criteria: coverage of all states, arcs, or paths. It is provided with auxiliary means for creating the automaton description and the implementation of the classes *Driver* and *Oracle*.

A technique of an abstract finite automaton construction using formal specifications of a class written in the Object-Z language is described in [5]; this study is closely related to [4]. The automaton description is constructed in accordance with the requirements of the tool used (ClassBench).

In [1, 3], the toolkit implementing the testing by the algorithm $\mathbb{A}^*(A, \varnothing)$ and automating the creation of the corresponding generator of test sequences is described. Only the following test components are created manually: the mapping function real states onto abstract states, the generalized input symbol iterator, and the symbol calculation function. All the remaining parts are generated automatically from formal specifications in the RAISE (RSL) language. It is important to men-

tion that [4, 5, 8] are of a theoretical and experimental nature, whereas [1, 3] describe a practical result—a technology, applied in the process of the development and verification of a large software project.

The present paper elaborates the line of investigations suggested in [1, 3]. It provides a rigorous treatment of empirical solutions found in the process of the implementation of real software projects. The technique considered here differs from that proposed in the theoretical papers mentioned above in three basic positions. The first difference is in the approach to the solution of the arc adjacency problem, which was mentioned above. Because of this difference, in ClassBench, generally speaking, a path in the real graph corresponds to a passage through a single arc of the abstract graph; in our algorithms, this is a passage through exactly one arc of the real graph. On the one hand, this difference leads to a difference in the testing time estimates; on the other hand, it results in different requirements to the relationship between the real and the abstract graphs. In particular, the abstract graph for ClassBench can be non-homomorphic to the real one.

The second difference of our study is the opportunity of constructing the graph traversal and the graph itself in the process of testing; i.e., no explicit description of the graph is required. All information about the automaton is grouped in the oracle, which is generated automatically using specifications. The correspondence of the abstract and the real graphs is defined by the most economical way—by the reflection of real states into abstract states (or a predicate of the equivalence of states), a generalized input symbol iterator, and a symbol calculation function. It is also important to mention that no special reset operation is required for our algorithms; only the strong connectedness of the graph is required.

Finally, the third difference is as follows: the application of the technique involved in nondeterministic automatons is not described in [4, 5].

10. CONCLUSION

A natural line of further investigations is a formalization of the extraction process from formal specifications of the information necessary for the generation of the components that are currently created manually. The formalization should result, on the one hand, in formal requirements (methodology) to the form of specifications and, on the other hand, in a toolkit (technology) for automatic generation of test set components from formal specifications. In particular, one can design tools for constructing deterministic and completely definite factor graphs (Algorithms 1 and 2).

Another line of investigations can be aimed at the use of other abstract models of automatons and their state graphs, in particular, models investigated in [4, 5]. We note here that, in practice, when manually implementing the test components discussed above, we frequently used methods that were not formalized in the present paper. In particular, sometimes a path, rather than an arc, of the real graph corresponded to an arc of the abstract graph. In some special cases, testing by the algorithm $\mathbb{A}^*(A, \varnothing)$ was successful using non-deterministic factor graphs. However, not all expedients that are useful in manual implementation can be implemented as a tool. Therefore, it seems to be important to analyze the accumulated experience in order to extract formalizable techniques and implement them in the form of tools.

ACKNOWLEDGMENTS

This work was supported by the Russian Foundation for Basic Research (project nos. 96-01-01277 and 99-01-00207).

REFERENCES

1. Wong, H., Barantsev, A., Burdonov, I., and Kosachev, A., Report on Test Generation Methodology, *NORTEL*, 1997.
2. Beizer, B., *Software Testing Techniques*, New York: Van Nostrand Reinhold, 1990, 2nd edition.
3. Burdonov, I., Kosachev, A., Petrenko, A., Cheng, S., and Wong, H., Formal Specification and Verification of SOS Kernel, *BNR/NORTEL Design Forum*, 1996.
4. Hoffman, D. and Strooper, P., ClassBench: a Framework for Automated Class Testing, *Software Maintenance: Practice and Experience*, 1997, vol. 27, no. 5, pp. 573–579.
5. Murray, L., Carrington, D., MacColl, I., McDonald, J., and Strooper, P., Formal Derivation of Finite State Machines for Class Testing, *ZUM'98: The Z Formal Specification Notation, 11th Int. Conf. of Z Users*, Bowen, J.P., Fett, A., and Hinchey, M.G., Eds., *Lect. Notes Comput. Sci.*, Springer, 1998, vol. 1493, pp. 42–59.
6. Peters, D.K and Parnas, D.L., Using Test Oracles Generated from Program Documentation, *IEEE Trans. Software Eng.*, 1998, vol. 24, no. 3, pp. 161–173.
7. Petrenko, A.K., Burdonov, I.B., Drojjina, A.Yu., Kossatchev, A.S., Maximov, A.V., Sazanov, Yu.L., and Sumar, H., Preliminary Test Methodology and Test System Report, *NORTEL*, 1995.
8. Turner, C.D. and Robson, D.J., The State-based Testing of Object-Oriented Programs, *Proc. IEEE Conf. Software Maintenance*, 1993, pp. 302–310.